Boots N' Cats Drum Machine
Design Document

Noel Gomez (ng2703), Jordan Lin (kl3758), Aaron Zhu (azz2111)
Embedded Systems (CSEE 4840)
Professor Stephen A. Edwards
Spring 2026

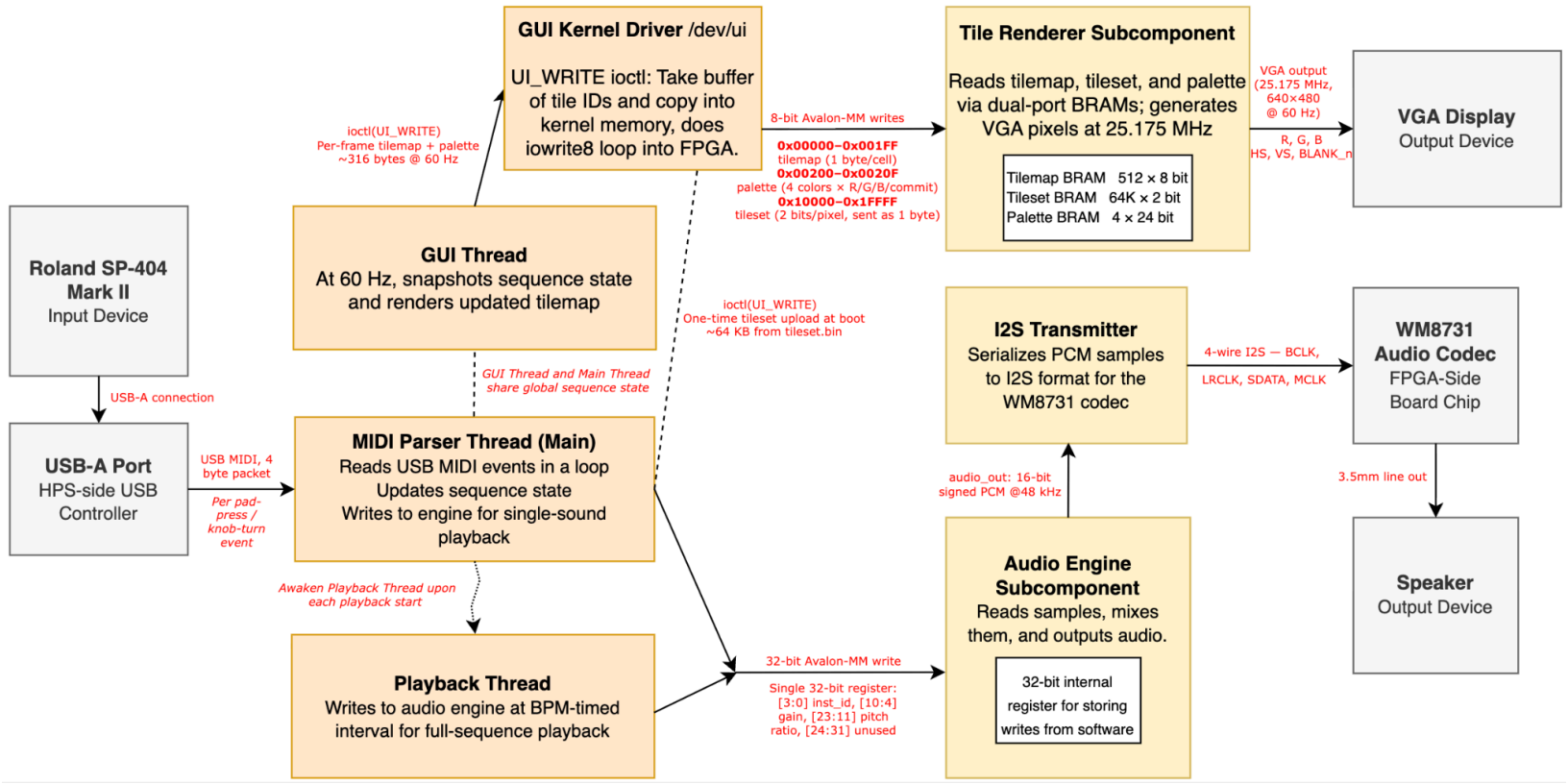
Contents

1. Introduction

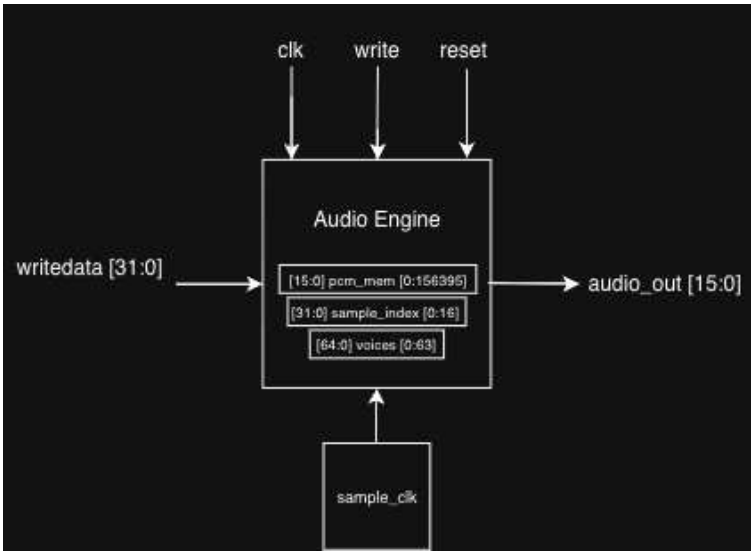
The SP-404 MKII's pitch knob (CTRL 2) outputs a 7-bit MIDI value in the range [0, 127]. We linearly map this range down to a semitone offset in [-24, 24] using the following formula:

The goal of this project is to implement a software/hardware drum machine inspired by the Roland TR-909 on an FPGA, with a custom user interface that feels like a modern DAW. The system will allow a user to program rhythmic patterns using a visual beatgrid and step sequencer interface, select drum samples from a pre-processed 909 audio sample bank, choose certain parameters for each sample such as pitch and tempo, and play the resulting beat in real time out of a speaker. We will use a 128 step beat grid matched to 8 measures that loops the user-created beat to output audio in playback mode. The system combines hardware sequencing logic implemented in Verilog on the FPGA with software-based UI logic written in C to manage the on-screen beat grid and keyboard controls, as well as keep everything in time with the tempo chosen by the user.

2. System Block Diagrams



High-Level System-Wide Block Diagram



Audio Engine Component Block Diagram

A detailed block diagram of the GUI Display can be found at the end of Section 5.

3. User Interface

3.1 Input UI Device (Roland SP-404 MKII)

The Roland SP-404 MKII serves as the primary hardware controller for our drum machine, providing a tactile interface for both creating a beat sequence and playing it back. Users interact with the SP-404 through sixteen velocity-sensitive pads, three primary control knobs, and various function buttons to manipulate their beat sequence in real-time. This physical layout allows for quick navigation between different steps, tracks, and measures, as well as immediate adjustment of sound parameters like pitch and volume. By leveraging the MIDI capabilities of the device, we map these hardware controls to software state changes within our embedded system. The interface is designed to be intuitive, using button backlighting to communicate current operational modes to the user. Together, this setup provides a comprehensive set of inputs for creating and playing complex beats.

