Pop'n Engine: A Hardware-Software Co-Designed Rhythm Game

Final Project Report

CSEE 4840 — Embedded System Design

Vince-Arvin Magno (vm2787)

Columbia University

Spring 2026

1. Abstract and Project Overview

This report describes the design and implementation of Pop'n Engine, a hardware-software co-designed recreation of Konami's arcade rhythm game Pop'n Music, running on an Altera Cyclone V DE1-SoC development board. The project pairs a custom SystemVerilog VGA rendering engine on the FPGA fabric with a C-based game runtime executing on the on-board ARM Cortex-A9 hard processor system (HPS) under a Linaro Linux kernel.

The original project proposal called for a nine-lane falling-note rhythm game with synchronized audio playback, hit-timing scoring, and a complete user-interface flow. The implementation realizes all of these requirements and adds substantial polish beyond the proposal: arcade-accurate “plushie” character notes rendered directly from combinational logic, a hardware bitmap font that draws on-screen score, combo, and statistics text without consuming any block RAM, a segmented Groove Gauge styled after the arcade original, and a hardware-rendered button row at the bottom of the playfield that responds to the player's input in real time.

The system is partitioned along a clean hardware/software boundary. The FPGA owns everything visual and timing-critical at the pixel level: VGA timing generation at 640×480 resolution at 60 Hz, sprite compositing, text rendering, and the memory-mapped register interface. The HPS owns everything stateful: chart loading, the falling-note simulation, hit detection against the player's button presses, scoring, the Menu/Playing/Results state machine, and audio playback coordination. The two halves communicate over the heavyweight HPS-to-FPGA AXI bridge through a set of memory-mapped registers exposed by the custom Avalon-MM peripheral.

Core deliverables include: chart-driven note spawning across nine lanes; pixel-perfect timing math that places the white-stripe center of each note exactly on the judgment line at the perfect-hit moment; PERFECT, GOOD, and MISS judgment windows of 60 ms, 120 ms, and 150 ms respectively; live score, combo, and per-judgment counters rendered entirely in hardware; and a full state machine covering Menu, Playing, and Results screens. Audio is decoupled into a separate forked process that streams a 48 kHz WAV file to the on-board WM8731 audio codec while the main game loop drives the visuals.

The final system runs at a stable 60 Hz, is responsive to a custom nine-button Pop'n controller with sub-frame input latency, and produces a visually polished arcade experience that closely tracks the original game.

2. Detailed Project Design

2.1 System Architecture

The system is built around the Cyclone V SoC's tight HPS-to-FPGA coupling. A standard Linaro-based Linux distribution boots on the HPS from the SD card; the FPGA is configured at boot time by the U-Boot bootloader, which loads the project bitstream (“soc_system.rbf”) and brings the HPS-to-FPGA

bridges out of reset. Once Linux is up, the userspace game binary opens /dev/mem and memory-maps the heavyweight bridge region (base address 0xC0000000), giving it direct access to the custom popn_engine peripheral's register file.

The Qsys (Platform Designer) system instantiates popn_engine as an Avalon-MM slave on the heavyweight bridge, alongside the existing audio_0 and audio_and_video_config_0 IP blocks which remain on the lightweight bridge for audio I/O. The popn_engine occupies sixteen 32-bit registers (a 64-byte address span). Its outputs drive the VGA DAC pins directly: vga_r, vga_g, vga_b, plus the horizontal sync, vertical sync, blank, and pixel-clock signals.

2.2 Hardware Design

2.2.1 Register Map

Communication with the C runtime is entirely through memory-mapped registers. The register layout was chosen to match the natural granularity of game state updates: per-frame note positions in their own slots, while scoring and statistics are packed efficiently to fit the full HUD into the 16-register address window.

Offset	Register	Purpose
0x00	lane_glow_mask	9-bit bitmask of currently pressed lanes; lights up the bottom button row when set
0x04 - 0x24	note_y[0..8]	10-bit Y position of the head note in each of the nine lanes; 600 = hidden
0x28	score_bcd	6-digit BCD score, displayed in yellow at the top-left during play
0x2C	combo_bcd	4-digit BCD combo counter, displayed in cyan at the top-right
0x30	judgment	2-bit judgment code: 0=none, 1=COOL, 2=GOOD, 3=MISS; drives the floating message
0x34	groove	8-bit Groove Gauge level (0-255), drives the segmented bar at the bottom
0x38	state + miss	Packed: bits[1:0] game_state (Menu/Play/Results), bits[31:16] miss_bcd
0x3C	cool + good	Packed: bits[31:16] cool_bcd, bits[15:0] good_bcd; drives bottom-left counters

2.2.2 VGA Timing and Pixel Generation

The engine generates standard VGA 640×480 at 60 Hz with an approximately 25 MHz pixel clock derived by toggling on every input clock edge from the 50 MHz Qsys clock. Horizontal and vertical counters (hcount, vcount) cycle through the 800×525-pixel total frame and produce the active-region blanking signal vga_blank_n along with the standard sync pulses.

Every visible pixel is decided by a single combinational always_comb block that selects the output color from a priority-ordered set of layers. The priority order is: text overlays (menu, results, judgment

messages, statistic labels) at the top, then HUD digits (score, combo, hit counters), then the Groove Gauge, then notes, then the strike line and the bottom button row, and finally the playfield background. Because the entire pixel pipeline is combinational, there is no frame buffer and no need for any block RAM beyond small constant tables that infer as logic.

2.2.3 Bitmap Font and Text Rendering

Score, combo, hit counters, the menu prompt, and judgment messages all share a single hardware text renderer. A 3×5-pixel bitmap font is stored as a constant array of 32 fifteen-bit entries in the Verilog source. Each character glyph is one entry indexed by character code; the renderer is exposed as a SystemVerilog function that takes a character code, an origin (x0, y0), a scale factor, and the current (hcount, vcount), and returns a single bit indicating whether the current pixel is set in that glyph. Higher-level functions render multi-digit BCD values by computing the digit index from the relative x-offset and dispatching to render_char. Three flavors exist: render_3digit, render_4digit, and render_6digit, used respectively for the COOL/GOOD/MISS counters, the combo display, and the score display.

The scale parameter is a runtime input rather than a compile-time constant, which lets the same pipeline draw a small two-pixel-tall counter at the corner of the screen and a six-pixel-tall scale-six judgment banner across the middle without duplicating any logic. The end-of-game Results screen reuses the same renderer at scale six to draw the final score in the center of the screen.

2.2.4 Sprite Rendering: Plushie Notes and Buttons

The notes are rendered as oval base shapes with a stylized character face. The render_note function takes a center coordinate (cx, cy) and the current pixel position, and returns one of four pixel types: empty (outside the sprite), base color, white (sclera or middle stripe), or black (pupils). The oval boundary is approximated as |dx| + |dy| <= 30 with absolute clamps at |dx| <= 20, |dy| <= 15. Two small sclera regions sit at offsets (±8, -6) within the oval, each with a tiny black pupil at its center, and a one-pixel-thick white horizontal stripe runs across the middle. The base color is selected from a per-lane palette mapped to the arcade Pop'n color sequence (white, yellow, green, blue, red, blue, green, yellow, white from left to right).

The button row at the bottom of the playfield uses a similar pattern. Each lane has a circular button drawn at y=465 with a radius of 22 pixels. A dark rim runs from radius 17 to 22, the interior is filled with the lane's base color, and a small white highlight is placed in the upper-left of the disk to give a glossy look. When the corresponding bit of lane_glow_mask is asserted by the C runtime, the interior switches from the dark resting color to the bright lane color, providing instant visual feedback to the player without any C-side polling overhead.

Crucially, neither the note sprites nor the button sprites use any block RAM. Every pixel decision is the result of small integer arithmetic on (dx, dy) offsets, which synthesizes into comparators and adders. This keeps the design's RAM utilization at zero and makes it easy to add additional sprite variations in the future without worrying about memory layout.

