Go on DE1-SoC

Final Report

Maximilian Comfere (mkc2182)   Owen Cooper (odc2106)

1. Introduction

This document describes the design and implementation of our 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 and thus has been used in computer science and specifically AI (alpha go) as a baseline for performance. Its rules are 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 and Player vs. Computer modes, selecting from three difficulty levels ranging from a random-move beginner to an AI player utilizing Monte Carlo Tree Search.

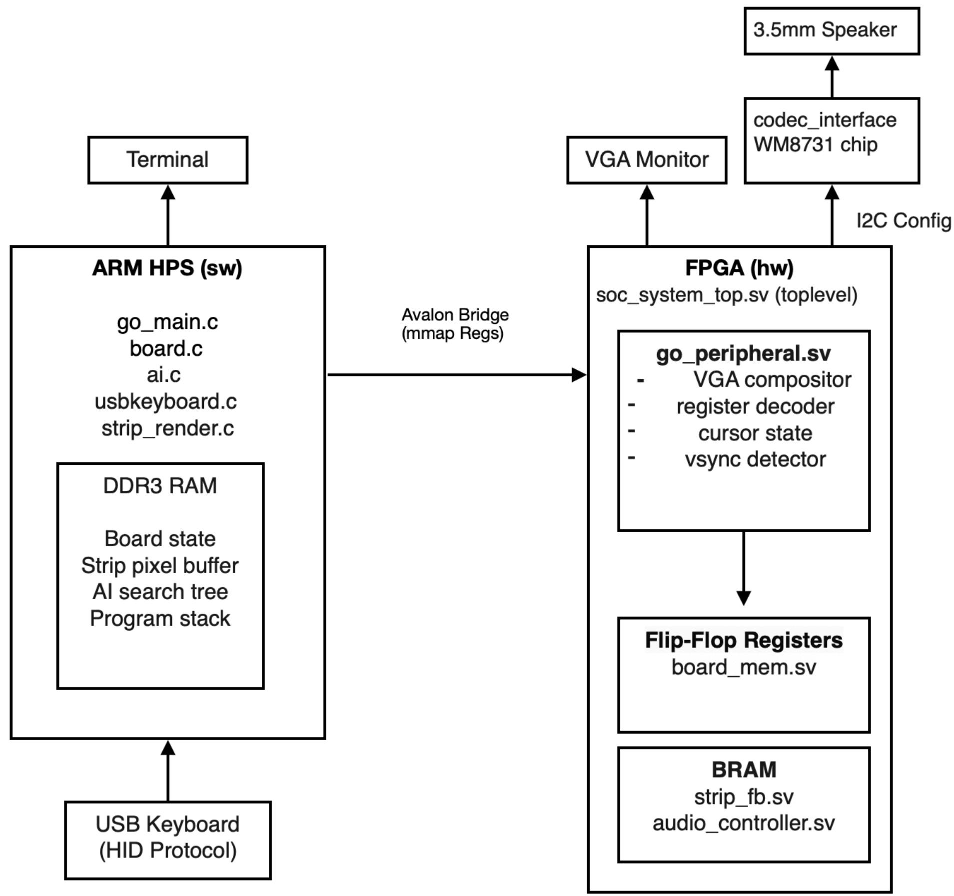
Design Decisions

FPGA (hardware): Handles VGA signal generation, board rendering, and audio playback. The board is rendered entirely in hardware using combinational logic for each pixel driven by hcount/vcount scan counters. The FPGA computes stone positions, grid lines, cursor rings, and star points for every pixel every frame. Audio playback runs from on-chip ROM (in BRAM) with zero ARM involvement after the initial trigger. A 640×60 double-buffered score strip framebuffer in M10K BRAM handles the score panel at the bottom of the screen.

ARM HPS (software): Game logic, AI, USB input handling, and score strip rendering run on the HPS. The rule engine requires recursive flood-fill for liberty counting and capture detection. The score strip renderer draws text and UI elements into a local DDR3 buffer which is bulk-written to the FPGA's strip framebuffer each frame via the avalon bridge.

Communication: The HPS writes to FPGA registers through memory-mapped I/O on the Avalon Lightweight HPS-to-FPGA bridge (base address 0xFF200000). Two Avalon slaves are used: slave 0 for byte-wide control registers (stone placement, cursor, audio, strip swap) and slave 1 for 32-bit bulk writes to the strip framebuffer back buffer.

2. System Block Diagram



Block Descriptions

Module	Layer	Description
usbkeyboard.c	SW	Uses libusb to poll USB keyboard via libusb_interrupt_transfer. Reads 8-byte HID boot-protocol reports. Decodes keycodes into game actions.
go_main.c	SW	UI state machine (TITLE -> MODE_SELECT -> DIFFICULTY -> GAME). Reads keyboard input, enforces game flow, commands the FPGA through register writes.
board.c	SW	Maintains BoardState in DDR3. Validates moves, counts liberties via flood fill, detects captures, checks ko via Zobrist hash, scores territory.
ai.c	SW	Three AI levels: random, greedy heuristic, UCT-MCTS with 200 simulations. All run on ARM HPS in C.
strip_render.c	SW	Draws text and shapes into a local 38,400-byte DDR3 buffer using a 5×7 bitmap font. strip_present() bulk-writes 9,600 32-bit words to the FPGA back buffer and arms a vsync swap.
go_peripheral.sv	HW	Decodes Avalon slave 0 register writes, drives combinational VGA pixel compositor, instantiates all sub-modules.

Module	Layer	Description
vga_counters	HW	Generates hcount/vcount scan counters and VGA timing signals (HS, VS, BLANK, SYNC, CLK) at 640×480 60Hz. Pixel clock derived from hcount[0].
board_mem.sv	HW	81-cell × 2-bit flip-flop register array. Synchronous write from Avalon decoder, combinational zero-latency read to VGA compositor.
strip_fb.sv	HW	640×60 8 bits per peixel double-buffered framebuffer in M10K BRAM. ARM writes to the back buffer; swaps at vsync rising edge.
audio_controller.sv	HW	Four 16-bit PCM ROM banks loaded via $readmemh. Zero-order-hold upsamples 8kHz to 48kHz by holding each sample 6 advance pulses.
codec_interface	HW	Configures WM8731 over I2C at startup and streams 24-bit samples over I2S.

3. Algorithms

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:** flood-fill liberty count on the placed group. If liberties == 0 and no opponent group was captured, the move is suicide, then reject it.
3. **Ko check:** compare the resulting Zobrist board hash against prev_board_hash. If they match the move recreates the immediately previous position, then reject it as a ko violation.

If all checks pass the move is applied: opponent groups with zero liberties are removed, capture counts updated, hash updated, turn toggled. Passing is always legal; two consecutive passes end the game.

Liberty Counting via Flood Fill

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

Ko detection via Zobrist hashing: A 9×9×2 table of random 64-bit integers initialized once at startup. The board hash is the XOR of all entries for occupied cells. Incremental O(1) updates: XOR in the changed cell's entry on each place or remove.

3.2 Scoring (Software)

At game end board_score() computes Chinese-style area scoring: stones on board plus enclosed empty intersections. Territory determined by flood-filling each empty region and checking which player's stones form the complete boundary. Contested intersections are neutral. White receives 5.5 komi (score metric).

3.3 Board Rendering (Hardware)

The board is rendered entirely in the FPGA using combinational logic. No framebuffer exists for the board area. For every pixel the VGA scanner visits inside the board region, go_peripheral evaluates in a single combinational chain:

```
col      = (px - 131) / 42
row      = (py - 24)  / 42
cell_idx = row * 9 + col
local_x  = (px - 131) - col * 42
local_y  = (py - 24)  - row * 42
dx = local_x - 21,  dy = local_y - 21
d2 = dx² + dy²
cell_value = board_mem[cell_idx]   // combinational, zero latency

Priority:
  on_cursor  → d2 ∈ [264,400] AND cursor_idx == cell_idx  → green
  in_stone   → cell_value ≠ 0 AND d2 ≤ 324                → black or white
  on_outline → cell_value == WHITE AND d2 ∈ [289,324]      → black outline
  in_star    → star cell AND d2 ≤ 9                        → black dot
  on_grid    → local_x == 21 OR local_y == 21              → black line
  default                                                  → burlywood
```

The result drives VGA_R, VGA_G, VGA_B directly in the same clock cycle.

3.4 Score Strip Rendering (Software + Hardware)

strip_render.c draws into a local 38,400-byte C array in DDR3 using a 5×7 bitmap font. Each pixel is stored as an 8-bit palette index (bottom 3 bits used). When a frame is ready strip_present() bulk-writes 9,600 32-bit words (4 pixels packed per word) to the FPGA strip framebuffer back buffer via Avalon slave 1, then writes REG_STRIP_SWAP. The FPGA latches the swap request and fires it on the next vsync rising edge.

3.5 Audio Playback (Hardware)

Four ROM banks in M10K BRAM, loaded at synthesis from .vh hex files via $readmemh. Each bank stores 16-bit signed mono PCM at 8kHz. The codec_interface advance strobe fires at 48kHz. Each ROM sample is held for 6 advance pulses (zero-order-hold upsample, 8kHz × 6 = 48kHz). State machine:

```
S_IDLE: on audio_cmd_valid → load ROM bank, reset counters → S_PLAY
S_PLAY: on advance pulse:
    if sub_count < 5 → sub_count++   (hold sample)
    else → sub_count = 0, sample_idx++
    if sample_idx >= rom_length → S_IDLE

16-bit samples sign-extended to 24 bits for codec_interface.
```

3.6 AI - Level 1: Random

Builds list of all legal intersections, picks one uniformly at random. If none exist, passes. $O(N^2)$.

3.7 AI - Level 2: Greedy

Evaluates each legal move with a priority score. Highest priority selected; ties broken randomly. $O(N^4)$.

Priority	Condition
P3	Move reduces opponent group to 0 liberties (+10 per captured stone)
P2	Move rescues friendly group in atari (+8)
P1	Move places opponent group in atari (+2 per liberty reduced)
P0	Random legal move with center bias (+0–3)

3.8 AI - Level 3: MCTS

UCT-MCTS with 200 simulations per move. UCB1: wins/visits + 1.41 × √(ln(parent_visits)/visits). Each simulation: select by UCB1, expand, random playout to max depth $4 \times N^2 = 324$, backpropagate win/loss. Returns the child of the root with the highest visit count. Completes in approximately 1–2 seconds on ARM Cortex-A9.

4. Hardware / Software Interface

Two Avalon slaves exposed to the ARM. Slave 0 is byte-addressed for control registers. Slave 1 is 32-bit word-addressed for strip framebuffer bulk writes.

4.1 Register Map

Slave 0 - Control Registers (base 0xFF200000)

Offset	Name	R/W	Bits	Description
`0x00`	`SET_BLACK`	W	`[6:0]`	Place black stone at cell index
`0x01`	`SET_WHITE`	W	`[6:0]`	Place white stone at cell index
`0x02`	`CLEAR_CELL`	W	`[6:0]`	Remove stone at cell index
`0x03`	`RESET_BOARD`	W	`any`	Clear all 81 cells to empty
`0x04`	`CURSOR`	W	`[7]: visible, [6:0]: cell_idx`	Move/hide cursor ring
`0x05`	`STRIP_SWAP`	W	`any`	Arm strip buffer swap at next vsync
`0x06`	`AUDIO_CMD`	W	`[2:0]`	Trigger sound: 1=place 2=capture 3=illegal 4=gameover
`0x07`	`AUDIO_STATUS`	R	`[0]`	1 = audio currently playing

Slave 1 - Strip Framebuffer (base 0xFF210000)

Address	Description
0x00000 – 0x09600	9,600 × 32-bit words — BACK buffer pixels (4 packed 8-bit palette indices per word)

Cell index = row × 9 + col, range 0–80.

4.2 Userspace Program

Keyboard input: usbkeyboard.c calls libusb_interrupt_transfer with a 10ms timeout. Returns 8-byte HID boot-protocol packet. go_main.c reads pkt.keycode[0] and debounces by comparing to previous keycode.

Board updates: After each valid move go_main.c calls wr8(REG_SET_BLACK/WHITE, cell_idx), a single byte write per cell. After AI moves hw_push_board() syncs all 81 cells.

Strip display: strip_render.c draws into local DDR3 buffer, strip_present() bulk-writes to slave 1, then wr8(REG_STRIP_SWAP, 1) arms the vsync swap.

Audio: wr8(REG_AUDIO_CMD, cmd), one byte write. FPGA handles playback independently.

4.3 Hardware Modules

- go_peripheral (top): Two Avalon slaves. Decodes slave 0 register writes. Routes slave 1 writes to strip_fb. Runs combinational VGA pixel compositor. Detects vsync rising edge for strip swap timing.
- vga_counters: Generates hcount [10:0] and vcount [9:0] from 50MHz clock. Pixel x = hcount[10:1], pixel y = vcount. Outputs HS, VS, BLANK_n, SYNC_n, CLK.
- board_mem: 81 × 2-bit flip-flop registers. Synchronous write port (one cell per clock), combinational read port (zero latency). Reset clears all 81 cells in one cycle.
- strip_fb: Two M10K BRAM banks (buf_a, buf_b), 9,600 × 32-bit each. The active flag determines front/back. Swap armed by swap_request, fires on vsync_pulse.
- audio_controller: Four ROM banks via $readmemh. Two-state FSM (IDLE/PLAY). Zero-order-hold upsampling. 24-bit sign-extended output to codec_interface.
- codec_interface (Altera-UP, toplevel): Configures WM8731 over I2C at startup. Streams 24-bit stereo PCM over I2S at 48kHz. Generates advance strobe consumed by audio_controller.

4.4 UI State Machine (Software)

```
UI_TITLE → (ENTER) → UI_MODE_SELECT → (ENTER, PvP) → UI_GAME
                                    → (ENTER, PvC) → UI_DIFFICULTY → UI_GAME
ESC goes back one state.  R in UI_GAME returns to UI_TITLE.
```

4.5 Write Sequence: Processing a Move

```
1. board_place(&board, row, col)        // validate + update BoardState in DDR3
   → if illegal: wr8(REG_AUDIO_CMD, AUDIO_ILLEGAL); return
2. wr8(REG_SET_BLACK, cell_idx)         // sync display copy to FPGA board_mem
   (or SET_WHITE / CLEAR_CELL for captures)
3. wr8(REG_CURSOR, 0x80 | cell_idx)    // keep cursor visible at same position
4. strip_render draws new score strip   // update local DDR3 buffer
   strip_present()                      // bulk-write to FPGA back buffer + arm swap
5. wr8(REG_AUDIO_CMD, AUDIO_PLACE      // trigger sound
        or AUDIO_CAPTURE)
6. (PvC only) ai_get_move() → repeat steps 1-5 for AI response
```

5. Resource Budgets

5.1 On-chip BRAM (FPGA Cyclone V M10K)

Block	Size	Contents
Strip front buffer	37,500 B	640×60 pixels, 8bpp, active display
Strip back buffer	37,500 B	640×60 pixels, 8bpp, ARM writes here
Audio ROM — place	3,200 B	1,600 samples, 16-bit, 8kHz, 0.2s
Audio ROM — capture	4,800 B	2,400 samples, 0.3s
Audio ROM — illegal	2,400 B	1,200 samples, 0.15s
Audio ROM — game over	24,000 B	12,000 samples, 1.5s
Total on-chip	~109 KB	Well within Cyclone V capacity (~500 KB)

5.2 Flip-Flop Registers (FPGA)

Block	Size	Contents
board_mem cells	162 bits	81 × 2-bit stone values
cursor_idx + cursor_visible	8 bits	Cursor state
vs_d, swap_armed, active	3 bits	Vsync edge detect + swap state

5.3 DDR3 RAM (HPS)

Data	Size
BoardState struct	~200 bytes
Strip local pixel buffer	38,400 bytes
Zobrist table (9×9×2 × 8 bytes)	1,296 bytes
MCTS tree (200 simulations)	~400 KB
Program stack + OS	~128 MB