The user interacts with pads, buttons, and knobs in four functional groups of controls on the Roland SP-404 MKII. These groups are color-coordinated in the reference diagram on the next page to visually distinguish their functions. Here is an overview of their functionality.

Yellow Buttons

CTRL 1 is for global BPM. CTRL 2 is for per-step-and-track Pitch. CTRL 3 is for per-step-and-track Volume.

Red Pads and Green Buttons

The two green buttons function as mode selectors for the sixteen number pads. The device operates in a single mode at any given time, determined by whether the user has toggled the A/F or B/G button. Consequently, the semantics of the numbered pads change based on the active mode selection.

The user can always be in one of two modes, edit or playback mode. Within edit mode there are two sub-modes, navigation and instrument select mode. The two green group buttons — A/F or B/G — correspond to the navigation and instrument select mode, respectively. These modes are mutually exclusive; the user can only be in one mode at a time.

Roland SP-404 MK II (used controls grouped by color)

A/F: Navigation Mode

- Number pads used: 1-9, 11, 13-16 (pressing 10 or 12 cause no-ops)
    - 1-8: select measure 1-8
    - 9: toggle cell active/inactive
    - 11: move up one track
    - 15: move down one track
    - 14: move left one step (wraps across measures as ring buffer)

    - 16: move right one step (wraps across measures as ring buffer)
    - 13: clear instrument and sequence from current track
    - 12: save the current sequence to a file in a directory called “saves” in the current working directory
- Knobs used: CTRL 2 (Pitch), CTRL 3 (Volume)
- When in mode A/F, a user presses pads 1-8 to change the current measure, and pads 11 (up), 15 (down), 14 (left), and 16 (right) to change the currently selected cell. After selecting a cell, a user can flip the state of the cell (i.e. inactive or active) by pressing pad 9. If the cell was previously inactive, pressing 9 turns it active. Additionally, the user presses pad 13 to clear all programmed cells on the track and remove the current instrument assignment. The currently selected cell will be greyed out in the grid UI, indicating to the user where they are in the grid.
- Pad 12 saves the current sequence to a file in a directory called “saves” in the current working directory, in JSON format (files of this format can be loaded as an argument when running the `drum` process).
- Additionally, in this mode, adjusting the pitch (knob 2) or volume (knob 3) changes the pitch of the current cell (only if the cell is already active).