2.2.5 Groove Gauge and Judgment Line

The Groove Gauge is rendered as 40 individual pill segments stretching across the bottom of the screen from x=120 to x=520. Each pill is 8 pixels wide separated by 2-pixel gaps. The render_groove_px function converts the 8-bit groove level into a fill width and tags each pixel as either filled (cyan), empty (dim purple), or transparent (gap), mirroring the segmented look of the arcade original. The judgment line itself is a four-pixel-thick white horizontal band at y=438–441, drawn only inside the playfield region and only while the game is in the Playing state.

2.2.6 State-Driven Compositing

The game_state register switches the entire visual composition between three modes. In Menu (state 0), notes are hidden, the playfield is dimmed, and the “PRESS START” prompt is drawn in the center along with a yellow “POPN” title at the top. In Playing (state 1), the full HUD becomes active: notes fall, the strike line appears, the button row is visible, the Groove Gauge animates, and judgment messages flash on hit. In Results (state 2), notes disappear, the playfield dims again, the small HUD digits hide, and a large 6×-scale score is rendered in the center under a cyan “SCORE” label.

2.3 Software Design

2.3.1 Game Loop Structure

The C runtime, popn_main.c, is structured as a top-level state machine with three states matching the hardware: show_menu, play_song, and show_results. The main function sets up the input device, memory-maps the FPGA register region, and then loops indefinitely between these states. show_menu and show_results both block on a single call to wait_for_button, which polls /dev/input/event0 for any of the nine Pop'n button keycodes (304–312).

play_song is the heart of the system. It runs a tight 100 Hz loop (10 ms sleep per iteration) that performs four jobs: spawn upcoming notes, advance their on-screen positions, process input events from the player, and detect missed notes that have fallen past the strike line. The game's reference clock, g_start_ms, is captured with gettimeofday at the moment play_song begins; all subsequent timing is computed against this anchor.

2.3.2 Chart Format and Loading

Charts are stored as plain-text .chart files, with one note per line in the format “timestamp_ms lane_index”. The load_chart function performs a single linear pass over the file, skipping comment and blank lines, and populates the global g_notes array. The chart for “Beauty and the Beat” was generated offline using a Python script (make_chart.py) that runs the WAV file through librosa for tempo detection, beat tracking, and onset detection. The script merges beats and detected onsets, inserts half-beat fillers, and assigns each event to a lane via a percentile-based binning of the spectral centroid, so high-frequency events tend to spawn in the rightmost lanes and bass events spawn on the left. A 2-second lead-in is added to every event time so the player has time to see notes before the first hit.

2.3.3 Falling-Note Math and Live-Note Tracking

Each frame, the game loop walks the chart array forward and adds any note whose hit time falls within NOTE_TRAVEL_MS (2200 ms) of the current game time to its lane's live-note queue (g_live[lane]). The Y position of the head note in each lane is then computed using a linear interpolation:

y = (JUDGE_Y - 16) - (dt * ((JUDGE_Y - 16) - SPAWN_Y)) / NOTE_TRAVEL_MS

where dt is the time-until-hit in milliseconds, JUDGE_Y is the Y coordinate of the strike line (440), SPAWN_Y is the top of the screen (0), and the -16 offset shifts the target so that the white horizontal stripe at the center of the note (rather than its top edge) lands exactly on the strike line at dt=0. This subtle 16-pixel correction was the difference between the player's perception of "off by a note" timing and pixel-perfect alignment.

A safety guard in the spawn loop prevents notes that are already past their MISS window from appearing mid-screen on the first iteration: such notes are counted as misses and skipped without ever being assigned a Y coordinate. Without this guard, an unlucky scheduling delay could place a note partway down the screen on its first frame, breaking the visual contract that all notes enter from the top.

2.3.4 Input Handling and Hit Detection

The Pop'n controller enumerates as a standard USB HID gamepad and exposes its nine buttons as evdev keycodes 304–312. The game opens /dev/input/event0 in non-blocking mode and performs an EVIOCGRAB ioctl to claim exclusive access, preventing the keystrokes from leaking into the Linux console. On each loop iteration, all pending events are drained: EV_KEY key-down events set the corresponding bit in current_glow (which is pushed to the lane_glow_mask register verbatim) and trigger hit detection, while key-up events clear the bit.

Hit detection compares the absolute time difference between the press and the head note in that lane against three windows: PERFECT_WINDOW_MS (60 ms) for a COOL judgment, GOOD_WINDOW_MS (120 ms) for a GOOD judgment, and MISS_WINDOW_MS (150 ms) past the hit time for an auto-MISS. The judgment code, point delta, combo update, and groove change are all encapsulated in apply_hit, which then pushes the updated stats to the FPGA via push_state_stats.

2.3.5 BCD Encoding and Register Updates

All numeric displays use BCD (binary-coded decimal) so the hardware text renderer can index its font table directly with each digit. The to_bcd helper converts an integer into a packed BCD word with a configurable digit count. push_state_stats packs the score (6 digits), combo (4 digits), miss count (4 digits), and the cool/good counts (4 digits each) into the appropriate registers in a single batched call, which keeps the on-screen HUD in lockstep with the game state.

2.3.6 Audio Synchronization

Audio playback is delegated to a separate process to avoid coupling the WAV streaming to the tight game-loop cadence. spawn_audio uses fork() to create a child process and execl() to replace it with /root/audio_play, a small standalone binary (developed earlier in the course) that DMA-streams a 48 kHz WAV file to the WM8731 codec via the Audio Avalon peripheral on the lightweight bridge. The parent process records the child PID in g_audio_pid; at the end of the song, kill_audio sends SIGTERM and waits for the child to exit. Because the audio process starts at the same instant g_start_ms is captured, audio and video timing remain synchronized to within a single OS scheduling slice (typically under 10 ms).

3. Project Contributions, Lessons Learned, and Advice

3.1 Project Contributions

This was an individual project carried out solely by the author. The full scope of work—hardware, software, build infrastructure, deployment, and documentation—was designed and implemented end to end. Specific contributions included:

- Designed and authored the popn_engine SystemVerilog module, including the VGA timing generator, register file, hardware bitmap font, plushie note sprites with eyes, glossy circular buttons, segmented Groove Gauge, and the priority-ordered combinational pixel mux that composites every frame.
- Integrated popn_engine into the Qsys system on the heavyweight HPS-to-FPGA AXI bridge and brought the bridges up under Linux, including diagnosing and resolving the U-Boot boot-time issues that prevented the FPGA from being programmed reliably.
- Wrote the popn_main.c game runtime, including the chart loader, the falling-note simulation with the perfect-timing offset, the Menu/Playing/Results state machine, BCD-encoded register pushes, and the input-event pipeline against /dev/input/event0.
- Implemented the audio subsystem integration, decoupling WAV streaming into a forked child process so video and audio remain synchronized through a single shared timing anchor.
- Authored the offline make_chart.py beatmap generator using librosa for tempo detection, onset extraction, and percentile-based lane assignment via the spectral centroid, and authored the chart for “Beauty and the Beat.”
- Developed hardware/software co-debugging tools (the alive, regdump, and checkstats userspace probes; the force_fpga reflash utility) and the SD-card deployment workflow used throughout development.
- Performed all play-testing and timing-window calibration, and prepared the final project report and presentation.

3.2 Lessons Learned

3.2.1 Quartus Module Caching

The single biggest source of wasted time on this project was Quartus's habit of caching Qsys submodule sources. When popn_engine.sv was edited at the project root, Quartus would happily compile, but the resulting bitstream would still contain the old engine. The reason is that Qsys copies submodule sources into soc_system/synthesis/submodules/ at the time the system is generated, and quartus_sh --flow compile reads from that copy, not from the original. We hit this bug repeatedly: the synthesis log would show zero errors, the RBF would deploy successfully, and the on-screen output would be visibly identical to the previous build.