5.4 Avalon Bus Bandwidth

Transaction	Size	Frequency
Single cell update	8 bits	Once per move
Full board push	81 × 8 = 648 bits	Once per AI move

Transaction	Size	Frequency
Strip bulk write	9,600 × 32 = 307,200 bits	Once per frame (~770 μs)
Audio trigger	8 bits	On game events
Audio status read	8 bits	Optional polling

6. Verilog Module Interfaces

go_peripheral (top level)

```
module go_peripheral (
    input  logic        clk, reset,
    // Avalon slave 0 — control registers (byte-wide)
    input  logic [2:0]  address,
    input  logic        chipselect,
    input  logic        write,
    input  logic [7:0]  writedata,
    output logic [7:0]  readdata,
    // Avalon slave 1 — strip framebuffer (32-bit word-wide)
    input  logic [13:0] strip_address,
    input  logic        strip_chipselect,
    input  logic        strip_write,
    input  logic [31:0] strip_writedata,
    // VGA conduit
    output logic [7:0]  VGA_R, VGA_G, VGA_B,
    output logic        VGA_CLK, VGA_HS, VGA_VS,
                        VGA_BLANK_n, VGA_SYNC_n,
    // Audio conduit
    output logic [23:0] dac_left, dac_right,
    input  logic        advance
);
```

board_mem

```
module board_mem (
    input  logic        clk, reset,
    input  logic        write_en,
    input  logic [6:0]  write_addr,   // cell index 0..80
    input  logic [1:0]  write_data,   // 00=empty 01=black 10=white
    input  logic        reset_all,
    input  logic [6:0]  read_addr,
    output logic [1:0]  read_data     // combinational
);
```

strip_fb

```
module strip_fb (
    input  logic        clk, reset,
    input  logic        write_en,
    input  logic [13:0] write_addr,   // 0..9599 word index
    input  logic [31:0] write_data,
    input  logic        swap_request,
    input  logic        vsync_pulse,
    input  logic [15:0] pixel_addr,   // 0..38399
```

```
    output logic [7:0]  pixel_out
);
```

audio_controller

```
module audio_controller (
    input  logic        clk, reset,
    input  logic [2:0]  audio_cmd,
    input  logic        audio_cmd_valid,
    output logic        busy,
    input  logic        advance,
    output logic [23:0] dac_left, dac_right
);
```

vga_counters

```
module vga_counters (
    input  logic        clk50, reset,
    output logic [10:0] hcount,
    output logic [9:0]  vcount,
    output logic        VGA_CLK, VGA_HS, VGA_VS,
                        VGA_BLANK_n, VGA_SYNC_n
);
```

7. C Software Interfaces

Board.h - Board State and Rule Engine

```
typedef enum { EMPTY = 0, BLACK = 1, WHITE = 2 } Stone;
typedef enum {
    MOVE_OK,
    MOVE_ILLEGAL_OCCUPIED,
    MOVE_ILLEGAL_SUICIDE,
    MOVE_ILLEGAL_KO
} 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);
void       board_score(const BoardState *b, int *black, int *white);
```

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;
```

```
Move ai_get_move(const BoardState *b, AiLevel level);
void ai_seed(uint32_t seed);
```

strip_render.h — Score Strip Renderer

```
void strip_init(volatile uint8_t *lw_bridge_base,
                uint32_t strip_offset,
                uint32_t reg_strip_swap_offset);

void strip_clear(uint8_t color);
void strip_text(int x, int y, const char *s, int scale, uint8_t fg, uint8_t bg);
void strip_text_centered(int y, const char *s, int scale, uint8_t fg, uint8_t bg);
void strip_text_box(int x, int y, const char *s, int scale,
                    uint8_t fg, uint8_t bg, int pad);
void strip_present(void);   // bulk-write to FPGA + arm vsync swap
```

usbkeyboard.h - USB HID Input

```
struct usb_keyboard_packet {
    uint8_t modifiers;
    uint8_t reserved;
    uint8_t keycode[6];
};

struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address);
```

Hardware Register Access

```
#define LW_BRIDGE_BASE       0xFF200000UL
#define GO_PERIPHERAL_OFFSET 0x0000
#define STRIP_FB_OFFSET      0x10000

#define REG_SET_BLACK    0x00
#define REG_SET_WHITE    0x01
#define REG_CLEAR_CELL   0x02
#define REG_RESET_BOARD  0x03
#define REG_CURSOR       0x04
#define REG_STRIP_SWAP   0x05
#define REG_AUDIO_CMD    0x06
#define REG_AUDIO_STATUS 0x07

static inline void wr8(uint32_t off, uint8_t v) { go_regs[off] = v; }
```

8. Game Rules and Mechanics

Stone Placement: Black and white alternate placing stones on 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. Captured stones are removed immediately and added to the opponent's capture count.

Ko Rule: A move that would recreate the immediately previous board position is forbidden. Detected via 64-bit Zobrist hash comparison against prev_board_hash.

Suicide Rule: A move that would leave the placed group with zero liberties, without capturing an opponent group in the same move, is illegal.

Passing and End of Game: A player may pass their turn at any time. The game ends when both players pass consecutively (consecutive_passes == 2).

Scoring: Chinese-style area scoring: stones on board plus surrounded empty intersections. White receives 5.5 komi. Winner determined at game end; displayed in score strip.

9. Game Modes

Mode	Description
Player vs. Player	Two players share one USB keyboard. Cursor navigated with arrow keys; Enter places stone, Space passes, R returns to menu, Esc quits.
PvC — Level 1 (Random)	AI picks uniformly random legal move. Suitable for learning rules.
PvC — Level 2 (Greedy)	AI uses heuristic priority: captures > defends atari > reduces opponent liberties > center bias.
PvC — Level 3 (MCTS)	UCT-MCTS, 200 simulations, max rollout depth 324. Provides a legitimately challenging opponent.

CLI shortcut: ./go_main 0 = PvP, ./go_main 1/2/3 = PvC at that level, skipping menus.

10. CODEC Interface

The WM8731 audio codec is configured at startup by the codec_interface Altera-UP core via I2C. Configuration is automatic and happens before any software runs. After configuration the codec accepts 24-bit stereo PCM samples over I2S at 48kHz.

Register	Value	Function
Left HP out	0x079	Headphone volume
Right HP out	0x079	Headphone volume
Analog path	0x010	DAC select, mic mute
Digital path	0x000	No de-emphasis, no mute
Power down	0x000	All blocks powered on
Format	0x001	I2S, 16-bit, slave mode
Sampling	0x002	Normal mode, 8kHz
Active	0x001	Activate interface

audio_controller feeds 24-bit samples (16-bit ROM values sign-extended) to codec_interface on each advance pulse. The advance strobe is generated by codec_interface at 48kHz.

Appendix A - Verilog Module Hierarchy

```
soc_system_top.sv  (board toplevel — pin assignments)
├─ soc_system  (Qsys-generated — ARM HPS + Avalon bus)
│   └─ go_peripheral.sv  (custom IP)
│       ├─ vga_counters  (hcount/vcount, VGA timing)
│       ├─ board_mem.sv  (81-cell flip-flop register array)
│       ├─ strip_fb.sv   (640×60 double-buffered M10K BRAM)
│       └─ audio_controller.sv  (ROM playback FSM)
└─ codec_interface  (Altera-UP audio core, NOT in Qsys)
    ├─ Altera_UP_I2C_AV_Auto_Initialize.v
    ├─ Altera_UP_Audio_Out_Serializer.v
    └─ xck_generator.v
```

Appendix B - 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 / confirm menu
0x2C	Space	Pass turn
0x29	Escape	Back / quit
0x15	R	Restart / return to menu

Appendix C - Code Files

Software

go_main.c

```
C/C++
/*
 * go_main.c — Phase 8: full game flow
 *
 * UI state machine:
 *   TITLE          → "GO 9x9 — Press ENTER to start"
 *   MODE_SELECT    → PvP / PvC (left/right + Enter)
 *   DIFFICULTY_SEL → Level 1 / 2 / 3 (PvC only)
 *   GAME           → live game, Phase 4–7 logic, score panel updates
 *
 * Esc quits from any state. R restarts back to TITLE from GAME / GAME_OVER.
 *
 * argv shortcut: ./go_main [N]
 *   N absent     → start at TITLE
```

```c
 *   N=1/2/3      → skip menus; PvC at that level
 *   N=0          → skip menus; PvP
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

#include "board.h"
#include "ai.h"
#include "usbkeyboard.h"
#include "strip_render.h"
#include <time.h>

/* ─── HW interface ───────────────────────────────────────────────── */

#define LW_BRIDGE_BASE   0xFF200000UL
#define LW_BRIDGE_SPAN   0x00200000UL
#define GO_PERIPHERAL_OFFSET  0x0000        /* avalon_slave_0 */
#define STRIP_FB_OFFSET       0x10000       /* avalon_slave_1 (set in Qsys) */

#define REG_SET_BLACK     0x00
#define REG_SET_WHITE     0x01
#define REG_CLEAR_CELL    0x02
#define REG_RESET_BOARD   0x03
#define REG_CURSOR        0x04
#define REG_STRIP_SWAP    0x05
#define REG_AUDIO_CMD     0x06
#define REG_AUDIO_STATUS  0x07

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

static volatile uint8_t *go_regs;
static volatile uint8_t *lw_base;     /* shared with strip_render */

static int hw_init(void)
{
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd < 0) { perror("open /dev/mem"); return -1; }
    void *base = mmap(NULL, LW_BRIDGE_SPAN, PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd, LW_BRIDGE_BASE);
    if (base == MAP_FAILED) { perror("mmap"); close(fd); return -1; }
    lw_base  = (volatile uint8_t *)base;
    go_regs  = lw_base + GO_PERIPHERAL_OFFSET;
    close(fd);
    return 0;
```

```c
}

static inline void wr8(uint32_t off, uint8_t v) { go_regs[off] = v; }
static inline int  cell_idx(int r, int c)       { return r * 9 + c; }

static void hw_reset_board(void) { wr8(REG_RESET_BOARD, 1); }

static inline void hw_play_audio(uint8_t cmd) { wr8(REG_AUDIO_CMD, cmd); }

static void hw_cursor(int row, int col, int visible)
{
    wr8(REG_CURSOR, (visible ? 0x80 : 0) | (cell_idx(row, col) & 0x7F));
}

/* Push the whole board state to FPGA tilemap. ~81 byte writes. */
static void hw_push_board(const BoardState *b)
{
    for (int r = 0; r < BOARD_N; r++)
        for (int c = 0; c < BOARD_N; c++) {
            int idx = cell_idx(r, c);
            switch (b->cells[r][c]) {
                case BLACK: wr8(REG_SET_BLACK,  idx); break;
                case WHITE: wr8(REG_SET_WHITE,  idx); break;
                case EMPTY: wr8(REG_CLEAR_CELL, idx); break;
            }
        }
}

/* ── HID keycodes ──────────────────────────────────────────────────────── */
#define KEY_RIGHT 0x4F
#define KEY_LEFT  0x50
#define KEY_DOWN  0x51
#define KEY_UP    0x52
#define KEY_ENTER 0x28
#define KEY_SPACE 0x2C
#define KEY_ESC   0x29
#define KEY_R     0x15

static int clamp(int v, int lo, int hi) { return v < lo ? lo : (v > hi ? hi : v); }

static const char *moveresult_str(MoveResult r)
{
    switch (r) {
    case MOVE_OK:                return "OK";
    case MOVE_ILLEGAL_OCCUPIED: return "occupied";
    case MOVE_ILLEGAL_SUICIDE:  return "suicide";
    case MOVE_ILLEGAL_KO:       return "ko";
    }
    return "?";
}

static const char *stone_str(Stone s)
{
    return s == BLACK ? "Black" : s == WHITE ? "White" : "?";
```

```c
}

/* ─── Menu screens ──────────────────────────────────────────────────────── */

/* Center one option of an N-option row at fixed slot `i` of `n` slots,
 * spaced evenly across the strip. Returns the x coordinate at which to
 * begin painting a glyph or box of width `item_w`. */
static int slot_x(int i, int n, int item_w)
{
    /* Slot center: STRIP_W * (i + 0.5) / n. */
    int center = (STRIP_W * (2 * i + 1)) / (2 * n);
    return center - item_w / 2;
}

static void render_title(void)
{
    strip_clear(COLOR_STRIP_BG);
    strip_text_centered(2,  "GO 9X9",      5, COLOR_STRIP_GOLD,  COLOR_STRIP_BG);
    strip_text_centered(46, "PRESS ENTER", 2, COLOR_STRIP_WHITE, COLOR_STRIP_BG);
    strip_present();
}

/* Paint one menu option at slot (i,n). When `selected`, draws a filled gold
 * pill (dark text on gold bg); otherwise plain white-on-bg. */
static void draw_menu_option(int i, int n, int y, const char *label,
                             int scale, int selected)
{
    int pad = 4;
    int w   = strip_text_width(label, scale) + 2 * pad;
    int x   = slot_x(i, n, w);
    if (selected) {
        strip_text_box(x, y, label, scale,
                       COLOR_STRIP_BG, COLOR_STRIP_GOLD, pad);
    } else {
        strip_text(x + pad, y + pad, label, scale,
                   COLOR_STRIP_WHITE, COLOR_STRIP_BG);
    }
}

static void render_mode_select(int sel /* 0=PvP, 1=PvC */)
{
    strip_clear(COLOR_STRIP_BG);
    strip_text_centered(2, "SELECT MODE", 2,
                        COLOR_STRIP_WHITE, COLOR_STRIP_BG);
    draw_menu_option(0, 2, 18, "PVP", 3, sel == 0);
    draw_menu_option(1, 2, 18, "PVC", 3, sel == 1);
    strip_text_centered(52, "<- -> SELECT   ENTER OK   ESC BACK", 1,
                        COLOR_STRIP_GRAY, COLOR_STRIP_BG);
    strip_present();
}

static void render_difficulty(int sel /* 0..2 */)
{
    strip_clear(COLOR_STRIP_BG);
```

```
    strip_text_centered(2, "AI DIFFICULTY", 2,
                        COLOR_STRIP_WHITE, COLOR_STRIP_BG);
    const char *labels[3] = { "L1", "L2", "L3" };
    for (int i = 0; i < 3; i++)
        draw_menu_option(i, 3, 18, labels[i], 3, sel == i);
    strip_text_centered(52, "<- -> SELECT   ENTER OK   ESC BACK", 1,
                        COLOR_STRIP_GRAY, COLOR_STRIP_BG);
    strip_present();
}

static void render_panel(const BoardState *b)
{
    char buf[64];
    strip_clear(COLOR_STRIP_BG);

    if (b->game_over) {
        int blk, wht;
        board_score(b, &blk, &wht);
        double w_total = wht + 5.5;
        const char *winner = (blk > w_total) ? "BLACK" : "WHITE";

        strip_text_centered(2, "GAME OVER", 3,
                            COLOR_STRIP_GOLD, COLOR_STRIP_BG);
        snprintf(buf, sizeof(buf),
                 "BLACK %d   WHITE %d+5.5   WINNER %s",
                 blk, wht, winner);
        strip_text_centered(30, buf, 2,
                            COLOR_STRIP_WHITE, COLOR_STRIP_BG);
        strip_text_centered(52, "PRESS R NEW GAME   ESC QUIT", 1,
                            COLOR_STRIP_GRAY, COLOR_STRIP_BG);
    } else {
        /* Row 1 (y=4): turn indicator, prominent. */
        snprintf(buf, sizeof(buf), "TURN  %s",
                 b->turn == BLACK ? "BLACK" : "WHITE");
        strip_text_centered(4, buf, 2,
                            b->turn == BLACK ? COLOR_STRIP_WHITE
                                             : COLOR_STRIP_BURLY,
                            COLOR_STRIP_BG);

        /* Row 2 (y=26): captures + pass indicator. */
        if (b->consecutive_passes == 1) {
            snprintf(buf, sizeof(buf),
                     "CAPTURED  B %d   W %d     1 PASS",
                     b->captured_white, b->captured_black);
        } else {
            snprintf(buf, sizeof(buf),
                     "CAPTURED  B %d   W %d",
                     b->captured_white, b->captured_black);
        }
        strip_text_centered(26, buf, 2,
                            COLOR_STRIP_GRAY, COLOR_STRIP_BG);