The fix is to overwrite every cached copy before each compile. We adopted a small one-liner that runs before every quartus_sh invocation:

```
for f in $(find . -name "*popn_engine*.sv"); do
   cp popn_engine.sv "$f"
done
```

This guarantees every cached copy matches the working file. A second sanity check that became part of our workflow was a grep for a known new identifier in the synthesis copy:

```
grep "score_bcd" soc_system/synthesis/submodules/popn_engine.sv
```

If that returns no matches, the cache was not overwritten and there is no point in proceeding to compile.

3.2.2 RBF Format and the Frozen Bus

By default, Quartus emits an RBF in Passive Serial format. The Cyclone V HPS FPGA Manager, however, expects Fast Passive Parallel (FPP) format when programming the fabric from the HPS side. A PS-format RBF will load without error, but the resulting bitstream will not respond on the AXI bridges, and reads return a characteristic 0xDEADDEAD pattern from the configured address space. Worse, after the FPGA Manager streams a PS-format RBF and asserts CONF_DONE, subsequent attempts to write the bridge-control registers from userspace can hang the entire AXI fabric, requiring a hard power cycle to recover. Converting the bitstream after each compile fixes this, and is a one-line addition to the build:

```
quartus_cpf -c -m FPP output_files/soc_system.sof output_files/soc_system.rbf
```

We further discovered that the stock Terasic U-Boot script for the DE1-SoC contains a subtle self-corruption bug: it loads the script into memory at $fpgadata (0x02000000), and then the script itself issues a fatload of soc_system.rbf into the same address, overwriting the currently-executing script bytes. Some U-Boot builds tolerate this because they fully cache the command stream before executing, but the build shipped on our SD card does not. The cleanest workaround is to load the bitstream into a different address. We rewrote the boot script to use 0x10000000 instead of $fpgadata for the RBF, then re-wrapped it with a custom mkimage replacement and dropped it onto partition 1.

3.2.3 Pipelined Visual Rendering

Sprite-based rendering on a small FPGA tempts the designer toward block RAM for character glyphs and sprite atlases. We deliberately avoided this. The 3×5 bitmap font occupies 32 entries of 15 bits each, which infers cleanly as a constant lookup table and consumes essentially zero RAM blocks. The notes, buttons, and Groove Gauge are all derived from arithmetic predicates on the (dx, dy) offset from a center, so they too need no RAM. The result is a complete arcade-style rendering pipeline that fits comfortably in the Cyclone V's logic resources with substantial headroom for future additions.

The trade-off is that the always_comb pixel mux contains roughly forty cascaded else-if branches, and SystemVerilog strictness rules around variable initialization in procedural blocks bit us twice during synthesis. Specifically, declaring a logic variable and assigning it from a function call on the same line is treated as a static initialization (constant required) inside always_comb. The fix is to split the declaration and the assignment into two statements:

```
logic [1:0] px_type;
px_type = render_note(int'(hcount), int'(vcount), LANE_CX[i], note_y[i] + 16);
```

Quartus's error number for this is 10748. Once we adopted this pattern uniformly, we never hit the issue again. A second, related lesson: pixel-mux ordering matters. The first pass of the engine drew lane backgrounds before the bitmap text and the score never appeared. Re-ordering the priority chain so that text overlays sit ahead of background fills resolved it. The general rule, learned the hard way: in a flat priority mux, foreground always goes first.

3.2.4 SD Card Reader Reliability

Several hours of board-side debugging turned out to be caused by an SD-card reader on one of the lab workstations that silently corrupted FAT32 writes. md5 verification on the workstation passed, but bytes that landed on the partition did not match the source after the card was moved to a different reader. The takeaway: keep one known-good workstation as the deployment machine, and consider md5-verifying each file on the SD card from a second machine before assuming the issue is on the board.

3.3 Advice for Future Projects

3.3.1 Generate Dense, Continuous Beatmaps for Visual Stress Testing

A rhythm game looks broken when there are visible empty stretches between notes, even if the chart is musically correct. We generated charts via a Python script (make_chart.py) using librosa, then explicitly thickened the chart by inserting half-beat fillers and gap-filling notes during stretches longer than 500 ms. This not only produces a better-looking demo, it also stress-tests the renderer: many simultaneous notes per lane reveal off-by-one errors and pipeline ordering bugs that a sparse chart would never trigger. Aim for an on-screen note count of three to six at any given time and pair this with a NOTE_TRAVEL_MS of at least 2 seconds so the player can parse the falling stream visually.

3.3.2 Bypass U-Boot for Iteration

Once you have a working bitstream once, you can program subsequent bitstreams directly from Linux by writing to the FPGA Manager registers at 0xFF706000 via /dev/mem, bypassing U-Boot entirely. This is much faster than rebooting, and it sidesteps the U-Boot script issues described above. We built a small standalone tool, /root/force_fpga, that performs the FPGA Manager state machine handshake (nCONFIG reset, AXI streaming of the RBF, post-configuration bridge release) and used it for almost every rebuild after the initial board bring-up. With the FPP-format RBF on the SD card, the entire reflash cycle drops from "reboot, wait, log in, hope U-Boot ran the script" to a single command that takes under a second.

3.3.3 Test Verilog UI Incrementally

Pixel-mux logic is hard to debug because the only output is a 60 Hz video signal. Two practices helped enormously. First, render every new feature with an obviously wrong color (magenta, lime green) at first, so any presence on screen confirms the layer is active and the priority order is correct; only after that, refine the color. Second, write a small userspace probe (we built a checkstats.c that wrote known patterns to each register and printed the readback) so that hardware changes can be verified register by register before any new on-screen content is expected.

3.3.4 Keep One Source of Truth for the Register Map

The single most error-prone class of bugs we hit during integration was register-offset mismatches between the Verilog case statement and the C #define block. We recommend either generating both from a shared header (e.g., a Python script that emits both an SV include and a C header) or, at minimum, keeping them adjacent in a single comment block at the top of both files. The packed registers (0x38 and 0x3C) are particularly hazardous because the bit positions of the sub-fields must agree on both sides exactly.

4. Appendix: Source Code

Complete source code for the three files we authored for this project is included in the accompanying project.tar.gz archive. The three files are popn_main.c (the C-language game runtime that runs on the HPS under Linux), popn_engine.sv (the SystemVerilog hardware module that implements the VGA rendering pipeline and the Avalon-MM register interface on the FPGA fabric), and make_chart.py (the offline Python script that generates the .chart beatmap files from a source WAV by performing tempo detection, onset extraction, and percentile-based lane assignment via the librosa library).

Files generated automatically by Quartus, Qsys, or the Linux build system are not listed here, in accordance with the project specification. The archive also contains the Quartus project files (.qpf, .qsf, .sdc, .qsys, .sopcinfo) needed to rebuild the bitstream from a clean checkout, plus a README.md with the build and deploy procedure.

Appendix A: popn_main.c

```
/*
 * Pop'n Engine — main game loop
 * Updates vs previous:
 *   - NOTE_TRAVEL_MS increased to 2200ms so notes visible longer
 *   - Notes that are already past miss window on spawn are counted as miss but NOT displayed mid-screen
 *   - Smoother stream of notes
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <linux/input.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/ioctl.h>

#define HW_REGS_BASE        0xC0000000
#define HW_REGS_SPAN        0x04000000

#define GLOW_OFFSET         0x00
#define NOTE_Y_OFFSET       0x04
#define SCORE_OFFSET        0x28
#define COMBO_OFFSET        0x2C
#define JUDGMENT_OFFSET     0x30
#define GROOVE_OFFSET       0x34
#define STATE_OFFSET        0x38
#define STATS_OFFSET        0x3C

#define NUM_LANES           9
#define MAX_NOTES           4096
#define MAX_LIVE_PER_LANE   16

#define JUDGE_Y             440
#define SPAWN_Y             0
#define NOTE_TRAVEL_MS      2200
#define OFFSCREEN_Y         600

#define PERFECT_WINDOW_MS   60
#define GOOD_WINDOW_MS      120
#define MISS_WINDOW_MS      150

#define SCORE_PERFECT       1000
#define SCORE_GOOD          500

#define JUDGMENT_HOLD_MS    400

#define STATE_MENU          0
#define STATE_PLAYING       1
#define STATE_RESULTS       2

#define JUDGE_NONE          0
#define JUDGE_COOL          1
#define JUDGE_GOOD          2
#define JUDGE_MISS          3

#define GROOVE_COOL_DELTA   +4
#define GROOVE_GOOD_DELTA   +2
#define GROOVE_MISS_DELTA   -8
#define GROOVE_MIN          0
#define GROOVE_MAX          255
#define GROOVE_START        128
```

```
static const int LANE_KEYS[NUM_LANES] = { 304, 305, 306, 307, 308, 309, 310, 311, 312 };

typedef struct { int time_ms; int lane; int hit; } Note;
typedef struct { int idx; } LiveRef;

static Note g_notes[MAX_NOTES];
static int g_nnotes = 0, g_next_spawn = 0;
static LiveRef g_live[NUM_LANES][MAX_LIVE_PER_LANE];
static int g_nlive[NUM_LANES];

static int g_score = 0, g_combo = 0, g_max_combo = 0;
static int g_perfect = 0, g_good = 0, g_miss = 0;
static int g_groove = GROOVE_START;

static volatile sig_atomic_t g_running = 1;
static pid_t g_audio_pid = 0;

static volatile uint32_t *g_glow_reg;
static volatile uint32_t *g_note_y_reg[NUM_LANES];
static volatile uint32_t *g_score_reg;
static volatile uint32_t *g_combo_reg;
static volatile uint32_t *g_judgment_reg;
static volatile uint32_t *g_groove_reg;
static volatile uint32_t *g_state_reg;
static volatile uint32_t *g_stats_reg;

static long long g_start_ms = 0;
static long long g_judgment_clear_ms = 0;

static long long now_ms(void) {
    struct timeval tv; gettimeofday(&tv, NULL);
    return (long long)tv.tv_sec * 1000 + tv.tv_usec / 1000;
}
static int game_t_ms(void) { return (int)(now_ms() - g_start_ms); }

static uint32_t to_bcd(uint32_t n, int digits) {
    uint32_t out = 0;
    for (int i = 0; i < digits; i++) { out |= (n % 10) << (i * 4); n /= 10; }
    return out;
}
static void on_sigint(int s) { (void)s; g_running = 0; }

static int load_chart(const char *path) {
    FILE *f = fopen(path, "r");
    if (!f) return -1;
    char line[256];
    g_nnotes = 0;
    while (fgets(line, sizeof(line), f) && g_nnotes < MAX_NOTES) {
        char *p = line; while (*p == ' ' || *p == '\t') p++;
        if (*p == '#' || *p == '\n' || *p == '\0') continue;
        int t, lane;
        if (sscanf(p, "%d %d", &t, &lane) == 2 && lane >= 0 && lane < NUM_LANES) {
            g_notes[g_nnotes].time_ms = t;
            g_notes[g_nnotes].lane = lane;
            g_notes[g_nnotes].hit = 0;
            g_nnotes++;
        }
    }
    fclose(f); return 0;
}

static void spawn_audio(const char *wav) {
    if (!wav || strcmp(wav, "NONE") == 0 || strcmp(wav, "-") == 0) return;
    pid_t pid = fork();
    if (pid == 0) { execl("/root/audio_play", "audio_play", wav, (char*)NULL); _exit(127); }
    else if (pid > 0) g_audio_pid = pid;
}
static void kill_audio(void) {
    if (g_audio_pid > 0) { kill(g_audio_pid, SIGTERM); waitpid(g_audio_pid, NULL, 0); g_audio_pid = 0; }
}
```

```
static void add_live(int lane, int idx) {
    if (g_nlive[lane] < MAX_LIVE_PER_LANE) g_live[lane][g_nlive[lane]++].idx = idx;
}
static void remove_live(int lane, int slot) {
    for (int k = slot; k < g_nlive[lane] - 1; k++) g_live[lane][k] = g_live[lane][k + 1];
    g_nlive[lane]--;
}

static void push_state_stats(int state) {
    *g_score_reg = to_bcd((uint32_t)g_score, 6);
    *g_combo_reg = to_bcd((uint32_t)g_combo, 4);
    *g_groove_reg = (uint32_t)g_groove;
    *g_state_reg = (to_bcd((uint32_t)g_miss, 4) << 16) | (uint32_t)state;
    *g_stats_reg = (to_bcd((uint32_t)g_perfect, 4) << 16) | to_bcd((uint32_t)g_good, 4);
}
static void set_judgment(int code) {
    *g_judgment_reg = (uint32_t)code;
    if (code != JUDGE_NONE) g_judgment_clear_ms = now_ms() + JUDGMENT_HOLD_MS;
}
static void hide_all_notes(void) {
    for (int i = 0; i < NUM_LANES; i++) *g_note_y_reg[i] = OFFSCREEN_Y;
    *g_glow_reg = 0;
}
static int wait_for_button(int fd_kbd) {
    struct input_event ev;
    while (g_running) {
        if (read(fd_kbd, &ev, sizeof(ev)) == sizeof(ev) && ev.type == EV_KEY && ev.value == 1) {
            for (int lane = 0; lane < NUM_LANES; lane++) if (ev.code == LANE_KEYS[lane]) return 1;
        }
        usleep(10000);
    }
    return 0;
}

static void apply_hit(int judge_code, int lane, int dt, int t) {
    (void)lane; (void)dt; (void)t;
    switch (judge_code) {
        case JUDGE_COOL: g_perfect++; g_score += SCORE_PERFECT; g_combo++; g_groove += GROOVE_COOL_DELTA; break;
        case JUDGE_GOOD: g_good++; g_score += SCORE_GOOD; g_combo++; g_groove += GROOVE_GOOD_DELTA; break;
        case JUDGE_MISS: g_miss++; g_combo = 0; g_groove += GROOVE_MISS_DELTA; break;
    }
    if (g_combo > g_max_combo) g_max_combo = g_combo;
    if (g_groove < GROOVE_MIN) g_groove = GROOVE_MIN;
    if (g_groove > GROOVE_MAX) g_groove = GROOVE_MAX;
    set_judgment(judge_code);
    push_state_stats(STATE_PLAYING);
}

static void play_song(int fd_kbd) {
    g_next_spawn = 0;
    for (int i = 0; i < NUM_LANES; i++) g_nlive[i] = 0;
    for (int i = 0; i < g_nnotes; i++) g_notes[i].hit = 0;
    g_score = 0; g_combo = 0; g_max_combo = 0; g_perfect = 0; g_good = 0; g_miss = 0; g_groove = GROOVE_START;

    push_state_stats(STATE_PLAYING);
    set_judgment(JUDGE_NONE);
    hide_all_notes();

    g_start_ms = now_ms();
    int current_glow = 0;
    int song_ms = (g_nnotes > 0) ? g_notes[g_nnotes - 1].time_ms : 0;
    int finish_ms = song_ms + MISS_WINDOW_MS + 1000;

    while (g_running) {
        int t = game_t_ms();
        if (t > finish_ms) break;

        if (g_judgment_clear_ms > 0 && now_ms() >= g_judgment_clear_ms) {
            set_judgment(JUDGE_NONE); g_judgment_clear_ms = 0;
```

```
        }

        /* Spawn notes. If a note is already past its miss window, skip it entirely
         * so it never displays mid-screen. */
        while (g_next_spawn < g_nnotes && g_notes[g_next_spawn].time_ms - t <= NOTE_TRAVEL_MS) {
            int dt_spawn = g_notes[g_next_spawn].time_ms - t;
            if (dt_spawn < -MISS_WINDOW_MS) {
                g_notes[g_next_spawn].hit = 2;
                g_miss++;
                g_next_spawn++;
            } else {
                add_live(g_notes[g_next_spawn].lane, g_next_spawn);
                g_next_spawn++;
            }
        }

        for (int lane = 0; lane < NUM_LANES; lane++) {
            if (g_nlive[lane] == 0) { *g_note_y_reg[lane] = OFFSCREEN_Y; continue; }
            int ni = g_live[lane][0].idx;
            int dt = g_notes[ni].time_ms - t;

            int y = (JUDGE_Y - 16) - (dt * ((JUDGE_Y - 16) - SPAWN_Y)) / NOTE_TRAVEL_MS;

            if (y < 0) y = 0;
            if (y > OFFSCREEN_Y) y = OFFSCREEN_Y;
            *g_note_y_reg[lane] = (uint32_t)y;

            if (dt < -MISS_WINDOW_MS) {
                g_notes[ni].hit = 2; apply_hit(JUDGE_MISS, lane, dt, t); remove_live(lane, 0);
            }
        }

        struct input_event ev;
        while (read(fd_kbd, &ev, sizeof(ev)) == sizeof(ev)) {
            if (ev.type != EV_KEY) continue;
            if (ev.value == 0) {
                for (int lane = 0; lane < NUM_LANES; lane++) if (ev.code == LANE_KEYS[lane]) current_glow &= ~(1 << lane);
            } else if (ev.value == 1) {
                for (int lane = 0; lane < NUM_LANES; lane++) {
                    if (ev.code != LANE_KEYS[lane]) continue;
                    current_glow |= (1 << lane);
                    if (g_nlive[lane] == 0) break;
                    int ni = g_live[lane][0].idx;
                    int dt = g_notes[ni].time_ms - t;
                    int adt = dt < 0 ? -dt : dt;
                    if (adt <= PERFECT_WINDOW_MS) {
                        g_notes[ni].hit = 1; apply_hit(JUDGE_COOL, lane, dt, t); remove_live(lane, 0);
                    } else if (adt <= GOOD_WINDOW_MS) {
                        g_notes[ni].hit = 1; apply_hit(JUDGE_GOOD, lane, dt, t); remove_live(lane, 0);
                    }
                    break;
                }
            }
        }
        *g_glow_reg = (uint32_t)current_glow;
        usleep(10000);
    }
    set_judgment(JUDGE_NONE); hide_all_notes();
}

static void show_menu(int fd_kbd) {
    hide_all_notes();
    g_score = 0; g_combo = 0; g_perfect = 0; g_good = 0; g_miss = 0; g_groove = GROOVE_START;
    push_state_stats(STATE_MENU);
    set_judgment(JUDGE_NONE);
    wait_for_button(fd_kbd);
}

static void show_results(int fd_kbd) {
    g_combo = g_max_combo;
```

```
    push_state_stats(STATE_RESULTS);
    hide_all_notes();
    set_judgment(JUDGE_NONE);
    wait_for_button(fd_kbd);
}

int main(int argc, char **argv) {
    const char *dev_path   = (argc > 1) ? argv[1] : "/dev/input/event0";
    const char *chart_path = (argc > 2) ? argv[2] : "/media/boot/beauty_beat.chart";
    const char *wav_path   = (argc > 3) ? argv[3] : "/media/boot/beauty_beat_48k.wav";

    signal(SIGINT, on_sigint);
    if (load_chart(chart_path) < 0) return 1;

    int fd_kbd = open(dev_path, O_RDONLY | O_NONBLOCK);
    if (fd_kbd < 0) { perror("open input"); return 1; }
    if (ioctl(fd_kbd, EVIOCGRAB, 1) < 0) fprintf(stderr, "warn: EVIOCGRAB failed\n");

    int fd_mem = open("/dev/mem", O_RDWR | O_SYNC);
    void *vbase = mmap(NULL, HW_REGS_SPAN, PROT_READ | PROT_WRITE, MAP_SHARED, fd_mem, HW_REGS_BASE);

    g_glow_reg     = (volatile uint32_t*)((char*)vbase + GLOW_OFFSET);
    for (int i = 0; i < NUM_LANES; i++) g_note_y_reg[i] = (volatile uint32_t*)((char*)vbase + NOTE_Y_OFFSET + 4*i);
    g_score_reg    = (volatile uint32_t*)((char*)vbase + SCORE_OFFSET);
    g_combo_reg    = (volatile uint32_t*)((char*)vbase + COMBO_OFFSET);
    g_judgment_reg = (volatile uint32_t*)((char*)vbase + JUDGMENT_OFFSET);
    g_groove_reg   = (volatile uint32_t*)((char*)vbase + GROOVE_OFFSET);
    g_state_reg    = (volatile uint32_t*)((char*)vbase + STATE_OFFSET);
    g_stats_reg    = (volatile uint32_t*)((char*)vbase + STATS_OFFSET);

    while (g_running) {
        show_menu(fd_kbd);
        if (!g_running) break;
        spawn_audio(wav_path);
        play_song(fd_kbd);
        kill_audio();
        if (!g_running) break;
        show_results(fd_kbd);
    }

    ioctl(fd_kbd, EVIOCGRAB, 0);
    close(fd_kbd); munmap(vbase, HW_REGS_SPAN); close(fd_mem);
    return 0;
}
```