        /* Row 3 (y=50): controls hint. */
        strip_text_centered(50,
```

```
            "ENTER PLACE   SPACE PASS   R MENU   ESC QUIT", 1,
            COLOR_STRIP_GRAY, COLOR_STRIP_BG);
    }

    strip_present();
}

/* Apply an AI move to the BoardState; sync HW; print log. */
static void apply_ai_move(BoardState *b, AiLevel level,
                          int cursor_row, int cursor_col)
{
    Stone moved = b->turn;
    int prev_caps = b->captured_black + b->captured_white;
    Move m = ai_get_move(b, level);
    if (m.pass) {
        board_pass(b);
        printf("AI (%s) passes.\n", stone_str(moved));
    } else {
        MoveResult r = board_place(b, m.row, m.col);
        if (r != MOVE_OK) {
            /* AI returned an illegal move — shouldn't happen with our
             * legal_moves() generator, but be defensive: pass instead. */
            board_pass(b);
            printf("AI tried illegal (%d,%d), passes instead.\n", m.row, m.col);
        } else {
            int new_caps = b->captured_black + b->captured_white;
            hw_play_audio(new_caps > prev_caps ? AUDIO_CAPTURE : AUDIO_PLACE);
            printf("AI (%s) plays (%d,%d).\n", stone_str(moved), m.row, m.col);
        }
    }
    hw_push_board(b);
    hw_cursor(cursor_row, cursor_col, 1);
    render_panel(b);
}

typedef enum {
    UI_TITLE,
    UI_MODE_SELECT,
    UI_DIFFICULTY,
    UI_GAME,
} UiState;

/* Helpers to enter each state cleanly (resets HW + redraws strip). */
static void enter_title(void)
{
    hw_reset_board();
    hw_cursor(0, 0, 0);
    render_title();
}
static void enter_mode_select(int sel)
{
    hw_reset_board();
    hw_cursor(0, 0, 0);
    render_mode_select(sel);
```

```c
}
static void enter_difficulty(int sel)
{
    render_difficulty(sel);
}
static void enter_game(BoardState *b, int *crow, int *ccol)
{
    board_init(b);
    hw_reset_board();
    *crow = 4; *ccol = 4;
    hw_cursor(*crow, *ccol, 1);
    render_panel(b);
}

int main(int argc, char **argv)
{
    /* CLI shortcuts:
     *   no arg     → start at TITLE menu
     *   0          → skip menus, PvP
     *   1 / 2 / 3  → skip menus, PvC at level N */
    AiLevel ai_level    = 0;
    int     pvc         = 0;
    int     skip_menu   = 0;
    if (argc >= 2) {
        int n = atoi(argv[1]);
        if (n == 0)                   { skip_menu = 1; pvc = 0; }
        else if (n >= 1 && n <= 3)   { skip_menu = 1; pvc = 1; ai_level = (AiLevel)n; }
    }
    ai_seed((uint32_t)time(NULL));

    /* Open keyboard + map FPGA */
    uint8_t endpoint;
    struct libusb_device_handle *kbd = openkeyboard(&endpoint);
    if (!kbd) { fprintf(stderr, "No USB keyboard.\n"); return 1; }
    if (hw_init() != 0) { libusb_close(kbd); return 1; }
    strip_init(lw_base, STRIP_FB_OFFSET, GO_PERIPHERAL_OFFSET + REG_STRIP_SWAP);

    BoardState b;
    int cursor_row = 4, cursor_col = 4;
    int mode_sel = 0;          /* 0 = PvP, 1 = PvC */
    int diff_sel = 1;          /* 0..2 → level 1..3 */
    UiState ui;

    if (skip_menu) {
        ui = UI_GAME;
        enter_game(&b, &cursor_row, &cursor_col);
        printf(pvc
               ? "PvC (AI=White, level %d). Black (you) to play.\n"
               : "PvP. Black to play.\n",
               ai_level);
    } else {
        ui = UI_TITLE;
        enter_title();
        printf("Phase 8: title menu. Enter to start, Esc to quit.\n");
```

```c
    }

    struct usb_keyboard_packet pkt;
    int xferred;
    uint8_t prev_key = 0;

    for (;;) {
        int r = libusb_interrupt_transfer(kbd, endpoint,
                                          (unsigned char *)&pkt, sizeof(pkt),
                                          &xferred, 10);
        if (r != 0 && r != LIBUSB_ERROR_TIMEOUT) {
            fprintf(stderr, "libusb error: %d\n", r);
            break;
        }
        if (xferred != sizeof(pkt)) continue;

        uint8_t key = pkt.keycode[0];
        if (key == prev_key) continue;
        prev_key = key;
        if (key == 0) continue;

        if (key == KEY_ESC) {
            if (ui == UI_TITLE || ui == UI_GAME) goto done;
            if (ui == UI_MODE_SELECT) { ui = UI_TITLE; enter_title(); continue; }
            if (ui == UI_DIFFICULTY)  { ui = UI_MODE_SELECT;
                                        enter_mode_select(mode_sel); continue; }
        }

        switch (ui) {
        /* ─────────────── TITLE ─────────────── */
        case UI_TITLE:
            if (key == KEY_ENTER) {
                ui = UI_MODE_SELECT;
                enter_mode_select(mode_sel);
            }
            break;

        /* ─────────────── MODE_SELECT ─────────────── */
        case UI_MODE_SELECT:
            if (key == KEY_LEFT)  { mode_sel = 0; render_mode_select(mode_sel); }
            if (key == KEY_RIGHT) { mode_sel = 1; render_mode_select(mode_sel); }
            if (key == KEY_ENTER) {
                if (mode_sel == 0) {
                    pvc = 0; ai_level = 0;
                    ui = UI_GAME;
                    enter_game(&b, &cursor_row, &cursor_col);
                    printf("\nPvP. Black to play.\n");
                } else {
                    ui = UI_DIFFICULTY;
                    enter_difficulty(diff_sel);
                }
            }
            break;
```

```
        /* ─────────────── DIFFICULTY ─────────────── */
        case UI_DIFFICULTY:
            if (key == KEY_LEFT  && diff_sel > 0) { diff_sel--;
render_difficulty(diff_sel); }
            if (key == KEY_RIGHT && diff_sel < 2) { diff_sel++;
render_difficulty(diff_sel); }
            if (key == KEY_ENTER) {
                pvc = 1; ai_level = (AiLevel)(diff_sel + 1);
                ui = UI_GAME;
                enter_game(&b, &cursor_row, &cursor_col);
                printf("\nPvC (AI=White, level %d). Black to play.\n", ai_level);
            }
            break;

        /* ─────────────── GAME ─────────────── */
        case UI_GAME:
            switch (key) {
            case KEY_UP:    cursor_row = clamp(cursor_row - 1, 0, 8);
                            hw_cursor(cursor_row, cursor_col, 1); break;
            case KEY_DOWN:  cursor_row = clamp(cursor_row + 1, 0, 8);
                            hw_cursor(cursor_row, cursor_col, 1); break;
            case KEY_LEFT:  cursor_col = clamp(cursor_col - 1, 0, 8);
                            hw_cursor(cursor_row, cursor_col, 1); break;
            case KEY_RIGHT: cursor_col = clamp(cursor_col + 1, 0, 8);
                            hw_cursor(cursor_row, cursor_col, 1); break;
            case KEY_ENTER: {
                if (b.game_over) { printf("Game over — press R to return to menu.\n");
break; }
                if (pvc && b.turn == WHITE) {
                    printf("AI is thinking; please wait.\n");
                    break;
                }
                Stone moved = b.turn;
                int prev_caps = b.captured_black + b.captured_white;
                MoveResult mr = board_place(&b, cursor_row, cursor_col);
                if (mr == MOVE_OK) {
                    int new_caps = b.captured_black + b.captured_white;
                    hw_push_board(&b);
                    hw_cursor(cursor_row, cursor_col, 1);
                    render_panel(&b);
                    hw_play_audio(new_caps > prev_caps ? AUDIO_CAPTURE : AUDIO_PLACE);
                    printf("%s plays (%d,%d). Captured: B=%d W=%d.\n",
                           stone_str(moved), cursor_row, cursor_col,
                           b.captured_black, b.captured_white);
                    if (pvc && !b.game_over && b.turn == WHITE) {
                        printf("AI (level %d) thinking...\n", ai_level);
                        fflush(stdout);
                        apply_ai_move(&b, ai_level, cursor_row, cursor_col);
                    }
                } else {
                    hw_play_audio(AUDIO_ILLEGAL);
                    printf("Illegal: %s\n", moveresult_str(mr));
                }
                break;
```

```
            }
            case KEY_SPACE: {
                if (b.game_over) { printf("Game over — press R to return to menu.\n");
break; }
                Stone passer = b.turn;
                board_pass(&b);
                render_panel(&b);
                printf("%s passes (%d/2). %s to move.\n",
                       stone_str(passer), b.consecutive_passes,
                       stone_str(b.turn));
                if (b.game_over) {
                    int blk, wht;
                    board_score(&b, &blk, &wht);
                    hw_play_audio(AUDIO_GAME_OVER);
                    printf("\nGAME OVER\n  Black: %d points\n  White: %d + 5.5 komi =
%.1f\n",
                           blk, wht, wht + 5.5);
                    printf("  Winner: %s\n",
                           (blk > wht + 5.5) ? "Black" : "White");
                }
                break;
            }
            case KEY_R:
                /* From GAME / GAME_OVER: back to TITLE menu (or skip back to
                 * GAME if launched with the argv shortcut). */
                if (skip_menu) {
                    enter_game(&b, &cursor_row, &cursor_col);
                    printf("\n--- new game ---\n");
                } else {
                    ui = UI_TITLE;
                    enter_title();
                    printf("\n--- back to menu ---\n");
                }
                break;
            default: break;
            }
            break;
        }
    }

done:
    hw_cursor(cursor_row, cursor_col, 0);
    libusb_close(kbd);
    printf("Bye.\n");
    return 0;
}
```

ai.c

C/C++

```c
/*
 * ai.c — Random / Greedy / MCTS Go AI
 *
 * The MCTS implementation here is intentionally simple:
 *   - Fixed-size node pool (no malloc per node).
 *   - Untried-moves list per node, populated lazily.
 *   - Rollouts use a fast play path: no ko check, just suicide rejection.
 *   - Win-count is from the perspective of the parent node's turn.
 *
 * If MCTS strength is insufficient at 9×9, the next levers (in order):
 *   1. Increase MCTS_SIMULATIONS (currently 200).
 *   2. Add RAVE / progressive bias.
 *   3. Move rollouts to FPGA accelerator (Phase 7b).
 */

#include "ai.h"

#include <math.h>
#include <string.h>
#include <stdlib.h>

#define N BOARD_N

/* ─── PRNG: xorshift32 ────────────────────────────────────────────────────── */

static uint32_t rng_state = 0xDEADBEEFu;

void ai_seed(uint32_t seed)
{
    rng_state = seed ? seed : 0xDEADBEEFu;
}

static inline uint32_t rand32(void)
{
    uint32_t x = rng_state;
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    rng_state = x;
    return x;
}

static inline int rand_below(int n)
{
    return (int)(rand32() % (uint32_t)n);
}

/* ─── Local utility: liberty count + capture (fast, no ko check) ──────────── */

/* Returns liberties of group at (sr, sc). */
static int libs_at(const Stone cells[N][N], int sr, int sc)
{
    Stone c = cells[sr][sc];
    if (c == EMPTY) return -1;
```

```
    char visited[N][N];
    memset(visited, 0, sizeof(visited));
    int q[N * N][2]; int qh = 0, qt = 0;
    q[qt][0] = sr; q[qt][1] = sc; qt++;
    visited[sr][sc] = 1;
    int libs = 0;
    while (qh < qt) {
        int r = q[qh][0], cc = q[qh][1]; qh++;
        const int dr[4]={-1,1,0,0}, dc[4]={0,0,-1,1};
        for (int i = 0; i < 4; i++) {
            int nr = r + dr[i], nc = cc + dc[i];
            if (nr < 0 || nr >= N || nc < 0 || nc >= N) continue;
            if (visited[nr][nc]) continue;
            if (cells[nr][nc] == EMPTY) { libs++; visited[nr][nc] = 1; }
            else if (cells[nr][nc] == c) { visited[nr][nc] = 1;
                                           q[qt][0]=nr; q[qt][1]=nc; qt++; }
        }
    }
    return libs;
}

/* Apply move at (r, c) for color, removing dead opponent groups.
 * Caller guarantees the cell is EMPTY and the move is non-suicide.
 * Returns # captured stones. */
static int fast_apply_move(BoardState *b, int row, int col, Stone color)
{
    b->cells[row][col] = color;
    Stone opp = (color == BLACK) ? WHITE : BLACK;
    int total = 0;
    const int dr[4]={-1,1,0,0}, dc[4]={0,0,-1,1};
    for (int i = 0; i < 4; i++) {
        int nr = row + dr[i], nc = col + dc[i];
        if (nr < 0 || nr >= N || nc < 0 || nc >= N) continue;
        if (b->cells[nr][nc] != opp) continue;
        if (libs_at((const Stone (*)[N])b->cells, nr, nc) == 0) {
            /* flood-fill remove */
            char visited[N][N];
            memset(visited, 0, sizeof(visited));
            int q[N*N][2]; int qh=0, qt=0;
            q[qt][0]=nr; q[qt][1]=nc; qt++;
            visited[nr][nc] = 1;
            while (qh < qt) {
                int r = q[qh][0], cc = q[qh][1]; qh++;
                if (b->cells[r][cc] == opp) {
                    b->cells[r][cc] = EMPTY;
                    total++;
                    const int ddr[4]={-1,1,0,0}, ddc[4]={0,0,-1,1};
                    for (int j = 0; j < 4; j++) {
                        int rr = r + ddr[j], cc2 = cc + ddc[j];
                        if (rr<0||rr>=N||cc2<0||cc2>=N) continue;
                        if (visited[rr][cc2]) continue;
                        if (b->cells[rr][cc2] == opp) {
                            visited[rr][cc2] = 1;
                            q[qt][0]=rr; q[qt][1]=cc2; qt++;
```

```
                    }
                }
            }
        }
    }
    if (opp == BLACK) b->captured_black += total;
    else              b->captured_white += total;
    return total;
}

/* Is (r, c) a legal move for `color`? Skips ko. */
static int can_play_fast(const BoardState *b, int row, int col, Stone color)
{
    if (row < 0 || row >= N || col < 0 || col >= N) return 0;
    if (b->cells[row][col] != EMPTY) return 0;
    /* Speculative copy */
    BoardState scratch = *b;
    scratch.cells[row][col] = color;
    Stone opp = (color == BLACK) ? WHITE : BLACK;
    int captures = 0;
    const int dr[4]={-1,1,0,0}, dc[4]={0,0,-1,1};
    for (int i = 0; i < 4; i++) {
        int nr = row + dr[i], nc = col + dc[i];
        if (nr<0||nr>=N||nc<0||nc>=N) continue;
        if (scratch.cells[nr][nc] != opp) continue;
        if (libs_at((const Stone (*)[N])scratch.cells, nr, nc) == 0) {
            captures++;
            break;
        }
    }
    if (captures == 0) {
        if (libs_at((const Stone (*)[N])scratch.cells, row, col) == 0)
            return 0;   /* suicide */
    }
    return 1;
}

/* Enumerate legal moves for b->turn into out[], return count. */
static int legal_moves(const BoardState *b, Move *out)
{
    int n = 0;
    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++)
            if (can_play_fast(b, r, c, b->turn)) {
                out[n].row = r; out[n].col = c; out[n].pass = 0;
                n++;
            }
    return n;
}

/* ── Level 1: random ─────────────────────────────────────────── */

static Move move_random(const BoardState *b)
```

```c
{
    Move legals[N * N];
    int n = legal_moves(b, legals);
    if (n == 0) {
        Move pass = { 0, 0, 1 };
        return pass;
    }
    return legals[rand_below(n)];
}