Appendix B: popn_engine.sv

```
/*
 * Pop'n FPGA Rhythm Engine - Arcade Polish Edition
 */
module popn_engine (
    input  logic        clk,
    input  logic        reset,
    input  logic [3:0]  avs_address,
    input  logic        avs_write,
    input  logic [31:0] avs_writedata,
    output logic [7:0]  vga_r, vga_g, vga_b,
    output logic        vga_clk, vga_hs, vga_vs, vga_blank_n, vga_sync_n
);

    logic clk_25M;
    always_ff @(posedge clk) begin
        if (reset) clk_25M <= 1'b0;
        else       clk_25M <= ~clk_25M;
    end

    logic [9:0] hcount, vcount;
    always_ff @(posedge clk_25M) begin
        if (reset) begin hcount <= 0; vcount <= 0; end
        else if (hcount == 10'd799) begin
            hcount <= 0;
            vcount <= (vcount == 10'd524) ? 10'd0 : vcount + 10'd1;
        end else hcount <= hcount + 10'd1;
    end

    assign vga_clk = clk_25M;
    assign vga_hs = ~(hcount >= 10'd656 && hcount < 10'd752);
    assign vga_vs = ~(vcount >= 10'd490 && vcount < 10'd492);
    assign vga_blank_n = (hcount < 10'd640 && vcount < 10'd480);
    assign vga_sync_n = 1'b0;

    logic [8:0]  lane_glow_mask;
    logic [9:0]  note_y [0:8];
    logic [23:0] score_bcd;
    logic [15:0] combo_bcd;
    logic [1:0]  judgment;
    logic [7:0]  groove;
    logic [1:0]  game_state;
    logic [15:0] cool_bcd, good_bcd, miss_bcd;

    always_ff @(posedge clk) begin
        if (reset) begin
            lane_glow_mask <= 9'b0;
            for (int i=0; i<9; i++) note_y[i] <= 10'd600;
            score_bcd <= 24'h0; combo_bcd <= 16'h0;
            judgment <= 2'd0; groove <= 8'd128; game_state <= 2'd0;
            cool_bcd <= 16'h0; good_bcd <= 16'h0; miss_bcd <= 16'h0;
        end else if (avs_write) begin
            case (avs_address)
                4'h0: lane_glow_mask <= avs_writedata[8:0];
                4'h1: note_y[0] <= avs_writedata[9:0]; 4'h2: note_y[1] <= avs_writedata[9:0];
                4'h3: note_y[2] <= avs_writedata[9:0]; 4'h4: note_y[3] <= avs_writedata[9:0];
                4'h5: note_y[4] <= avs_writedata[9:0]; 4'h6: note_y[5] <= avs_writedata[9:0];
                4'h7: note_y[6] <= avs_writedata[9:0]; 4'h8: note_y[7] <= avs_writedata[9:0];
                4'h9: note_y[8] <= avs_writedata[9:0];
                4'hA: score_bcd <= avs_writedata[23:0];
                4'hB: combo_bcd <= avs_writedata[15:0];
                4'hC: judgment  <= avs_writedata[1:0];
                4'hD: groove    <= avs_writedata[7:0];
                4'hE: begin game_state <= avs_writedata[1:0]; miss_bcd <= avs_writedata[31:16]; end
                4'hF: begin cool_bcd <= avs_writedata[31:16]; good_bcd <= avs_writedata[15:0]; end
            endcase
        end
    end
```

```
// Playfield centered: x=80..560 (480 wide), 9 lanes of 53 px
localparam int PLAYFIELD_X_START = 80;
localparam int PLAYFIELD_X_END   = 560;
localparam int LANE_CX [0:8] = '{106, 159, 212, 265, 318, 371, 424, 477, 530};

// Pop'n arcade colors
localparam logic [7:0] NOTE_R [0:8] = '{8'hFF, 8'hFF, 8'h00, 8'h22, 8'hFF, 8'h22, 8'h00, 8'hFF, 8'hFF};
localparam logic [7:0] NOTE_G [0:8] = '{8'hFF, 8'hE0, 8'hC8, 8'h66, 8'h22, 8'h66, 8'hC8, 8'hE0, 8'hFF};
localparam logic [7:0] NOTE_B [0:8] = '{8'hFF, 8'h22, 8'h44, 8'hFF, 8'h22, 8'hFF, 8'h44, 8'h22, 8'hFF};

localparam logic [7:0] BTN_DARK_R [0:8] = '{8'h66, 8'h77, 8'h00, 8'h11, 8'h77, 8'h11, 8'h00, 8'h77, 8'h66};
localparam logic [7:0] BTN_DARK_G [0:8] = '{8'h66, 8'h66, 8'h66, 8'h22, 8'h11, 8'h22, 8'h66, 8'h66, 8'h66};
localparam logic [7:0] BTN_DARK_B [0:8] = '{8'h66, 8'h11, 8'h22, 8'h77, 8'h11, 8'h77, 8'h22, 8'h11, 8'h66};

localparam logic [14:0] FONT [0:31] = '{
   15'b111_101_101_101_111, 15'b010_110_010_010_111, 15'b111_001_111_100_111, 15'b111_001_111_001_111,
   15'b101_101_111_001_001, 15'b111_100_111_001_111, 15'b111_100_111_101_111, 15'b111_001_001_001_001,
   15'b111_101_111_101_111, 15'b111_101_111_001_111, 15'b111_101_111_101_101, 15'b111_100_100_100_111,
   15'b110_101_101_101_110, 15'b111_100_111_100_111, 15'b111_100_101_101_111, 15'b111_010_010_010_111,
   15'b100_100_100_100_111, 15'b101_111_101_101_101, 15'b111_101_101_101_111, 15'b111_101_111_100_100,
   15'b111_101_110_101_101, 15'b111_100_111_001_111, 15'b111_010_010_010_010, 15'b010_010_010_000_010,
   15'b000_000_000_000_000, 15'b000_000_000_000_000, 15'b000_000_000_000_000, 15'b000_000_000_000_000,
   15'b000_000_000_000_000, 15'b000_000_000_000_000, 15'b000_000_000_000_000, 15'b000_000_000_000_000
};

function automatic logic render_char(input logic [4:0] char_code, input int x0, input int y0, input int scale, input int hc, input int vc);
   int rel_x, rel_y, px, py; logic [14:0] glyph;
   rel_x = hc - x0; rel_y = vc - y0;
   if (rel_x < 0 || rel_x >= 3 * scale || rel_y < 0 || rel_y >= 5 * scale) return 1'b0;
   px = rel_x / scale; py = rel_y / scale;
   glyph = FONT[char_code]; return glyph[(4 - py) * 3 + (2 - px)];
endfunction

function automatic logic render_3digit(input logic [15:0] bcd, input int x0, input int y0, input int scale, input int hc, input int vc);
   int rel_x, rel_y, digit_idx, px, py; logic [3:0] dv; logic [14:0] glyph;
   rel_x = hc - x0; rel_y = vc - y0;
   if (rel_x < 0 || rel_x >= (scale*4)*3 || rel_y < 0 || rel_y >= 5*scale) return 1'b0;
   digit_idx = rel_x / (scale * 4); px = (rel_x % (scale * 4)) / scale; py = rel_y / scale;
   if (px >= 3) return 1'b0;
   case (digit_idx) 0: dv=bcd[11:8]; 1: dv=bcd[7:4]; 2: dv=bcd[3:0]; default: dv=4'h0; endcase
   glyph = FONT[dv]; return glyph[(4 - py) * 3 + (2 - px)];
endfunction

function automatic logic render_4digit(input logic [15:0] bcd, input int x0, input int y0, input int scale, input int hc, input int vc);
   int rel_x, rel_y, digit_idx, px, py; logic [3:0] dv; logic [14:0] glyph;
   rel_x = hc - x0; rel_y = vc - y0;
   if (rel_x < 0 || rel_x >= (scale*4)*4 || rel_y < 0 || rel_y >= 5*scale) return 1'b0;
   digit_idx = rel_x / (scale * 4); px = (rel_x % (scale * 4)) / scale; py = rel_y / scale;
   if (px >= 3) return 1'b0;
   case (digit_idx) 0: dv=bcd[15:12]; 1: dv=bcd[11:8]; 2: dv=bcd[7:4]; 3: dv=bcd[3:0]; default: dv=4'h0; endcase
   glyph = FONT[dv]; return glyph[(4 - py) * 3 + (2 - px)];
endfunction

function automatic logic render_6digit(input logic [23:0] bcd, input int x0, input int y0, input int scale, input int hc, input int vc);
   int rel_x, rel_y, digit_idx, px, py; logic [3:0] dv; logic [14:0] glyph;
   rel_x = hc - x0; rel_y = vc - y0;
   if (rel_x < 0 || rel_x >= (scale*4)*6 || rel_y < 0 || rel_y >= 5*scale) return 1'b0;
   digit_idx = rel_x / (scale * 4); px = (rel_x % (scale * 4)) / scale; py = rel_y / scale;
   if (px >= 3) return 1'b0;
   case (digit_idx)
      0: dv=bcd[23:20]; 1: dv=bcd[19:16]; 2: dv=bcd[15:12];
      3: dv=bcd[11:8];  4: dv=bcd[7:4];   5: dv=bcd[3:0];
      default: dv=4'h0;
   endcase
   glyph = FONT[dv]; return glyph[(4 - py) * 3 + (2 - px)];
endfunction

function automatic logic [1:0] render_note(input int hc, input int vc, input int cx, input int cy);
   int dx = hc - cx, dy = vc - cy;
```

```
    int adx = dx < 0 ? -dx : dx, ady = dy < 0 ? -dy : dy;
    int lex = dx - (-8), ley = dy - (-6);
    int alex = lex < 0 ? -lex : lex, aley = ley < 0 ? -ley : ley;
    int rex = dx - 8, rey = dy - (-6);
    int arex = rex < 0 ? -rex : rex, arey = rey < 0 ? -rey : rey;
    if ((adx <= 20) && (ady <= 15) && (adx + ady <= 30)) begin
        if ((alex <= 1 && aley <= 2) || (arex <= 1 && arey <= 2)) return 2'd3;
        if ((alex <= 3 && aley <= 3 && (alex+aley <= 5)) || (arex <= 3 && arey <= 3 && (arex+arey <= 5))) return 2'd2;
        if (dy >= -1 && dy <= 2) return 2'd2;
        return 2'd1;
    end
    return 2'd0;
endfunction

function automatic logic [1:0] render_button(input int hc, input int vc, input int cx, input int cy);
    int dx = hc - cx, dy = vc - cy;
    int d2 = dx*dx + dy*dy;
    if (d2 <= 22*22 && d2 > 17*17) return 2'd3;
    if (d2 <= 17*17) begin
        if ((dx+8)*(dx+8) + (dy+5)*(dy+5) <= 5*5) return 2'd2;
        return 2'd1;
    end
    return 2'd0;
endfunction

function automatic logic [1:0] render_groove_px(input logic [7:0] level, input int hc, input int vc);
    int bar_width = (level * 400) / 255;
    int rel_x = hc - 120, rel_y = vc - 455;
    if (rel_x >= 0 && rel_x < 400 && rel_y >= 0 && rel_y <= 8) begin
        if (rel_x < bar_width) begin
            if (rel_x % 10 < 8) return 2'd1;
        end else begin
            if (rel_x % 10 < 8) return 2'd2;
        end
    end
    return 2'd0;
endfunction

logic is_stat_txt, is_judge_txt, is_menu_txt, is_res_txt, is_title_txt;
logic [2:0] judge_color;
logic [1:0] active_note_px, active_groove_px, active_btn_px;
logic [3:0] active_note_lane, active_btn_lane;
logic in_playfield, in_lane_zone;
logic [3:0] current_lane;

always_comb begin
    is_stat_txt = 1'b0; is_judge_txt = 1'b0; judge_color = 3'd0;
    is_menu_txt = 1'b0; is_res_txt = 1'b0; is_title_txt = 1'b0;

    if (game_state == 2'd0) begin
        is_menu_txt = render_char(19, 260, 200, 4, int'(hcount), int'(vcount)) | render_char(20, 276, 200, 4, int'(hcount), int'(vcount)) |
                      render_char(13, 292, 200, 4, int'(hcount), int'(vcount)) | render_char(21, 308, 200, 4, int'(hcount), int'(vcount)) |
                      render_char(21, 324, 200, 4, int'(hcount), int'(vcount)) | render_char(21, 260, 240, 4, int'(hcount), int'(vcount)) |
                      render_char(22, 276, 240, 4, int'(hcount), int'(vcount)) | render_char(10, 292, 240, 4, int'(hcount), int'(vcount)) |
                      render_char(20, 308, 240, 4, int'(hcount), int'(vcount)) | render_char(22, 324, 240, 4, int'(hcount), int'(vcount));
        is_title_txt = render_char(19, 280, 80, 6, int'(hcount), int'(vcount)) | render_char(18, 304, 80, 6, int'(hcount), int'(vcount)) |
                       render_char(19, 328, 80, 6, int'(hcount), int'(vcount)) | render_char(17, 352, 80, 6, int'(hcount), int'(vcount));
    end else if (game_state == 2'd2) begin
        is_res_txt = render_char(21, 260, 160, 4, int'(hcount), int'(vcount)) | render_char(11, 276, 160, 4, int'(hcount), int'(vcount)) |
                     render_char(18, 292, 160, 4, int'(hcount), int'(vcount)) | render_char(20, 308, 160, 4, int'(hcount), int'(vcount)) |
                     render_char(13, 324, 160, 4, int'(hcount), int'(vcount));
    end

    if (game_state == 2'd1 || game_state == 2'd2) begin
        is_stat_txt = render_char(11,10,380,2,int'(hcount),int'(vcount)) | render_char(18,18,380,2,int'(hcount),int'(vcount)) |
                      render_char(18,26,380,2,int'(hcount),int'(vcount)) | render_char(16,34,380,2,int'(hcount),int'(vcount)) |
                      render_char(14,10,400,2,int'(hcount),int'(vcount)) | render_char(18,18,400,2,int'(hcount),int'(vcount)) |
                      render_char(18,26,400,2,int'(hcount),int'(vcount)) | render_char(12,34,400,2,int'(hcount),int'(vcount)) |
                      render_char(17,10,420,2,int'(hcount),int'(vcount)) | render_char(15,18,420,2,int'(hcount),int'(vcount)) |
                      render_char(21,26,420,2,int'(hcount),int'(vcount)) | render_char(21,34,420,2,int'(hcount),int'(vcount));
```

```
    end

    if (game_state == 2'd1 && judgment != 2'd0) begin
      if (judgment == 2'd1) begin
        is_judge_txt = render_char(11,260,180,6,int'(hcount),int'(vcount)) | render_char(18,284,180,6,int'(hcount),int'(vcount)) |
                       render_char(18,308,180,6,int'(hcount),int'(vcount)) | render_char(16,332,180,6,int'(hcount),int'(vcount)) |