/* ─── Level 2: greedy heuristic ─────────────────────────────────────────── */

static Move move_greedy(const BoardState *b)
{
    Move best = { 0, 0, 1 };
    int  best_score = -1;

    Stone color = b->turn;
    Stone opp   = (color == BLACK) ? WHITE : BLACK;

    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++) {
            if (!can_play_fast(b, r, c, color)) continue;

            BoardState scratch = *b;
            int caps = fast_apply_move(&scratch, r, c, color);

            int score = 0;
            score += caps * 10;

            /* Defend an atari group: was an own group at 1 lib that now has >1? */
            const int dr[4]={-1,1,0,0}, dc[4]={0,0,-1,1};
            for (int i = 0; i < 4; i++) {
                int nr = r + dr[i], nc = c + dc[i];
                if (nr<0||nr>=N||nc<0||nc>=N) continue;
                if (b->cells[nr][nc] == color) {
                    int before = libs_at((const Stone(*)[N])b->cells, nr, nc);
                    int after  = libs_at((const Stone(*)[N])scratch.cells, nr, nc);
                    if (before == 1 && after > 1) score += 8;
                }
                /* Reduce opponent libs */
                if (b->cells[nr][nc] == opp && scratch.cells[nr][nc] == opp) {
                    int before = libs_at((const Stone(*)[N])b->cells, nr, nc);
                    int after  = libs_at((const Stone(*)[N])scratch.cells, nr, nc);
                    score += (before - after) * 2;
                }
            }

            /* Center bias: max(0, 4 - chebyshev_dist_to_(4,4)) */
            int cd = abs(r - 4) > abs(c - 4) ? abs(r - 4) : abs(c - 4);
            score += (cd <= 4) ? (4 - cd) : 0;

            /* Random tiebreak for diversity */
            score = score * 16 + (rand32() & 0xF);
```

```c
            if (score > best_score) {
                best_score = score;
                best.row = r; best.col = c; best.pass = 0;
            }
        }
    }
    return best;
}

/* ─── Level 3: serial MCTS (UCT) ─────────────────────────────────────────── */

#define MCTS_POOL       6000
#define MCTS_SIMS       200
#define MCTS_MAX_DEPTH  324    /* 4 × N² */
#define MCTS_CHILDREN_MAX (N * N + 1)   /* 81 cells + pass */
#define UCT_C           1.41421356237

typedef struct MctsNode {
    Move   move;       /* move that led to this node */
    Stone  color;      /* color that just played to reach this node */
    int    visits;
    int    wins;       /* wins for `color` */
    struct MctsNode *parent;
    int    n_children;
    int    n_untried;
    Move   untried[MCTS_CHILDREN_MAX];
    struct MctsNode *children[MCTS_CHILDREN_MAX];
} MctsNode;

static MctsNode mcts_pool[MCTS_POOL];
static int      mcts_pool_n;

static MctsNode *node_new(Move m, Stone color, MctsNode *parent,
                          const BoardState *state)
{
    if (mcts_pool_n >= MCTS_POOL) return NULL;
    MctsNode *n = &mcts_pool[mcts_pool_n++];
    n->move = m;
    n->color = color;
    n->visits = 0;
    n->wins = 0;
    n->parent = parent;
    n->n_children = 0;
    n->n_untried = legal_moves(state, n->untried);
    /* Always allow pass as an option in the tree (otherwise the AI never
     * passes even when surrounded by territory). */
    if (n->n_untried < MCTS_CHILDREN_MAX) {
        Move pass = { 0, 0, 1 };
        n->untried[n->n_untried++] = pass;
    }
    return n;
}
```

```c
static void apply_move(BoardState *b, Move m)
{
    if (m.pass) {
        b->prev_board_hash = 0; /* irrelevant for fast rollout */
        b->turn = (b->turn == BLACK) ? WHITE : BLACK;
        b->consecutive_passes++;
        if (b->consecutive_passes >= 2) b->game_over = 1;
    } else {
        fast_apply_move(b, m.row, m.col, b->turn);
        b->turn = (b->turn == BLACK) ? WHITE : BLACK;
        b->consecutive_passes = 0;
    }
}

/* Pick child with highest UCB1 score. */
static MctsNode *best_uct_child(MctsNode *node)
{
    MctsNode *best = NULL;
    double    best_v = -1e9;
    double    log_parent = log((double)node->visits + 1.0);
    for (int i = 0; i < node->n_children; i++) {
        MctsNode *ch = node->children[i];
        double exploit = (double)ch->wins / (double)(ch->visits ? ch->visits : 1);
        double explore = UCT_C * sqrt(log_parent / (double)(ch->visits ? ch->visits :
1));
        double v = exploit + explore;
        if (v > best_v) { best_v = v; best = ch; }
    }
    return best;
}

/* Random rollout from `state` to terminal (2 passes or move limit).
 * Returns the winning color (BLACK or WHITE). */
static Stone rollout(BoardState state)
{
    int moves = 0;
    while (!state.game_over && moves < MCTS_MAX_DEPTH) {
        /* Pick a random legal move; if none, pass. */
        Move legals[N * N];
        int  n = legal_moves(&state, legals);
        Move m;
        if (n == 0) { m.row = 0; m.col = 0; m.pass = 1; }
        else        m = legals[rand_below(n)];
        apply_move(&state, m);
        moves++;
    }
    int blk, wht;
    board_score(&state, &blk, &wht);
    return (blk > wht + 5.5) ? BLACK : WHITE;
}

static Move move_mcts(const BoardState *b)
{
    mcts_pool_n = 0;
```

```c
Stone root_color = (b->turn == BLACK) ? WHITE : BLACK;
Move root_move = { 0, 0, 0 };
MctsNode *root = node_new(root_move, root_color, NULL, b);
if (!root) {
    /* Pool exhausted before root; fall back to greedy. */
    return move_greedy(b);
}

for (int sim = 0; sim < MCTS_SIMS; sim++) {
    MctsNode *node = root;
    BoardState state = *b;

    /* Selection: descend while node is fully expanded and has children. */
    while (node->n_untried == 0 && node->n_children > 0) {
        MctsNode *ch = best_uct_child(node);
        if (!ch) break;
        apply_move(&state, ch->move);
        node = ch;
    }

    /* Expansion */
    if (node->n_untried > 0) {
        int idx = rand_below(node->n_untried);
        Move m = node->untried[idx];
        node->untried[idx] = node->untried[--node->n_untried];
        apply_move(&state, m);
        MctsNode *child = node_new(m, state.turn == BLACK ? WHITE : BLACK,
                                   node, &state);
        if (child) {
            node->children[node->n_children++] = child;
            node = child;
        }
    }

    /* Simulation */
    Stone winner = rollout(state);

    /* Backpropagation */
    for (MctsNode *p = node; p != NULL; p = p->parent) {
        p->visits++;
        if (p->color == winner) p->wins++;
    }
}

/* Pick child with highest visit count. */
MctsNode *best = NULL;
int best_visits = -1;
for (int i = 0; i < root->n_children; i++) {
    if (root->children[i]->visits > best_visits) {
        best_visits = root->children[i]->visits;
        best = root->children[i];
    }
}
if (!best) {
```

```
        Move pass = { 0, 0, 1 };
        return pass;
    }
    return best->move;
}

/* ─── Public dispatch ──────────────────────────────────────────────────── */

Move ai_get_move(const BoardState *b, AiLevel level)
{
    switch (level) {
    case AI_RANDOM: return move_random(b);
    case AI_GREEDY: return move_greedy(b);
    case AI_MCTS:   return move_mcts(b);
    }
    Move pass = { 0, 0, 1 };
    return pass;
}
```

ai.h

```c
/*
 * ai.h — Computer-player API for the Go engine
 *
 * Three levels:
 *   AI_RANDOM — uniform-random over legal moves; passes if none.
 *   AI_GREEDY — heuristic (capture / defend atari / reduce libs / center bias).
 *   AI_MCTS   — UCT-MCTS, 200 simulations, max depth 4×N² = 324 moves per
 *               rollout. Uses simplified rules during rollouts (no ko check,
 *               suicide rejected) for speed.
 *
 * All three are pure C and run on the HPS Cortex-A9. The MCTS rollout phase
 * is the candidate for FPGA offload (design-document.md §7.5, Phase 7b);
 * not implemented yet.
 */

#ifndef _AI_H
#define _AI_H

#include "board.h"

typedef enum {
    AI_RANDOM = 1,
    AI_GREEDY = 2,
    AI_MCTS   = 3,
} AiLevel;

typedef struct {
    int row;        /* 0..8 */
    int col;        /* 0..8 */
    int pass;       /* 1 = pass, 0 = real move */
```

```
} Move;

/* Returns the AI's move for b->turn at the requested level.
 * If no legal moves exist (shouldn't happen except in extreme positions),
 * returns {.pass = 1}. */
Move ai_get_move(const BoardState *b, AiLevel level);

/* Seed the AI's PRNG. Call once at startup if you want non-deterministic
 * games; otherwise the AI plays the same way every run (useful for tests). */
void ai_seed(uint32_t seed);

#endif
```

board.c

```
C/C++
/*
 * board.c — Go rule engine implementation
 *
 * Pure C, no hardware dependencies. Algorithms straight from
 * design-document.md §7. Liberty counting is a queue-based BFS over
 * 4-connected neighbors. Captures are detected by counting liberties of each
 * opponent group adjacent to a placed stone after speculative placement.
 * Ko detection uses a 64-bit Zobrist hash compared to the previous board
 * position; this catches simple (one-move) ko but NOT positional super-ko.
 */

#include "board.h"

#include <string.h>
#include <stdlib.h>
#include <time.h>

#define N BOARD_N

/* ─── Zobrist hash table — initialized once on first board_init() ─────────── */

static uint64_t zob_table[N][N][3];   /* index: row, col, stone(0..2) */
static int      zob_initialized = 0;

static uint64_t rand64(void)
{
    /* Combine two rand() calls; rand() returns up to RAND_MAX which is at
     * least 2^15 - 1. Quality is fine for hash-table seeds. */
    uint64_t hi = (uint64_t)rand();
    uint64_t lo = (uint64_t)rand();
    return (hi << 32) ^ ((uint64_t)rand() << 16) ^ lo;
}

static void zob_init_once(void)
{
    if (zob_initialized) return;
```

```c
    /* Deterministic seed so that test runs are reproducible. The actual hash
     * values are irrelevant as long as they're well-distributed. */
    srand(0xC0FFEE);
    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++)
            for (int s = 0; s < 3; s++)
                zob_table[r][c][s] = rand64();
    zob_initialized = 1;
}

static uint64_t zob_hash(const Stone cells[N][N])
{
    uint64_t h = 0;
    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++)
            h ^= zob_table[r][c][cells[r][c]];
    return h;
}

/* ─── BFS liberty count + group collection ──────────────────────────────── */

/* Walks the connected group containing (sr, sc); fills group[] with up to
 * cap members; counts unique adjacent EMPTY cells. Returns liberty count.
 * If group is NULL, only the count is computed. */
static int liberties_and_group(const Stone cells[N][N], int sr, int sc,
                               int (*group)[2], int cap, int *group_size)
{
    Stone color = cells[sr][sc];
    if (color == EMPTY) { if (group_size) *group_size = 0; return -1; }

    /* visited tracks both group cells and counted liberty cells, so we don't
     * double-count a liberty bordering multiple stones in the group. */
    char visited[N][N];
    memset(visited, 0, sizeof(visited));

    int q[N * N][2];
    int qh = 0, qt = 0;
    q[qt][0] = sr; q[qt][1] = sc; qt++;
    visited[sr][sc] = 1;

    int liberties = 0;
    int g_n = 0;

    while (qh < qt) {
        int r = q[qh][0], c = q[qh][1]; qh++;
        if (group && g_n < cap) {
            group[g_n][0] = r;
            group[g_n][1] = c;
        }
        g_n++;

        const int dr[4] = {-1, 1,  0, 0};
        const int dc[4] = { 0, 0, -1, 1};
        for (int i = 0; i < 4; i++) {
```

```c
            int nr = r + dr[i], nc = c + dc[i];
            if (nr < 0 || nr >= N || nc < 0 || nc >= N) continue;
            if (visited[nr][nc]) continue;
            if (cells[nr][nc] == EMPTY) {
                liberties++;
                visited[nr][nc] = 1;
            } else if (cells[nr][nc] == color) {
                visited[nr][nc] = 1;
                q[qt][0] = nr; q[qt][1] = nc; qt++;
            }
        }
    }

    if (group_size) *group_size = g_n;
    return liberties;
}

/* Capture every opponent group adjacent to (r, c) that has 0 liberties.
 * Returns the count of stones removed. Updates b->captured_*. */
static int process_captures(BoardState *b, int r, int c)
{
    Stone placed = b->cells[r][c];
    Stone opp    = (placed == BLACK) ? WHITE : BLACK;
    int total = 0;

    const int dr[4] = {-1, 1,  0, 0};
    const int dc[4] = { 0, 0, -1, 1};
    for (int i = 0; i < 4; i++) {
        int nr = r + dr[i], nc = c + dc[i];
        if (nr < 0 || nr >= N || nc < 0 || nc >= N) continue;
        if (b->cells[nr][nc] != opp) continue;

        int group[N * N][2];
        int g_n = 0;
        int libs = liberties_and_group(
            (const Stone (*)[N])b->cells, nr, nc, group, N * N, &g_n);
        if (libs == 0) {
            for (int j = 0; j < g_n; j++)
                b->cells[group[j][0]][group[j][1]] = EMPTY;
            if (opp == BLACK) b->captured_black += g_n;
            else              b->captured_white += g_n;
            total += g_n;
        }
    }
    return total;
}

/* ─── Public API ──────────────────────────────────────────────────────────── */

void board_init(BoardState *b)
{
    zob_init_once();
    memset(b, 0, sizeof(*b));
    b->turn = BLACK;
```

```
    b->prev_board_hash = zob_hash((const Stone (*)[N])b->cells);
}

MoveResult board_place(BoardState *b, int row, int col)
{
    if (row < 0 || row >= N || col < 0 || col >= N) return MOVE_ILLEGAL_OCCUPIED;
    if (b->cells[row][col] != EMPTY) return MOVE_ILLEGAL_OCCUPIED;

    /* Speculatively place on a scratch board */
    Stone scratch[N][N];
    memcpy(scratch, b->cells, sizeof(scratch));
    scratch[row][col] = b->turn;

    /* Capture opponent groups (mutates scratch) */
    BoardState tmp = *b;
    memcpy(tmp.cells, scratch, sizeof(scratch));
    int captures = process_captures(&tmp, row, col);

    /* Suicide: own group has no liberties AND no captures occurred */
    int g_n = 0;
    int own_libs = liberties_and_group(
        (const Stone (*)[N])tmp.cells, row, col, NULL, 0, &g_n);
    if (own_libs == 0 && captures == 0)
        return MOVE_ILLEGAL_SUICIDE;

    /* Ko: resulting position == previous position */
    uint64_t new_hash = zob_hash((const Stone (*)[N])tmp.cells);
    if (new_hash == b->prev_board_hash)
        return MOVE_ILLEGAL_KO;

    /* Commit. Save current hash for next move's ko check. */
    b->prev_board_hash = zob_hash((const Stone (*)[N])b->cells);
    memcpy(b->cells, tmp.cells, sizeof(tmp.cells));
    b->captured_black = tmp.captured_black;
    b->captured_white = tmp.captured_white;
    b->turn = (b->turn == BLACK) ? WHITE : BLACK;
    b->consecutive_passes = 0;
    return MOVE_OK;
}

void board_pass(BoardState *b)
{
    b->prev_board_hash = zob_hash((const Stone (*)[N])b->cells);
    b->turn = (b->turn == BLACK) ? WHITE : BLACK;
    b->consecutive_passes++;
    if (b->consecutive_passes >= 2)
        b->game_over = 1;
}