                       render_char(23,356,180,6,int'(hcount),int'(vcount)); judge_color = 3'd1;
      end else if (judgment == 2'd2) begin
        is_judge_txt = render_char(14,260,180,6,int'(hcount),int'(vcount)) | render_char(18,284,180,6,int'(hcount),int'(vcount)) |
                       render_char(18,308,180,6,int'(hcount),int'(vcount)) | render_char(12,332,180,6,int'(hcount),int'(vcount)) |
                       render_char(23,356,180,6,int'(hcount),int'(vcount)); judge_color = 3'd2;
      end else if (judgment == 2'd3) begin
        is_judge_txt = render_char(17,260,180,6,int'(hcount),int'(vcount)) | render_char(15,284,180,6,int'(hcount),int'(vcount)) |
                       render_char(21,308,180,6,int'(hcount),int'(vcount)) | render_char(21,332,180,6,int'(hcount),int'(vcount)) |
                       render_char(23,356,180,6,int'(hcount),int'(vcount)); judge_color = 3'd3;
      end
    end

    in_playfield = (hcount >= 10'(PLAYFIELD_X_START)) && (hcount < 10'(PLAYFIELD_X_END));
    in_lane_zone = 1'b0;
    current_lane = 0;
    if (in_playfield) begin
      for (int i = 0; i < 9; i++) begin
        if (int'(hcount) >= (80 + 53*i + 2) && int'(hcount) < (80 + 53*(i+1) - 2)) begin
          in_lane_zone = 1'b1;
          current_lane = 4'(i);
        end
      end
    end

    active_note_px = 2'd0; active_note_lane = 0;
    if (game_state != 2'd0) begin
      for (int i=0; i<9; i++) begin
        if (note_y[i] < 10'd480) begin
          logic [1:0] px_type;
          px_type = render_note(int'(hcount), int'(vcount), LANE_CX[i], int'(note_y[i]) + 15);
          if (px_type != 2'd0) begin active_note_px = px_type; active_note_lane = 4'(i); end
        end
      end
    end

    active_btn_px = 2'd0; active_btn_lane = 0;
    if (game_state != 2'd0) begin
      for (int i=0; i<9; i++) begin
        logic [1:0] btn_px;
        btn_px = render_button(int'(hcount), int'(vcount), LANE_CX[i], 465);
        if (btn_px != 2'd0) begin active_btn_px = btn_px; active_btn_lane = 4'(i); end
      end
    end

    active_groove_px = 2'd0;
    if (game_state == 2'd1) active_groove_px = render_groove_px(groove, int'(hcount), int'(vcount));

    if (!vga_blank_n) begin
      vga_r = 8'h00; vga_g = 8'h00; vga_b = 8'h00;
    end
    else if (is_menu_txt) begin vga_r=8'hFF; vga_g=8'hFF; vga_b=8'hFF; end
    else if (is_title_txt) begin vga_r=8'hFF; vga_g=8'hDD; vga_b=8'h00; end
    else if (is_res_txt) begin vga_r=8'h00; vga_g=8'hFF; vga_b=8'hFF; end
    else if (game_state == 2'd2 && render_6digit(score_bcd, 236, 220, 6, int'(hcount), int'(vcount))) begin
      vga_r=8'hFF; vga_g=8'hFF; vga_b=8'h00;
    end
    else if (is_judge_txt) begin
      if (judge_color == 3'd1) begin vga_r=8'hFF; vga_g=8'hFF; vga_b=8'h00; end
      else if (judge_color == 3'd2) begin vga_r=8'h00; vga_g=8'hFF; vga_b=8'h00; end
      else begin vga_r=8'hFF; vga_g=8'h00; vga_b=8'h00; end
    end
    else if (is_stat_txt) begin vga_r=8'hFF; vga_g=8'hFF; vga_b=8'hFF; end
    else if ((game_state == 2'd1 || game_state == 2'd2) && render_3digit(cool_bcd, 46, 380, 2, int'(hcount), int'(vcount))) begin vga_r=8'hFF;
vga_g=8'hFF; vga_b=8'h00; end
```

```
      else if ((game_state == 2'd1 || game_state == 2'd2) && render_3digit(good_bcd, 46, 400, 2, int'(hcount), int'(vcount))) begin vga_r=8'h00;
vga_g=8'hFF; vga_b=8'h00; end
      else if ((game_state == 2'd1 || game_state == 2'd2) && render_3digit(miss_bcd, 46, 420, 2, int'(hcount), int'(vcount))) begin vga_r=8'hFF;
vga_g=8'h00; vga_b=8'h00; end
      else if (game_state != 2'd2 && render_6digit(score_bcd, 5, 8, 2, int'(hcount), int'(vcount))) begin vga_r=8'hFF; vga_g=8'hFF; vga_b=8'h00;
end
      else if (game_state != 2'd2 && render_4digit(combo_bcd, 585, 8, 2, int'(hcount), int'(vcount))) begin vga_r=8'h00; vga_g=8'hFF;
vga_b=8'hFF; end
      else if (active_groove_px == 2'd1) begin vga_r=8'h00; vga_g=8'hFF; vga_b=8'hFF; end
      else if (active_groove_px == 2'd2) begin vga_r=8'h22; vga_g=8'h22; vga_b=8'h44; end
      else if (active_note_px != 2'd0) begin
        if (active_note_px == 2'd2) begin vga_r=8'hFF; vga_g=8'hFF; vga_b=8'hFF; end
        else if (active_note_px == 2'd3) begin vga_r=8'h11; vga_g=8'h11; vga_b=8'h11; end
        else begin vga_r=NOTE_R[active_note_lane]; vga_g=NOTE_G[active_note_lane]; vga_b=NOTE_B[active_note_lane]; end
      end
      else if (game_state != 2'd0 && (vcount == 10'd438 || vcount == 10'd439 || vcount == 10'd440 || vcount == 10'd441) && in_playfield) begin
        vga_r=8'hFF; vga_g=8'hFF; vga_b=8'hFF;
      end
      else if (active_btn_px != 2'd0) begin
        if (active_btn_px == 2'd2) begin vga_r=8'hFF; vga_g=8'hFF; vga_b=8'hFF; end
        else if (active_btn_px == 2'd3) begin vga_r=8'h33; vga_g=8'h33; vga_b=8'h33; end
        else begin
          if (lane_glow_mask[active_btn_lane]) begin
            vga_r = NOTE_R[active_btn_lane]; vga_g = NOTE_G[active_btn_lane]; vga_b = NOTE_B[active_btn_lane];
          end else begin
            vga_r = BTN_DARK_R[active_btn_lane]; vga_g = BTN_DARK_G[active_btn_lane]; vga_b = BTN_DARK_B[active_btn_lane];
          end
        end
      end
      else if (in_playfield && vcount < 450) begin
        if (!in_lane_zone) begin
          vga_r = 8'h22; vga_g = 8'h22; vga_b = 8'h44;
        end else begin
          if (vcount[1:0] == 2'b00) begin
            vga_r = 8'h18; vga_g = 8'h18; vga_b = 8'h24;
          end else begin
            vga_r = 8'h0C; vga_g = 8'h0C; vga_b = 8'h18;
          end
        end
      end
      else begin
        vga_r = 8'h08;
        vga_g = 8'h08;
        vga_b = 8'h18;
        if (game_state == 2'd0 || game_state == 2'd2) begin
          vga_r = vga_r >> 1; vga_g = vga_g >> 1; vga_b = vga_b >> 1;
        end
      end
    end
endmodule
```