/* Flood-fill an empty region; record colors that border it. */
static void flood_empty_region(const Stone cells[N][N], int sr, int sc,
                               char visited[N][N],
                               int *region_size, int *seen_black, int *seen_white)
{
```

```c
    int q[N * N][2];
    int qh = 0, qt = 0;
    q[qt][0] = sr; q[qt][1] = sc; qt++;
    visited[sr][sc] = 1;
    *region_size = 0;
    *seen_black = 0;
    *seen_white = 0;

    while (qh < qt) {
        int r = q[qh][0], c = q[qh][1]; qh++;
        (*region_size)++;

        const int dr[4] = {-1, 1,  0, 0};
        const int dc[4] = { 0, 0, -1, 1};
        for (int i = 0; i < 4; i++) {
            int nr = r + dr[i], nc = c + dc[i];
            if (nr < 0 || nr >= N || nc < 0 || nc >= N) continue;
            if (cells[nr][nc] == EMPTY) {
                if (!visited[nr][nc]) {
                    visited[nr][nc] = 1;
                    q[qt][0] = nr; q[qt][1] = nc; qt++;
                }
            } else if (cells[nr][nc] == BLACK) {
                *seen_black = 1;
            } else if (cells[nr][nc] == WHITE) {
                *seen_white = 1;
            }
        }
    }
}

void board_score(const BoardState *b, int *black_pts, int *white_pts)
{
    int blk = 0, wht = 0;

    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++) {
            if (b->cells[r][c] == BLACK) blk++;
            else if (b->cells[r][c] == WHITE) wht++;
        }

    char visited[N][N];
    memset(visited, 0, sizeof(visited));
    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++) {
            if (b->cells[r][c] != EMPTY || visited[r][c]) continue;
            int sz, sb, sw;
            flood_empty_region((const Stone (*)[N])b->cells, r, c, visited,
                               &sz, &sb, &sw);
            if (sb && !sw)      blk += sz;
            else if (sw && !sb) wht += sz;
            /* contested or fully surrounded by neither (shouldn't happen on
             * a complete board): neutral, no points. */
        }
```

```
    *black_pts = blk;
    *white_pts = wht;
}
```

board.h

```
C/C++
/*
 * board.h — Go rule engine API
 *
 * Pure C, no hardware dependencies — board.c can be unit-tested on any host.
 * The 9x9 board state, move validation (occupancy / suicide / ko), capture
 * detection, scoring, and Zobrist hashing all live here.
 *
 * Reference: design-document.md §7.
 */

#ifndef _BOARD_H
#define _BOARD_H

#include <stdint.h>

#define BOARD_N 9

typedef enum {
    EMPTY = 0,
    BLACK = 1,
    WHITE = 2,
} Stone;

typedef enum {
    MOVE_OK = 0,
    MOVE_ILLEGAL_OCCUPIED,
    MOVE_ILLEGAL_SUICIDE,
    MOVE_ILLEGAL_KO,
} MoveResult;

typedef struct {
    Stone    cells[BOARD_N][BOARD_N];   /* [row][col]                         */
    Stone    turn;                      /* whose turn it is (BLACK or WHITE)   */
    int      captured_black;            /* count of black stones captured by W */
    int      captured_white;            /* count of white stones captured by B */
    uint64_t prev_board_hash;           /* Zobrist hash of position before our  *
                                         * last move — used to detect simple   *
                                         * ko (immediate capture-back).        */
    int      consecutive_passes;        /* 2 = game over                       */
    int      game_over;                 /* set when consecutive_passes >= 2    */
} BoardState;

/* Initialize an empty board with Black to move and a fresh Zobrist table.
 * Calling this multiple times resets state but seeds the Zobrist table only
```

```
 * once (so prior hashes remain valid across multiple board_init calls — useful
 * for tests). */
void board_init(BoardState *b);

/* Place a stone for b->turn at (row, col). On MOVE_OK:
 *   - the stone is placed,
 *   - any opponent groups with zero liberties are removed,
 *   - captured_black/white are updated,
 *   - prev_board_hash is updated to hold the pre-move hash,
 *   - turn is toggled to the opponent,
 *   - consecutive_passes reset to 0.
 * On any failure: state is unchanged. */
MoveResult board_place(BoardState *b, int row, int col);

/* Pass the turn. Always legal. Increments consecutive_passes; game_over
 * becomes true when it reaches 2. */
void board_pass(BoardState *b);

/* Chinese-style area score: stones on board + territory enclosed solely by
 * one color. Komi (5.5 for white) is NOT applied here; the caller adds it.
 * Writes integer point counts to *black_pts and *white_pts. */
void board_score(const BoardState *b, int *black_pts, int *white_pts);

#endif /* _BOARD_H */
```

font5x7.c

C/C++

```
/*
 * font5x7.c — minimal 5x7 ASCII bitmap font
 *
 * Glyph bytes derived from the public-domain/BSD Adafruit-GFX 5x7 stockfont.
 */

#include "font5x7.h"

/* The placeholder glyph for unsupported chars: a solid rectangle. */
static const uint8_t glyph_placeholder[5] = { 0x7F, 0x41, 0x41, 0x41, 0x7F };

/* Common punctuation */
static const uint8_t glyph_space [5] = { 0x00, 0x00, 0x00, 0x00, 0x00 };
static const uint8_t glyph_period[5] = { 0x00, 0x60, 0x60, 0x00, 0x00 };
static const uint8_t glyph_colon [5] = { 0x00, 0x36, 0x36, 0x00, 0x00 };
static const uint8_t glyph_equal [5] = { 0x14, 0x14, 0x14, 0x14, 0x14 };
static const uint8_t glyph_dash  [5] = { 0x08, 0x08, 0x08, 0x08, 0x08 };
static const uint8_t glyph_slash [5] = { 0x20, 0x10, 0x08, 0x04, 0x02 };
static const uint8_t glyph_excl  [5] = { 0x00, 0x00, 0x5F, 0x00, 0x00 };

/* Digits */
static const uint8_t glyph_digit[10][5] = {
    { 0x3E, 0x51, 0x49, 0x45, 0x3E }, // 0
    { 0x00, 0x42, 0x7F, 0x40, 0x00 }, // 1
```

```
    { 0x42, 0x61, 0x51, 0x49, 0x46 }, // 2
    { 0x21, 0x41, 0x45, 0x4B, 0x31 }, // 3
    { 0x18, 0x14, 0x12, 0x7F, 0x10 }, // 4
    { 0x27, 0x45, 0x45, 0x45, 0x39 }, // 5
    { 0x3C, 0x4A, 0x49, 0x49, 0x30 }, // 6
    { 0x01, 0x71, 0x09, 0x05, 0x03 }, // 7
    { 0x36, 0x49, 0x49, 0x49, 0x36 }, // 8
    { 0x06, 0x49, 0x49, 0x29, 0x1E }, // 9
};

/* Uppercase letters A..Z */
static const uint8_t glyph_upper[26][5] = {
    { 0x7C, 0x12, 0x11, 0x12, 0x7C }, // A
    { 0x7F, 0x49, 0x49, 0x49, 0x36 }, // B
    { 0x3E, 0x41, 0x41, 0x41, 0x22 }, // C
    { 0x7F, 0x41, 0x41, 0x22, 0x1C }, // D
    { 0x7F, 0x49, 0x49, 0x49, 0x41 }, // E
    { 0x7F, 0x09, 0x09, 0x09, 0x01 }, // F
    { 0x3E, 0x41, 0x49, 0x49, 0x7A }, // G
    { 0x7F, 0x08, 0x08, 0x08, 0x7F }, // H
    { 0x00, 0x41, 0x7F, 0x41, 0x00 }, // I
    { 0x20, 0x40, 0x41, 0x3F, 0x01 }, // J
    { 0x7F, 0x08, 0x14, 0x22, 0x41 }, // K
    { 0x7F, 0x40, 0x40, 0x40, 0x40 }, // L
    { 0x7F, 0x02, 0x0C, 0x02, 0x7F }, // M
    { 0x7F, 0x04, 0x08, 0x10, 0x7F }, // N
    { 0x3E, 0x41, 0x41, 0x41, 0x3E }, // O
    { 0x7F, 0x09, 0x09, 0x09, 0x06 }, // P
    { 0x3E, 0x41, 0x51, 0x21, 0x5E }, // Q
    { 0x7F, 0x09, 0x19, 0x29, 0x46 }, // R
    { 0x46, 0x49, 0x49, 0x49, 0x31 }, // S
    { 0x01, 0x01, 0x7F, 0x01, 0x01 }, // T
    { 0x3F, 0x40, 0x40, 0x40, 0x3F }, // U
    { 0x1F, 0x20, 0x40, 0x20, 0x1F }, // V
    { 0x3F, 0x40, 0x38, 0x40, 0x3F }, // W
    { 0x63, 0x14, 0x08, 0x14, 0x63 }, // X
    { 0x07, 0x08, 0x70, 0x08, 0x07 }, // Y
    { 0x61, 0x51, 0x49, 0x45, 0x43 }, // Z
};

const uint8_t *font5x7_glyph(char c)
{
    if (c >= '0' && c <= '9') return glyph_digit[c - '0'];
    if (c >= 'A' && c <= 'Z') return glyph_upper[c - 'A'];
    if (c >= 'a' && c <= 'z') return glyph_upper[c - 'a'];   /* upcase */
    switch (c) {
    case ' ': return glyph_space;
    case '.': return glyph_period;
    case ':': return glyph_colon;
    case '=': return glyph_equal;
    case '-': return glyph_dash;
    case '/': return glyph_slash;
    case '!': return glyph_excl;
    default:  return glyph_placeholder;
```

```
    }
}
```

font5x7.h

```
C/C++
/*
 * font5x7.h — minimal 5x7 ASCII bitmap font
 *
 * Each glyph is 5 columns × 7 rows. Storage is 5 bytes per glyph, one byte
 * per column. Bit 0 of each byte is the TOP row, bit 6 is the BOTTOM row;
 * bit 7 is unused.
 *
 * Coverage: digits 0-9, uppercase A-Z, and punctuation enough for score and
 * menu text (' ', '.', ':', '=', '-', '/'). Lowercase, symbols, and other
 * characters render as a solid block (the "?" placeholder).
 *
 * Glyph data is taken from the standard 5x7 font (e.g., Adafruit-GFX
 * stockfont) which is widely re-used and BSD-licensed.
 */

#ifndef _FONT5X7_H
#define _FONT5X7_H

#include <stdint.h>

#define FONT_W 5
#define FONT_H 7

/* Returns a pointer to 5 bytes describing the columns of glyph c.
 * Always returns a valid pointer; unsupported chars get a placeholder. */
const uint8_t *font5x7_glyph(char c);

#endif
```

strip_render.c

```
C/C++
/*
 * strip_render.c — score-strip framebuffer software renderer
 *
 * Buffers a full 38,400-byte 8bpp pixel array locally; flushes to FPGA back
 * buffer in one tight loop on strip_present(). The flush is 9,600 32-bit
 * word writes; at ~4 LW cycles/transaction × 50 MHz that's ~770 µs, well
 * under one frame budget.
 */

#include "strip_render.h"
#include "font5x7.h"

#include <string.h>
```

```c
static uint8_t scratch[STRIP_BYTES];

/* Cached pointers set once by strip_init() */
static volatile uint32_t *strip_fb_words;     /* avalon_slave_1 base, 32-bit */
static volatile uint8_t  *reg_strip_swap;     /* avalon_slave_0 + 0x05 */

void strip_init(volatile uint8_t *lw_bridge_base, uint32_t strip_offset,
                uint32_t reg_strip_swap_offset)
{
    strip_fb_words = (volatile uint32_t *)(lw_bridge_base + strip_offset);
    reg_strip_swap = lw_bridge_base + reg_strip_swap_offset;
    strip_clear(COLOR_STRIP_BG);
}

void strip_clear(uint8_t color)
{
    memset(scratch, color, STRIP_BYTES);
}

void strip_pixel(int x, int y, uint8_t color)
{
    if (x < 0 || x >= STRIP_W || y < 0 || y >= STRIP_H) return;
    scratch[y * STRIP_W + x] = color;
}

void strip_rect(int x, int y, int w, int h, uint8_t color)
{
    int x1 = x + w, y1 = y + h;
    if (x  < 0)        x  = 0;
    if (y  < 0)        y  = 0;
    if (x1 > STRIP_W)  x1 = STRIP_W;
    if (y1 > STRIP_H)  y1 = STRIP_H;
    for (int yy = y; yy < y1; yy++) {
        memset(&scratch[yy * STRIP_W + x], color, x1 - x);
    }
}

void strip_char(int x, int y, char c, int scale, uint8_t fg, uint8_t bg)
{
    const uint8_t *g = font5x7_glyph(c);
    if (scale < 1) scale = 1;
    for (int gx = 0; gx < FONT_W; gx++) {
        uint8_t col = g[gx];
        for (int gy = 0; gy < FONT_H; gy++) {
            uint8_t color = (col & (1u << gy)) ? fg : bg;
            /* Plot a scale×scale block per font pixel */
            int px = x + gx * scale;
            int py = y + gy * scale;
            for (int dy = 0; dy < scale; dy++)
                for (int dx = 0; dx < scale; dx++)
                    strip_pixel(px + dx, py + dy, color);
        }
    }
}
```

```c
void strip_text(int x, int y, const char *s, int scale,
                uint8_t fg, uint8_t bg)
{
    int cx = x;
    /* 1-pixel inter-character gap × scale */
    int advance = (FONT_W + 1) * scale;
    while (*s) {
        strip_char(cx, y, *s, scale, fg, bg);
        cx += advance;
        s++;
    }
}

int strip_text_width(const char *s, int scale)
{
    if (scale < 1) scale = 1;
    int n = 0;
    for (const char *p = s; *p; p++) n++;
    if (n == 0) return 0;
    return n * FONT_W * scale + (n - 1) * scale;
}

void strip_text_centered(int y, const char *s, int scale,
                         uint8_t fg, uint8_t bg)
{
    int w = strip_text_width(s, scale);
    int x = (STRIP_W - w) / 2;
    if (x < 0) x = 0;
    strip_text(x, y, s, scale, fg, bg);
}

void strip_text_box(int x, int y, const char *s, int scale,
                    uint8_t fg, uint8_t bg, int pad)
{
    if (scale < 1) scale = 1;
    if (pad   < 0) pad   = 0;
    int tw = strip_text_width(s, scale);
    int th = FONT_H * scale;
    strip_rect(x, y, tw + 2 * pad, th + 2 * pad, bg);
    strip_text(x + pad, y + pad, s, scale, fg, bg);
}

void strip_present(void)
{
    /* Push 9,600 32-bit words to the back buffer. Each word holds 4 packed
     * 8-bit pixels in little-endian: word w → pixels 4w, 4w+1, 4w+2, 4w+3 */
    const uint32_t *src = (const uint32_t *)scratch;
    for (int w = 0; w < STRIP_WORDS; w++) {
        strip_fb_words[w] = src[w];
    }
    /* Arm the swap. Hardware will toggle the active buffer on the next VS. */
    *reg_strip_swap = 1;
}
```

strip_render.h

```c
/*
 * strip_render.h — software rendering into the score-strip framebuffer
 *
 * The strip is 640×60 pixels, 8 bits-per-pixel palette indices. Software
 * draws into a local 38,400-byte buffer with the primitives below, then
 * calls strip_present() to bulk-write the buffer to the FPGA back buffer
 * and arm a vsync-aligned swap.
 *
 * Palette (matches the 8-color case statement in go_peripheral.sv):
 *   0 = bg dark gray  1 = white text   2 = light gray   3 = green
 *   4 = gold (winner) 5 = red (error)  6 = blue         7 = burlywood
 */

#ifndef _STRIP_RENDER_H
#define _STRIP_RENDER_H

#include <stdint.h>

#define STRIP_W       640
#define STRIP_H       60
#define STRIP_BYTES   (STRIP_W * STRIP_H)   /* 38,400 */
#define STRIP_WORDS   (STRIP_BYTES / 4)     /* 9,600  */

#define COLOR_STRIP_BG       0
#define COLOR_STRIP_WHITE    1
#define COLOR_STRIP_GRAY     2
#define COLOR_STRIP_GREEN    3
#define COLOR_STRIP_GOLD     4
#define COLOR_STRIP_RED      5
#define COLOR_STRIP_BLUE     6
#define COLOR_STRIP_BURLY    7

/* Once at startup: the caller passes the mmap'd LW-bridge byte pointer
 * (so this module doesn't need its own /dev/mem code). */
void strip_init(volatile uint8_t *lw_bridge_base, uint32_t strip_offset,
                uint32_t reg_strip_swap_offset);

/* Buffer manipulation (no HW writes — buffer stays local until present()). */
void strip_clear(uint8_t color);
void strip_pixel(int x, int y, uint8_t color);
void strip_rect(int x, int y, int w, int h, uint8_t color);

/* Draw 5×7 character at (x, y). 'scale' multiplies font dimensions:
 *   scale=1: 5×7 px glyph     scale=2: 10×14 px glyph     etc.
 * fg is drawn for set bits; bg is drawn for clear bits.  */
void strip_char(int x, int y, char c, int scale, uint8_t fg, uint8_t bg);
void strip_text(int x, int y, const char *s, int scale,
                uint8_t fg, uint8_t bg);
```

```
/* Pixel width of `s` painted at `scale`. Matches strip_text exactly: N chars
 * occupy N*FONT_W*scale + (N-1)*scale pixels (no trailing inter-char gap). */
int  strip_text_width(const char *s, int scale);

/* strip_text with the start-x computed so the string is horizontally centered
 * in the 640-px-wide strip. Clamps x ≥ 0 if the string overflows. */
void strip_text_centered(int y, const char *s, int scale,
                         uint8_t fg, uint8_t bg);

/* Filled `bg`-colored box sized to fit `s` plus `pad` px of padding on every
 * side, with `s` painted on top in `fg`. Use for "selected" pills on menus. */
void strip_text_box(int x, int y, const char *s, int scale,
                    uint8_t fg, uint8_t bg, int pad);

/* Push the local buffer to FPGA (9,600 word writes, ~770 µs at 50 MHz LW)
 * and arm a vsync-aligned swap. After this returns, the next VS pulse
 * makes the new contents visible. */
void strip_present(void);

#endif
```

usbkeyboard.c

```
C/C++
/*
 * USB HID keyboard via libusb-1.0
 *
 * Adapted from references/FlappyBird/source/4840-Flappy-Bird/FB_sw/usbkeyboard.c
 * (CSEE 4840 Spring 2025). Used verbatim except for include path / formatting.
 *
 * Library/protocol references:
 *   https://libusb.sourceforge.io/api-1.0/
 *   USB HID 1.11: https://www.usb.org/sites/default/files/documents/hid1_11.pdf
 *   USB HID Usage Tables 1.5: https://usb.org/sites/default/files/hut1_5.pdf
 */

#include "usbkeyboard.h"

#include <stdio.h>
#include <stdlib.h>

struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address)
{
    libusb_device **devs;
    struct libusb_device_handle *keyboard = NULL;
    struct libusb_device_descriptor desc;
    ssize_t num_devs, d;
    uint8_t i, k;

    if (libusb_init(NULL) < 0) {
        fprintf(stderr, "Error: libusb_init failed\n");
        exit(1);
```

```
    }

    if ((num_devs = libusb_get_device_list(NULL, &devs)) < 0) {
        fprintf(stderr, "Error: libusb_get_device_list failed\n");
        exit(1);
    }

    /* Walk every device. Take the first that exposes an HID interface
     * speaking the keyboard boot protocol. */
    for (d = 0; d < num_devs; d++) {
        libusb_device *dev = devs[d];
        if (libusb_get_device_descriptor(dev, &desc) < 0) {
            fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
            exit(1);
        }

        if (desc.bDeviceClass != LIBUSB_CLASS_PER_INTERFACE)
            continue;

        struct libusb_config_descriptor *config;
        libusb_get_config_descriptor(dev, 0, &config);
        for (i = 0; i < config->bNumInterfaces; i++) {
            for (k = 0; k < config->interface[i].num_altsetting; k++) {
                const struct libusb_interface_descriptor *inter =
                    config->interface[i].altsetting + k;
                if (inter->bInterfaceClass == LIBUSB_CLASS_HID &&
                    inter->bInterfaceProtocol == USB_HID_KEYBOARD_PROTOCOL) {
                    int r;
                    if ((r = libusb_open(dev, &keyboard)) != 0) {
                        fprintf(stderr,
                                "Error: libusb_open failed: %d\n", r);
                        exit(1);
                    }
                    if (libusb_kernel_driver_active(keyboard, i))
                        libusb_detach_kernel_driver(keyboard, i);
                    libusb_set_auto_detach_kernel_driver(keyboard, i);
                    if ((r = libusb_claim_interface(keyboard, i)) != 0) {
                        fprintf(stderr,
                                "Error: libusb_claim_interface failed: %d\n", r);
                        exit(1);
                    }
                    *endpoint_address = inter->endpoint[0].bEndpointAddress;
                    goto found;
                }
            }
        }
    }

found:
    libusb_free_device_list(devs, 1);
    return keyboard;
}
```

usbkeyboard.h

C/C++

```
/*
 * USB HID keyboard via libusb-1.0
 *
 * Adapted from references/FlappyBird/source/4840-Flappy-Bird/FB_sw/usbkeyboard.h
 * (CSEE 4840 Spring 2025). Used verbatim except for the include guard rename.
 *
 * The keyboard report format is the standard USB HID Boot Keyboard Report:
 *   uint8_t modifiers;     (Ctrl/Shift/Alt/GUI bitmask)
 *   uint8_t reserved;
 *   uint8_t keycode[6];    (up to 6 simultaneously held keys, scan codes
 *                           per USB HID Usage Tables 1.5, Keyboard/Keypad
 *                           page 0x07; 0 = no key in this slot)
 */

#ifndef _USBKEYBOARD_H
#define _USBKEYBOARD_H

#include <stdint.h>
#include <libusb-1.0/libusb.h>

#define USB_HID_KEYBOARD_PROTOCOL 1

/* Modifier bits (packet.modifiers) */
#define USB_LCTRL  (1 << 0)
#define USB_LSHIFT (1 << 1)
#define USB_LALT   (1 << 2)
#define USB_LGUI   (1 << 3)
#define USB_RCTRL  (1 << 4)
#define USB_RSHIFT (1 << 5)
#define USB_RALT   (1 << 6)
#define USB_RGUI   (1 << 7)

struct usb_keyboard_packet {
    uint8_t modifiers;
    uint8_t reserved;
    uint8_t keycode[6];
};

/*
 * Find and open a USB keyboard device.
 *
 * Argument: pointer to space where the function will store the endpoint
 * address used for libusb_interrupt_transfer.
 *
 * Returns: opened libusb_device_handle on success, NULL on failure (no
 * keyboard found / open failed).
 */
extern struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address);

#endif /* _USBKEYBOARD_H */
```

Hardware

go_peripheral.sv

```
None
/*
 * go_peripheral — Avalon memory-mapped VGA peripheral for the 9x9 Go game
 *
 * Phase 5: adds the score-strip framebuffer (640×60 8bpp double-buffered).
 *   - New second Avalon slave (strip_*) for the bulk pixel write window.
 *   - STRIP_SWAP register on the first slave (offset 0x05) arms a swap
 *     that fires on the next VS rising edge.
 *   - Strip region (py ∈ [420, 479]) sources pixels from the active strip
 *     buffer via a tiny inline 8-color palette.
 *
 * (Earlier phases: Phase 1 static board, Phase 2 tilemap + stones, Phase 3
 *  cursor + USB keyboard.)
 *
 * Register map (avalon_slave_0, byte-addressed):
 *   0x00  SET_BLACK    W   data = cell_idx → cells[idx] = BLACK
 *   0x01  SET_WHITE    W   data = cell_idx → cells[idx] = WHITE
 *   0x02  CLEAR_CELL   W   data = cell_idx → cells[idx] = EMPTY
 *   0x03  RESET_BOARD  W   any             → all cells = EMPTY
 *   0x04  CURSOR       W   {visible[7], cell_idx[6:0]}
 *   0x05  STRIP_SWAP   W   any             → swap strip buffers at next VS
 *   0x06..0x07         (reserved for Phase 8 — RENDER_MODE)
 *
 * Strip framebuffer (avalon_slave_1, 32-bit word-addressed):
 *   word 0..9599: 38,400 pixels of the BACK buffer (4 packed 8-bit pixels each)
 *
 * Pixel encoding (bottom 3 bits index a small palette):
 *   0: dark gray bg    1: white text       2: light gray
 *   3: green accent    4: gold (winner)    5: red (illegal/error)
 *   6: blue accent     7: burlywood
 */

module go_peripheral(
    input  logic        clk,
    input  logic        reset,

    // —— avalon_slave_0: control registers ————————————————————————————————
    input  logic [2:0]  address,
    input  logic        chipselect,
    input  logic        write,
    input  logic [7:0]  writedata,
    output logic [7:0]  readdata,        // for AUDIO_STATUS reads

    // —— avalon_slave_1: strip framebuffer write window ————————————————————
    input  logic [13:0] strip_address,
    input  logic        strip_chipselect,
    input  logic        strip_write,
    input  logic [31:0] strip_writedata,

    // VGA conduit
    output logic [7:0]  VGA_R, VGA_G, VGA_B,
```

```
    output logic        VGA_CLK, VGA_HS, VGA_VS,
                        VGA_BLANK_n, VGA_SYNC_n,

    // —— Audio conduit (to/from codec_interface in toplevel) ——————————————
    output logic [23:0] dac_left,
    output logic [23:0] dac_right,
    input  logic        advance
);

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

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

    // VS rising-edge detector → vsync_pulse for strip swap timing.
    logic vs_d;
    logic vsync_pulse;
    always_ff @(posedge clk) vs_d <= VGA_VS;
    assign vsync_pulse = (VGA_VS == 1'b1) && (vs_d == 1'b0);

    // ——— Avalon register decoder (slave 0) ————————————————————————————————
    localparam logic [2:0] REG_SET_BLACK    = 3'h0;
    localparam logic [2:0] REG_SET_WHITE    = 3'h1;
    localparam logic [2:0] REG_CLEAR_CELL   = 3'h2;
    localparam logic [2:0] REG_RESET_BOARD  = 3'h3;
    localparam logic [2:0] REG_CURSOR       = 3'h4;
    localparam logic [2:0] REG_STRIP_SWAP   = 3'h5;
    localparam logic [2:0] REG_AUDIO_CMD    = 3'h6;
    localparam logic [2:0] REG_AUDIO_STATUS = 3'h7;

    logic        bm_write_en, bm_reset_all;
    logic [6:0]  bm_write_addr;
    logic [1:0]  bm_write_data;
    logic        strip_swap_request;
    logic        audio_cmd_valid;
    logic [2:0]  audio_cmd_value;

    logic        cursor_visible;
    logic [6:0]  cursor_idx;

    always_comb begin
        bm_write_en        = 1'b0;
        bm_write_addr      = writedata[6:0];
```

```
    bm_write_data      = 2'b00;
    bm_reset_all       = 1'b0;
    strip_swap_request = 1'b0;
    audio_cmd_valid    = 1'b0;
    audio_cmd_value    = writedata[2:0];

    if (chipselect && write) begin
        unique case (address)
            REG_SET_BLACK:   begin bm_write_en = 1; bm_write_data = 2'b01; end
            REG_SET_WHITE:   begin bm_write_en = 1; bm_write_data = 2'b10; end
            REG_CLEAR_CELL:  begin bm_write_en = 1; bm_write_data = 2'b00; end
            REG_RESET_BOARD: bm_reset_all       = 1'b1;
            REG_CURSOR:      ;   // handled in always_ff below
            REG_STRIP_SWAP:  strip_swap_request = 1'b1;
            REG_AUDIO_CMD:   audio_cmd_valid    = 1'b1;
            default: ;
        endcase
    end
end

/* AUDIO_STATUS reads: bit 0 = busy. */
logic audio_busy;
always_comb begin
    if (chipselect && !write && address == REG_AUDIO_STATUS)
        readdata = {7'b0, audio_busy};
    else
        readdata = 8'h00;
end

always_ff @(posedge clk) begin
    if (reset) begin
        cursor_visible <= 1'b1;
        cursor_idx     <= 7'd40;
    end else if (chipselect && write && address == REG_CURSOR) begin
        cursor_visible <= writedata[7];
        cursor_idx     <= writedata[6:0];
    end
end

// ─── Geometry ───────────────────────────────────────────────────────────
localparam int BOARD_LEFT    = 131;
localparam int BOARD_TOP     = 24;
localparam int CELL_PITCH    = 42;
localparam int HALF_CELL     = 21;
localparam int BOARD_W       = 9 * CELL_PITCH;
localparam int BOARD_H       = 9 * CELL_PITCH;
localparam int STONE_R2_MAX  = 324;
localparam int STAR_R2_MAX   = 9;
localparam int STRIP_TOP     = 420;
localparam int STRIP_BOTTOM  = 479;
localparam int STRIP_HEIGHT  = STRIP_BOTTOM - STRIP_TOP + 1;   // 60

logic [9:0] px, py;
assign px = hcount[10:1];
```

```
assign py = vcount;

// ─── Board area pipeline ────────────────────────────────────────────────
logic in_board;
assign in_board = (px >= BOARD_LEFT) && (px < BOARD_LEFT + BOARD_W)
               && (py >= BOARD_TOP)  && (py < BOARD_TOP + BOARD_H);

logic [9:0] bx, by;
assign bx = px - BOARD_LEFT;
assign by = py - BOARD_TOP;

logic [3:0] col, row;
logic [9:0] local_x, local_y;
always_comb begin
    col     = bx / CELL_PITCH;
    row     = by / CELL_PITCH;
    local_x = bx - col * CELL_PITCH;
    local_y = by - row * CELL_PITCH;
end

logic [6:0] cell_idx;
assign cell_idx = row * 4'd9 + col;

logic [1:0] cell_value;
board_mem bm (
    .clk        (clk),
    .reset      (reset),
    .write_en   (bm_write_en),
    .write_addr (bm_write_addr),
    .write_data (bm_write_data),
    .reset_all  (bm_reset_all),
    .read_addr  (cell_idx),
    .read_data  (cell_value)
);

logic is_star_cell;
assign is_star_cell = (row == 4'd4 && col == 4'd4)
                   || (row == 4'd2 && (col == 4'd2 || col == 4'd6))
                   || (row == 4'd6 && (col == 4'd2 || col == 4'd6));

logic signed [10:0] dx, dy;
logic        [21:0] d2;
assign dx = $signed({1'b0, local_x}) - 11'sd21;
assign dy = $signed({1'b0, local_y}) - 11'sd21;
assign d2 = dx*dx + dy*dy;

logic in_stone, on_grid, in_star_dot, on_cursor;
assign in_stone    = (cell_value != 2'b00) && (d2 <= STONE_R2_MAX);
assign on_grid     = (local_x == HALF_CELL) || (local_y == HALF_CELL);
assign in_star_dot = is_star_cell && (cell_value == 2'b00) && (d2 <= STAR_R2_MAX);
assign on_cursor   = cursor_visible && (cursor_idx == cell_idx)
                  && (d2 >= 22'd264) && (d2 <= 22'd400);

// ─── Strip area pipeline ────────────────────────────────────────────────
```

```
logic in_strip;
assign in_strip = (py >= STRIP_TOP) && (py <= STRIP_BOTTOM);

// pixel_addr = (py - 420) * 640 + px ; range 0..38399 within strip.
// Quartus will pick the implementation; on Cyclone V it'll likely use
// a single 18×18 DSP block for the multiply, free for our purposes.
logic [15:0] strip_pixel_full;
logic [15:0] strip_pixel_addr;
assign strip_pixel_full = (py - 16'd420) * 16'd640 + px;
assign strip_pixel_addr = strip_pixel_full;        // full 16 bits → strip_fb

logic [7:0] strip_pixel;
strip_fb sfb (
    .clk          (clk),
    .reset        (reset),
    .write_en     (strip_chipselect && strip_write),
    .write_addr   (strip_address),
    .write_data   (strip_writedata),
    .swap_request (strip_swap_request),
    .vsync_pulse  (vsync_pulse),
    .pixel_addr   (strip_pixel_addr),
    .pixel_out    (strip_pixel)
);