Appendix C: make_chart.py

```
#!/usr/bin/env python3
"""
Generate a denser chart for beauty_beat_30s.wav
Produces a chart with more notes so the playfield always has a continuous stream.
Usage: python3 make_chart.py <path_to_beauty_beat_30s.wav> <output_chart_path>
"""
import sys
import numpy as np
import librosa
from collections import Counter

if len(sys.argv) < 3:
    print("Usage: python3 make_chart.py <wav_path> <chart_out_path>")
    sys.exit(1)

wav_path = sys.argv[1]
out_path = sys.argv[2]

print(f"Loading {wav_path}...")
y, sr = librosa.load(wav_path, sr=22050, mono=True)
duration = len(y) / sr

tempo, beat_times = librosa.beat.beat_track(y=y, sr=sr, units='time')
tempo = float(tempo) if np.isscalar(tempo) else float(tempo.item() if tempo.size == 1 else tempo[0])
onset_times = librosa.onset.onset_detect(y=y, sr=sr, units='time', backtrack=True)

# Half-beat subdivisions for density
half_beats = []
for i in range(len(beat_times) - 1):
    mid = (beat_times[i] + beat_times[i+1]) / 2
    half_beats.append(mid)

# Merge beats + half-beats + onsets
all_events = sorted(list(beat_times) + half_beats + list(onset_times))
merged = []
last = -1
for t in all_events:
    if t - last > 0.08:
        merged.append(t)
        last = t

print(f"Events after merge: {len(merged)}")

# Spectral centroid per event for lane assignment
stft = librosa.stft(y, n_fft=2048, hop_length=512)
mag = np.abs(stft)
freqs = librosa.fft_frequencies(sr=sr, n_fft=2048)
hop_time = 512 / sr

def centroid_at(t):
    idx = min(int(t / hop_time), mag.shape[1]-1)
    s = mag[:, idx]
    if s.sum() < 1e-8:
        return 1000.0
    return float((s * freqs).sum() / s.sum())

centroids = [centroid_at(t) for t in merged]
sorted_cent = np.sort(centroids)
boundaries = [sorted_cent[int(len(sorted_cent) * p / 9)] for p in range(1, 9)]

def lane_for(c):
    for i, b in enumerate(boundaries):
        if c < b:
            return i
    return 8

chart = []
```

```
last_lane = -1
last_time = -1
for t, c in zip(merged, centroids):
    lane = lane_for(c)
    if lane == last_lane and (t - last_time) < 0.3:
        lane = (lane + 3) % 9
    last_lane = lane
    last_time = t
    ms = int(round(t * 1000))
    chart.append((ms, lane))

# Insert filler notes for large gaps
dense = []
for i, (ms, lane) in enumerate(chart):
    dense.append((ms, lane))
    if i + 1 < len(chart):
        next_ms, next_lane = chart[i+1]
        gap = next_ms - ms
        if gap > 900:
            dense.append((ms + gap // 3, (lane + 2) % 9))
            dense.append((ms + 2 * gap // 3, (lane + 5) % 9))
        elif gap > 500:
            dense.append((ms + gap // 2, (lane + 4) % 9))

dense.sort(key=lambda x: x[0])

# Dedup
final_pre = []
last_ms = -1000
for ms, lane in dense:
    if ms - last_ms >= 80:
        final_pre.append((ms, lane))
        last_ms = ms

LEAD_IN_MS = 2000
final = [(ms + LEAD_IN_MS, lane) for ms, lane in final_pre]

with open(out_path, "w") as f:
    f.write(f"# Pop'n chart, {len(final)} notes, {tempo:.1f} BPM\n")
    f.write("# Format: timestamp_ms lane_index\n")
    for ms, lane in final:
        f.write(f"{ms} {lane}\n")

print(f"Wrote {out_path}: {len(final)} notes")
print(f"First 10 notes:")
for ms, lane in final[:10]:
    print(f"  {ms}ms lane {lane}")
counts = Counter(l for _, l in final)
print(f"Lane usage: {dict(sorted(counts.items()))}")
```