// ─── Audio ──────────────────────────────────────────────────────────────
audio_controller ac (
    .clk             (clk),
    .reset           (reset),
    .audio_cmd       (audio_cmd_value),
    .audio_cmd_valid (audio_cmd_valid),
    .busy            (audio_busy),
    .advance         (advance),
    .dac_left        (dac_left),
    .dac_right       (dac_right)
);

// Strip palette — bottom 3 bits of the pixel byte index a small RGB table.
logic [23:0] strip_rgb;
always_comb begin
    unique case (strip_pixel[2:0])
        3'd0: strip_rgb = 24'h202020;   // dark gray bg
        3'd1: strip_rgb = 24'hF0F0F0;   // white text
        3'd2: strip_rgb = 24'h808080;   // light gray
        3'd3: strip_rgb = 24'h00C040;   // green accent
        3'd4: strip_rgb = 24'hFFD700;   // gold (winner)
        3'd5: strip_rgb = 24'hC04040;   // red (error)
        3'd6: strip_rgb = 24'h4080C0;   // blue accent
        3'd7: strip_rgb = 24'hDEB887;   // burlywood (board match)
    endcase
end

// ─── Color output ───────────────────────────────────────────────────────
localparam logic [23:0] COLOR_BG       = 24'h202020;
localparam logic [23:0] COLOR_BOARD_BG = 24'hDEB887;
```

```
    localparam logic [23:0] COLOR_LINE     = 24'h000000;
    localparam logic [23:0] COLOR_BLACK    = 24'h101010;
    localparam logic [23:0] COLOR_WHITE    = 24'hF0F0F0;
    localparam logic [23:0] COLOR_OUTLINE  = 24'h000000;
    localparam logic [23:0] COLOR_CURSOR   = 24'h00C040;

    always_comb begin
        if (!VGA_BLANK_n) begin
            {VGA_R, VGA_G, VGA_B} = 24'h000000;
        end else if (in_strip) begin
            {VGA_R, VGA_G, VGA_B} = strip_rgb;
        end else if (in_board) begin
            if (on_cursor) begin
                {VGA_R, VGA_G, VGA_B} = COLOR_CURSOR;
            end else if (in_stone) begin
                if (cell_value == 2'b01)
                    {VGA_R, VGA_G, VGA_B} = COLOR_BLACK;
                else if (d2 >= 22'd289)
                    {VGA_R, VGA_G, VGA_B} = COLOR_OUTLINE;
                else
                    {VGA_R, VGA_G, VGA_B} = COLOR_WHITE;
            end else if (on_grid || in_star_dot) begin
                {VGA_R, VGA_G, VGA_B} = COLOR_LINE;
            end else begin
                {VGA_R, VGA_G, VGA_B} = COLOR_BOARD_BG;
            end
        end else begin
            {VGA_R, VGA_G, VGA_B} = COLOR_BG;
        end
    end

endmodule


// ──────────────────────────────────────────────────────────────────────────────
// vga_counters — VERBATIM from lab3 vga_ball.sv (Stephen Edwards, Columbia)
// ──────────────────────────────────────────────────────────────────────────────
module vga_counters(
    input  logic        clk50, reset,
    output logic [10:0] hcount,
    output logic [9:0]  vcount,
    output logic        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n
);

    parameter HACTIVE      = 11'd 1280;
    parameter HFRONT_PORCH = 11'd 32;
    parameter HSYNC        = 11'd 192;
    parameter HBACK_PORCH  = 11'd 96;
    parameter HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH;

    parameter VACTIVE      = 10'd 480;
    parameter VFRONT_PORCH = 10'd 10;
    parameter VSYNC        = 10'd 2;
    parameter VBACK_PORCH  = 10'd 33;
```

```
    parameter VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH;

    logic endOfLine;
    always_ff @(posedge clk50 or posedge reset)
        if (reset)          hcount <= 0;
        else if (endOfLine) hcount <= 0;
        else                hcount <= hcount + 11'd 1;
    assign endOfLine = hcount == HTOTAL - 1;

    logic endOfField;
    always_ff @(posedge clk50 or posedge reset)
        if (reset)          vcount <= 0;
        else if (endOfLine)
            if (endOfField) vcount <= 0;
            else            vcount <= vcount + 10'd 1;
    assign endOfField = vcount == VTOTAL - 1;

    assign VGA_HS      = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111) );
    assign VGA_VS      = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2 );
    assign VGA_SYNC_n  = 1'b0;
    assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                         !( vcount[9] | (vcount[8:5] == 4'b1111) );
    assign VGA_CLK     = hcount[0];

endmodule
```

soc_system_top.sv

```
None
// ==================================================================
// Copyright (c) 2013 by Terasic Technologies Inc.
// ==================================================================
//
// Modified 2019 by Stephen A. Edwards
//
// Permission:
//
//   Terasic grants permission to use and modify this code for use
//   in synthesis for all Terasic Development Boards and Altera
//   Development Kits made by Terasic.  Other use of this code,
//   including the selling ,duplication, or modification of any
//   portion is strictly prohibited.
//
// Disclaimer:
//
//   This VHDL/Verilog or C/C++ source code is intended as a design
//   reference which illustrates how these types of functions can be
//   implemented.  It is the user's responsibility to verify their
//   design for consistency and functionality through the use of
//   formal verification methods.  Terasic provides no warranty
//   regarding the use or functionality of this code.
```

```
//
// ============================================================
//
//  Terasic Technologies Inc

//  9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan
//
//
//                     web: http://www.terasic.com/
//                     email: support@terasic.com
module soc_system_top(

 ///////// ADC /////////
 inout               ADC_CS_N,
 output        ADC_DIN,
 input               ADC_DOUT,
 output        ADC_SCLK,

 ///////// AUD /////////
 input               AUD_ADCDAT,
 inout               AUD_ADCLRCK,
 inout               AUD_BCLK,
 output        AUD_DACDAT,
 inout               AUD_DACLRCK,
 output        AUD_XCK,

 ///////// CLOCK2 /////////
 input               CLOCK2_50,

 ///////// CLOCK3 /////////
 input               CLOCK3_50,

 ///////// CLOCK4 /////////
 input               CLOCK4_50,

 ///////// CLOCK /////////
 input               CLOCK_50,

 ///////// DRAM /////////
 output [12:0] DRAM_ADDR,
 output [1:0]  DRAM_BA,
 output        DRAM_CAS_N,
 output        DRAM_CKE,
 output        DRAM_CLK,
 output        DRAM_CS_N,
 inout [15:0]  DRAM_DQ,
 output        DRAM_LDQM,
 output        DRAM_RAS_N,
 output        DRAM_UDQM,
 output        DRAM_WE_N,

 ///////// FAN /////////
 output        FAN_CTRL,
```

```
///////// FPGA /////////
output        FPGA_I2C_SCLK,
inout              FPGA_I2C_SDAT,

///////// GPIO /////////
inout [35:0]  GPIO_0,
inout [35:0]  GPIO_1,

///////// HEX0 /////////
output [6:0]  HEX0,

///////// HEX1 /////////
output [6:0]  HEX1,

///////// HEX2 /////////
output [6:0]  HEX2,

///////// HEX3 /////////
output [6:0]  HEX3,

///////// HEX4 /////////
output [6:0]  HEX4,

///////// HEX5 /////////
output [6:0]  HEX5,

///////// HPS /////////
inout              HPS_CONV_USB_N,
output [14:0] HPS_DDR3_ADDR,
output [2:0]  HPS_DDR3_BA,
output        HPS_DDR3_CAS_N,
output        HPS_DDR3_CKE,
output        HPS_DDR3_CK_N,
output        HPS_DDR3_CK_P,
output        HPS_DDR3_CS_N,
output [3:0]  HPS_DDR3_DM,
inout [31:0]  HPS_DDR3_DQ,
inout [3:0]   HPS_DDR3_DQS_N,
inout [3:0]   HPS_DDR3_DQS_P,
output        HPS_DDR3_ODT,
output        HPS_DDR3_RAS_N,
output        HPS_DDR3_RESET_N,
input              HPS_DDR3_RZQ,
output        HPS_DDR3_WE_N,
output        HPS_ENET_GTX_CLK,
inout              HPS_ENET_INT_N,
output        HPS_ENET_MDC,
inout              HPS_ENET_MDIO,
input              HPS_ENET_RX_CLK,
input [3:0]   HPS_ENET_RX_DATA,
input              HPS_ENET_RX_DV,
output [3:0]  HPS_ENET_TX_DATA,
output        HPS_ENET_TX_EN,
inout              HPS_GSENSOR_INT,
```

```
inout               HPS_I2C1_SCLK,
inout               HPS_I2C1_SDAT,
inout               HPS_I2C2_SCLK,
inout               HPS_I2C2_SDAT,
inout               HPS_I2C_CONTROL,
inout               HPS_KEY,
inout               HPS_LED,
inout               HPS_LTC_GPIO,
output        HPS_SD_CLK,
inout               HPS_SD_CMD,
inout [3:0]   HPS_SD_DATA,
output        HPS_SPIM_CLK,
input               HPS_SPIM_MISO,
output        HPS_SPIM_MOSI,
inout               HPS_SPIM_SS,
input               HPS_UART_RX,
output        HPS_UART_TX,
input               HPS_USB_CLKOUT,
inout [7:0]   HPS_USB_DATA,
input               HPS_USB_DIR,
input               HPS_USB_NXT,
output        HPS_USB_STP,

///////// IRDA /////////
input               IRDA_RXD,
output        IRDA_TXD,

///////// KEY /////////
input [3:0]   KEY,

///////// LEDR /////////
output [9:0]  LEDR,

///////// PS2 /////////
inout               PS2_CLK,
inout               PS2_CLK2,
inout               PS2_DAT,
inout               PS2_DAT2,

///////// SW /////////
input [9:0]   SW,

///////// TD /////////
input               TD_CLK27,
input [7:0]   TD_DATA,
input               TD_HS,
output        TD_RESET_N,
input               TD_VS,


///////// VGA /////////
output [7:0]  VGA_B,
output        VGA_BLANK_N,
output        VGA_CLK,
```

```
 output [7:0]  VGA_G,
 output        VGA_HS,
 output [7:0]  VGA_R,
 output        VGA_SYNC_N,
 output        VGA_VS
);

   soc_system soc_system0(
     .clk_clk                      ( CLOCK_50 ),
     .reset_reset_n                ( 1'b1 ),

     .hps_ddr3_mem_a               ( HPS_DDR3_ADDR ),
     .hps_ddr3_mem_ba              ( HPS_DDR3_BA ),
     .hps_ddr3_mem_ck              ( HPS_DDR3_CK_P ),
     .hps_ddr3_mem_ck_n            ( HPS_DDR3_CK_N ),
     .hps_ddr3_mem_cke             ( HPS_DDR3_CKE ),
     .hps_ddr3_mem_cs_n            ( HPS_DDR3_CS_N ),
     .hps_ddr3_mem_ras_n           ( HPS_DDR3_RAS_N ),
     .hps_ddr3_mem_cas_n           ( HPS_DDR3_CAS_N ),
     .hps_ddr3_mem_we_n            ( HPS_DDR3_WE_N ),
     .hps_ddr3_mem_reset_n         ( HPS_DDR3_RESET_N ),
     .hps_ddr3_mem_dq              ( HPS_DDR3_DQ ),
     .hps_ddr3_mem_dqs             ( HPS_DDR3_DQS_P ),
     .hps_ddr3_mem_dqs_n           ( HPS_DDR3_DQS_N ),
     .hps_ddr3_mem_odt             ( HPS_DDR3_ODT ),
     .hps_ddr3_mem_dm              ( HPS_DDR3_DM ),
     .hps_ddr3_oct_rzqin           ( HPS_DDR3_RZQ ),

     .hps_hps_io_emac1_inst_TX_CLK ( HPS_ENET_GTX_CLK ),
     .hps_hps_io_emac1_inst_TXD0   ( HPS_ENET_TX_DATA[0] ),
     .hps_hps_io_emac1_inst_TXD1   ( HPS_ENET_TX_DATA[1] ),
     .hps_hps_io_emac1_inst_TXD2   ( HPS_ENET_TX_DATA[2] ),
     .hps_hps_io_emac1_inst_TXD3   ( HPS_ENET_TX_DATA[3] ),
     .hps_hps_io_emac1_inst_RXD0   ( HPS_ENET_RX_DATA[0] ),
     .hps_hps_io_emac1_inst_MDIO   ( HPS_ENET_MDIO  ),
     .hps_hps_io_emac1_inst_MDC    ( HPS_ENET_MDC   ),
     .hps_hps_io_emac1_inst_RX_CTL ( HPS_ENET_RX_DV ),
     .hps_hps_io_emac1_inst_TX_CTL ( HPS_ENET_TX_EN ),
     .hps_hps_io_emac1_inst_RX_CLK ( HPS_ENET_RX_CLK ),
     .hps_hps_io_emac1_inst_RXD1   ( HPS_ENET_RX_DATA[1]  ),
     .hps_hps_io_emac1_inst_RXD2   ( HPS_ENET_RX_DATA[2]  ),
     .hps_hps_io_emac1_inst_RXD3   ( HPS_ENET_RX_DATA[3]  ),

     .hps_hps_io_sdio_inst_CMD     ( HPS_SD_CMD          ),
     .hps_hps_io_sdio_inst_D0      ( HPS_SD_DATA[0]      ),
     .hps_hps_io_sdio_inst_D1      ( HPS_SD_DATA[1]      ),
     .hps_hps_io_sdio_inst_CLK     ( HPS_SD_CLK          ),
     .hps_hps_io_sdio_inst_D2      ( HPS_SD_DATA[2]      ),
     .hps_hps_io_sdio_inst_D3      ( HPS_SD_DATA[3]      ),

     .hps_hps_io_usb1_inst_D0      ( HPS_USB_DATA[0]     ),
     .hps_hps_io_usb1_inst_D1      ( HPS_USB_DATA[1]     ),
     .hps_hps_io_usb1_inst_D2      ( HPS_USB_DATA[2]     ),
     .hps_hps_io_usb1_inst_D3      ( HPS_USB_DATA[3]     ),
```

```
    .hps_hps_io_usb1_inst_D4      ( HPS_USB_DATA[4]     ),
    .hps_hps_io_usb1_inst_D5      ( HPS_USB_DATA[5]     ),
    .hps_hps_io_usb1_inst_D6      ( HPS_USB_DATA[6]     ),
    .hps_hps_io_usb1_inst_D7      ( HPS_USB_DATA[7]     ),
    .hps_hps_io_usb1_inst_CLK     ( HPS_USB_CLKOUT      ),
    .hps_hps_io_usb1_inst_STP     ( HPS_USB_STP         ),
    .hps_hps_io_usb1_inst_DIR     ( HPS_USB_DIR         ),
    .hps_hps_io_usb1_inst_NXT     ( HPS_USB_NXT         ),

    .hps_hps_io_spim1_inst_CLK    ( HPS_SPIM_CLK  ),
    .hps_hps_io_spim1_inst_MOSI   ( HPS_SPIM_MOSI ),
    .hps_hps_io_spim1_inst_MISO   ( HPS_SPIM_MISO ),
    .hps_hps_io_spim1_inst_SS0    ( HPS_SPIM_SS   ),

    .hps_hps_io_uart0_inst_RX     ( HPS_UART_RX     ),
    .hps_hps_io_uart0_inst_TX     ( HPS_UART_TX     ),

    .hps_hps_io_i2c0_inst_SDA     ( HPS_I2C1_SDAT     ),
    .hps_hps_io_i2c0_inst_SCL     ( HPS_I2C1_SCLK     ),

    .hps_hps_io_i2c1_inst_SDA     ( HPS_I2C2_SDAT     ),
    .hps_hps_io_i2c1_inst_SCL     ( HPS_I2C2_SCLK     ),

    .hps_hps_io_gpio_inst_GPIO09  ( HPS_CONV_USB_N ),
    .hps_hps_io_gpio_inst_GPIO35  ( HPS_ENET_INT_N ),
    .hps_hps_io_gpio_inst_GPIO40  ( HPS_LTC_GPIO ),

    .hps_hps_io_gpio_inst_GPIO48  ( HPS_I2C_CONTROL ),
    .hps_hps_io_gpio_inst_GPIO53  ( HPS_LED ),
    .hps_hps_io_gpio_inst_GPIO54  ( HPS_KEY ),
    .hps_hps_io_gpio_inst_GPIO61  ( HPS_GSENSOR_INT ),

    // Phase 1 + Phase 5: VGA conduit
    .vga_r       (VGA_R),
    .vga_g       (VGA_G),
    .vga_b       (VGA_B),
    .vga_clk     (VGA_CLK),
    .vga_hs      (VGA_HS),
    .vga_vs      (VGA_VS),
    .vga_blank_n (VGA_BLANK_N),
    .vga_sync_n  (VGA_SYNC_N),

    // Phase 6: audio conduit (wired to codec_interface below)
    .audio_dac_left  (aud_dac_left),
    .audio_dac_right (aud_dac_right),
    .audio_advance   (aud_advance)
);

  // Phase 6: Audio CODEC wrapper (Altera-UP, sits at toplevel, NOT in Qsys).
  // codec_interface drives AUD_x and FPGA_I2C_x board pins directly and
  // exposes a 48 kHz sample-rate "advance" strobe to our audio_controller.
  wire [23:0] aud_dac_left, aud_dac_right;
  wire [23:0] aud_adc_left_unused, aud_adc_right_unused;
  wire        aud_advance;
```

```
codec_interface codec_inst (
    .CLOCK_50      (CLOCK_50),
    .reset         (1'b0),
    .dac_left      (aud_dac_left),
    .dac_right     (aud_dac_right),
    .adc_left      (aud_adc_left_unused),
    .adc_right     (aud_adc_right_unused),
    .advance       (aud_advance),
    .FPGA_I2C_SCLK (FPGA_I2C_SCLK),
    .FPGA_I2C_SDAT (FPGA_I2C_SDAT),
    .AUD_XCK       (AUD_XCK),
    .AUD_DACLRCK   (AUD_DACLRCK),
    .AUD_ADCLRCK   (AUD_ADCLRCK),
    .AUD_BCLK      (AUD_BCLK),
    .AUD_ADCDAT    (AUD_ADCDAT),
    .AUD_DACDAT    (AUD_DACDAT)
);

// The following quiet the "no driver" warnings for output
// pins and should be removed if you use any of these peripherals

assign ADC_CS_N = SW[1] ? SW[0] : 1'bZ;
assign ADC_DIN = SW[0];
assign ADC_SCLK = SW[0];

assign DRAM_ADDR = { 13{ SW[0] } };
assign DRAM_BA = { 2{ SW[0] } };
assign DRAM_DQ = SW[1] ? { 16{ SW[0] } } : { 16{ 1'bZ } };
assign {DRAM_CAS_N, DRAM_CKE, DRAM_CLK, DRAM_CS_N,
       DRAM_LDQM, DRAM_RAS_N, DRAM_UDQM, DRAM_WE_N} = { 8{SW[0]} };

assign FAN_CTRL = SW[0];

assign GPIO_0 = SW[1] ? { 36{ SW[0] } } : { 36{ 1'bZ } };
assign GPIO_1 = SW[1] ? { 36{ SW[0] } } : { 36{ 1'bZ } };

assign HEX0 = { 7{ SW[1] } };
assign HEX1 = { 7{ SW[2] } };
assign HEX2 = { 7{ SW[3] } };
assign HEX3 = { 7{ SW[4] } };
assign HEX4 = { 7{ SW[5] } };
assign HEX5 = { 7{ SW[6] } };

assign IRDA_TXD = SW[0];

assign LEDR = { 10{SW[7]} };

assign PS2_CLK = SW[1] ? SW[0] : 1'bZ;
assign PS2_CLK2 = SW[1] ? SW[0] : 1'bZ;
assign PS2_DAT = SW[1] ? SW[0] : 1'bZ;
assign PS2_DAT2 = SW[1] ? SW[0] : 1'bZ;

assign TD_RESET_N = SW[0];
```

```
endmodule
```

```
None
```

board_mem.sv

```
None
/*
 * board_mem — 9x9 Go board tilemap (81 cells × 2 bits = 162 bits)
 *
 * Each cell holds a Stone value:
 *   2'b00 = EMPTY
 *   2'b01 = BLACK
 *   2'b10 = WHITE
 *   2'b11 = (reserved — currently unused)
 *
 * Synchronous write port (driven by the Avalon register decoder),
 * combinational read port (driven by the VGA scanout pipeline).
 *
 * On reset OR reset_all (a synchronous "clear board" pulse from the
 * RESET_BOARD register), all 81 cells return to EMPTY in one cycle.
 *
 * 162 bits is well below the M10K threshold; Quartus will infer LUT-RAM
 * or simple registers, which is what we want — combinational read with
 * zero latency keeps the VGA pipeline simple.
 *
 * Reference: design-document.md §10.3 (board_mem interface),
 *            references/Chess/source/hw/board_mem.sv (analogous tilemap).
 */

module board_mem (
    input  logic        clk,
    input  logic        reset,        // active high

    // Write port (from Avalon register decoder)
    input  logic        write_en,
    input  logic [6:0]  write_addr,   // cell index 0..80 (row*9 + col)
    input  logic [1:0]  write_data,   // Stone value
    input  logic        reset_all,    // synchronous clear-all pulse

    // Read port (combinational, to VGA scanout)
    input  logic [6:0]  read_addr,
    output logic [1:0]  read_data
);

    logic [1:0] cells [0:80];
```

```
    always_ff @(posedge clk) begin
        if (reset || reset_all) begin
            for (int i = 0; i < 81; i++) cells[i] <= 2'b00;
        end else if (write_en && write_addr < 7'd81) begin
            cells[write_addr] <= write_data;
        end
    end

    assign read_data = (read_addr < 7'd81) ? cells[read_addr] : 2'b00;

endmodule
```

audio_controller.sv

```
None
/*
 * audio_controller — sample-ROM playback for game sound effects
 *
 * Reads 16-bit signed PCM samples at 8 kHz from one of four ROM banks and
 * outputs them on `dac_left` / `dac_right` whenever `advance` (from the
 * Altera-UP codec_interface, ~48 kHz) pulses. Each ROM sample is held for
 * 6 advance pulses (8 kHz × 6 = 48 kHz) — a zero-order-hold upsample with
 * no anti-aliasing. Audio quality is more than fine for short SFX (clicks,
 * captures, game-over chime).
 *
 * Sound banks (one-hot via the `sound_id` 3-bit code, 0=idle):
 *   1 = place       (0.20 s = 1600 samples = 25,600 bits ≈ 3 M10K)
 *   2 = capture     (0.30 s = 2400 samples ≈ 4 M10K)
 *   3 = illegal     (0.15 s = 1200 samples ≈ 2 M10K)
 *   4 = game_over   (1.50 s = 12000 samples ≈ 19 M10K)
 *
 * Sample data lives in `*.vh` files generated from .wav via the
 * Autotune-style python pipeline described in design-document.md §5.2.
 * For Phase 6 these files contain placeholder data (silence + 1-sample
 * click at start) — replace with real audio in Phase 6b.
 *
 * Avalon control:
 *   - Software writes AUDIO_CMD with values 1..4 to start playback. Writes
 *     while busy are ignored.
 *   - AUDIO_STATUS[0] is asserted while playing.
 *
 * Reference: design-document.md §5.3 (audio FSM); references/Pac-Man/
 * source/PacMan/hardware/audio.vh (similar $readmemh ROM pattern).
 */

module audio_controller (
    input  logic        clk,
    input  logic        reset,

    // Command interface (from Avalon register decoder)
    input  logic [2:0]  audio_cmd,        // 0=idle, 1..4 = sound id
```

```
    input  logic        audio_cmd_valid,  // 1-cycle pulse on new write
    output logic        busy,

    // codec_interface signals
    input  logic        advance,          // ~48 kHz strobe from codec_interface
    output logic [23:0] dac_left,
    output logic [23:0] dac_right
);

    // ─── ROMs (placeholder; populated by $readmemh) ─────────────────────────
    localparam int LEN_PLACE    = 1600;
    localparam int LEN_CAPTURE  = 2400;
    localparam int LEN_ILLEGAL  = 1200;
    localparam int LEN_GAMEOVER = 12000;

    logic signed [15:0] rom_place    [0:LEN_PLACE   -1];
    logic signed [15:0] rom_capture  [0:LEN_CAPTURE -1];
    logic signed [15:0] rom_illegal  [0:LEN_ILLEGAL -1];
    logic signed [15:0] rom_gameover [0:LEN_GAMEOVER-1];

    initial begin
        $readmemh("place.vh",    rom_place);
        $readmemh("capture.vh",  rom_capture);
        $readmemh("illegal.vh",  rom_illegal);
        $readmemh("gameover.vh", rom_gameover);
    end

    // ─── Playback state ─────────────────────────────────────────────────────
    typedef enum logic [0:0] { S_IDLE, S_PLAY } state_t;
    state_t      state;
    logic [2:0]  cur_sound;
    logic [13:0] sample_idx;       // up to 16383 — covers gameover (12000)
    logic [2:0]  sub_count;        // 0..5: hold each ROM sample for 6 advances

    logic [13:0] cur_length;
    always_comb begin
        unique case (cur_sound)
            3'd1:    cur_length = LEN_PLACE   [13:0];
            3'd2:    cur_length = LEN_CAPTURE [13:0];
            3'd3:    cur_length = LEN_ILLEGAL [13:0];
            3'd4:    cur_length = LEN_GAMEOVER[13:0];
            default: cur_length = 14'd0;
        endcase
    end

    logic signed [15:0] cur_sample;
    always_comb begin
        unique case (cur_sound)
            3'd1:    cur_sample = rom_place   [sample_idx[10:0]];   // <2048
            3'd2:    cur_sample = rom_capture [sample_idx[11:0]];   // <4096
            3'd3:    cur_sample = rom_illegal [sample_idx[10:0]];
            3'd4:    cur_sample = rom_gameover[sample_idx];
            default: cur_sample = 16'sd0;
        endcase
```

```
    end

    assign busy = (state != S_IDLE);

    // 16-bit signed → 24-bit signed (sign-extend)
    wire signed [23:0] cur24 = {{8{cur_sample[15]}}, cur_sample};
    assign dac_left  = (state == S_PLAY) ? cur24 : 24'sd0;
    assign dac_right = (state == S_PLAY) ? cur24 : 24'sd0;

    always_ff @(posedge clk) begin
        if (reset) begin
            state      <= S_IDLE;
            cur_sound  <= 3'd0;
            sample_idx <= 14'd0;
            sub_count  <= 3'd0;
        end else begin
            unique case (state)
                S_IDLE: begin
                    if (audio_cmd_valid && audio_cmd != 3'd0
                                        && audio_cmd <= 3'd4) begin
                        cur_sound  <= audio_cmd;
                        sample_idx <= 14'd0;
                        sub_count  <= 3'd0;
                        state      <= S_PLAY;
                    end
                end
                S_PLAY: begin
                    /* New audio_cmd while busy: ignored. */
                    if (advance) begin
                        if (sub_count == 3'd5) begin
                            sub_count <= 3'd0;
                            if (sample_idx + 14'd1 >= cur_length) begin
                                state <= S_IDLE;
                            end else begin
                                sample_idx <= sample_idx + 14'd1;
                            end
                        end else begin
                            sub_count <= sub_count + 3'd1;
                        end
                    end
                end
            endcase
        end
    end

endmodule
```

strip_fb.sv

```
None
/*
```

```
 * strip_fb — 640×60 8bpp double-buffered framebuffer
 *
 * Holds the score panel / menu / game-over text region at the bottom of
 * the screen (y ∈ [420, 479]).
 *
 * Two banks, each 9,600 × 32-bit words = 307,200 bits ≈ 30 M10K blocks.
 * Each 32-bit word holds 4 packed 8-bit pixels (little-endian: bits [7:0]
 * are the leftmost pixel of the four).
 *
 * Write side (Avalon): the HPS writes to the BACK buffer. Software flushes
 * the entire 38,400-byte buffer in a tight loop (~770 µs at 50 MHz LW),
 * then writes STRIP_SWAP to arm a swap. The swap takes effect on the next
 * rising edge of vsync_pulse, so the visible buffer never tears.
 *
 * Read side (VGA): for every pixel inside the strip region, the active
 * (front) buffer is indexed and the corresponding 8-bit pixel returned.
 * BRAM read has 1-cycle latency, which the consumer must account for.
 *
 * Pixel address layout: pixel_addr = (py - 420) * 640 + px, range 0..38399.
 *   word_idx = pixel_addr[13:2]
 *   byte_idx = pixel_addr[1:0]
 *
 * Reference: design-document.md §4.3 (strip framebuffer).
 */

module strip_fb (
    input  logic        clk,
    input  logic        reset,

    // Avalon-write side (back buffer)
    input  logic        write_en,
    input  logic [13:0] write_addr,        // 0..9599 word index
    input  logic [31:0] write_data,
    input  logic        swap_request,      // 1-cycle pulse from STRIP_SWAP
    input  logic        vsync_pulse,       // 1-cycle pulse at start of vblank

    // VGA-read side (front buffer)
    input  logic [15:0] pixel_addr,        // 0..38399 pixel index (needs 16 bits)
    output logic [7:0]  pixel_out
);

    /* Two banks. We keep them as separate arrays so Quartus infers two
     * dual-port BRAMs (write port + read port each) — much easier to time
     * than a single shared BRAM with mux'd ports. */
    logic [31:0] buf_a [0:9599];
    logic [31:0] buf_b [0:9599];

    logic active;        // 0 = buf_a is FRONT (read), 1 = buf_b is FRONT
    logic swap_armed;

    /* Write port: writes go to whichever bank is currently BACK. */
    wire write_to_a = write_en && (active == 1'b1);
    wire write_to_b = write_en && (active == 1'b0);
```

```
    always_ff @(posedge clk) begin
        if (write_to_a) buf_a[write_addr] <= write_data;
    end
    always_ff @(posedge clk) begin
        if (write_to_b) buf_b[write_addr] <= write_data;
    end

    /* Read port: synchronous reads from each bank. */
    logic [13:0] read_word_idx;
    logic [1:0]  read_byte_idx;
    assign read_word_idx = pixel_addr[15:2];   // top 14 bits → word index (max 9599)
    assign read_byte_idx = pixel_addr[1:0];

    logic [31:0] read_word_a, read_word_b;
    logic [1:0]  byte_idx_d;
    logic        active_d;

    always_ff @(posedge clk) begin
        read_word_a <= buf_a[read_word_idx];
        read_word_b <= buf_b[read_word_idx];
        byte_idx_d  <= read_byte_idx;
        active_d    <= active;
    end

    logic [31:0] front_word;
    assign front_word = (active_d == 1'b0) ? read_word_a : read_word_b;

    /* Pixel select within word — combinational from the registered word. */
    always_comb begin
        unique case (byte_idx_d)
            2'd0: pixel_out = front_word[7:0];
            2'd1: pixel_out = front_word[15:8];
            2'd2: pixel_out = front_word[23:16];
            2'd3: pixel_out = front_word[31:24];
        endcase
    end

    /* Swap state machine: arm on swap_request, fire on vsync_pulse. */
    always_ff @(posedge clk) begin
        if (reset) begin
            active     <= 1'b0;
            swap_armed <= 1'b0;
        end else begin
            if (swap_request)
                swap_armed <= 1'b1;
            if (swap_armed && vsync_pulse) begin
                active     <= ~active;
                swap_armed <= 1'b0;
            end
        end
    end

endmodule
```