B/G: Instrument Select Mode

- Number pads used: 1-16
    - Number pad X corresponds to instrument ID X - 1
- When in mode B/G, a user presses the pads to select which instrument it wants to have on the current track. All 8 tracks start out unassigned to instruments at the start of a new session. After moving to a particular track in navigation mode, the user switches to instrument select mode to set the instrument on that track, by selecting one of the 16 pads, corresponding to the 16 instruments. Selecting an instrument assigns it to the current track and clears the existing sequence (on all 128 steps, not just the 16 steps of the current measure).

Note that if either mode C/H, D/I, or E/J (the other channel buttons) is selected, all number pad presses will be no-ops.

Note that for all the channel buttons (i.e. A/F through E/J), the SP-404 does offer a distinction between the states X and Y for a channel X/Y. State X is indicated by a solid, non-blinking light from the button, whilst Y is indicated by a blinking light from the button. However, we do not distinguish between these states.

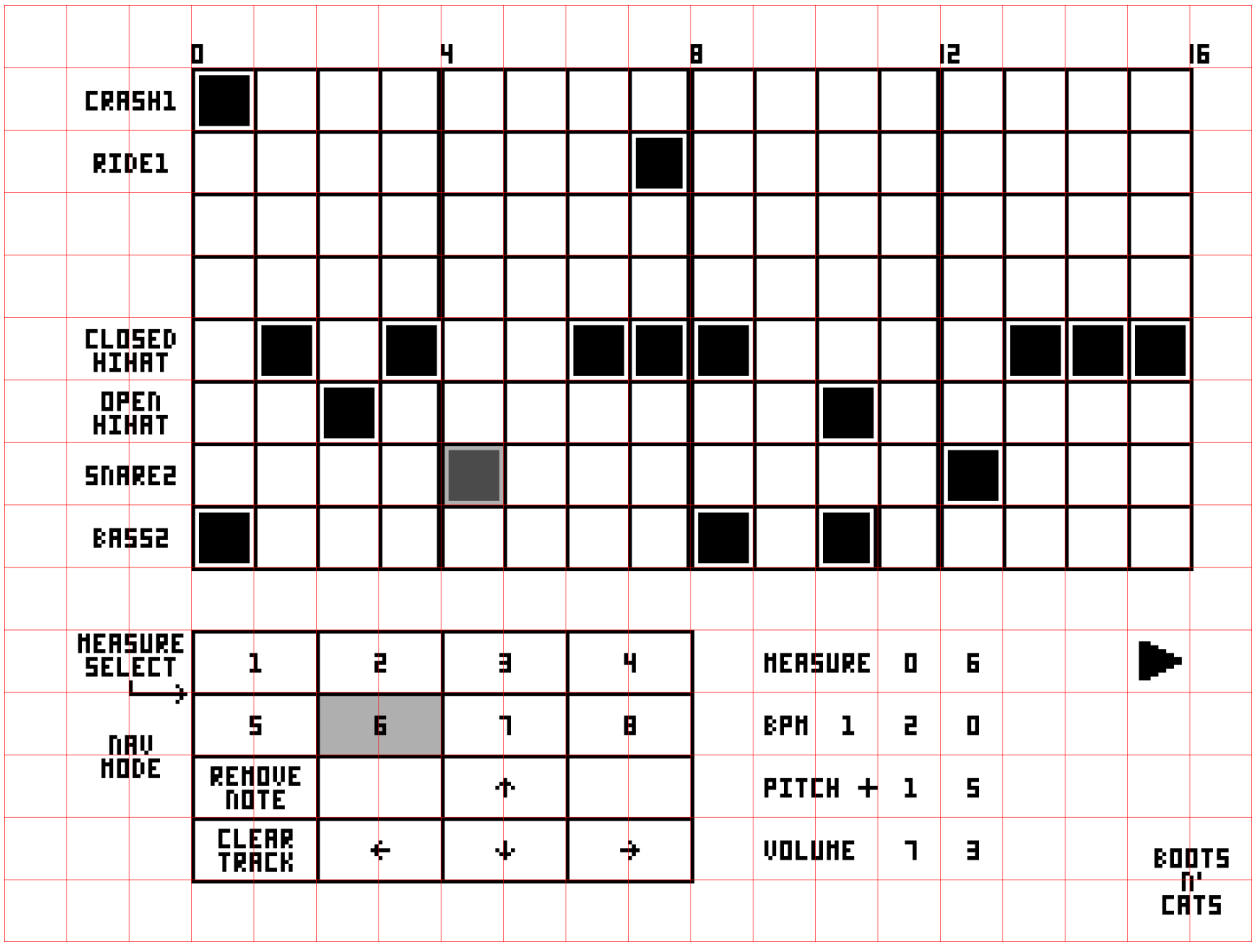
Purple Button

The button EXT SOURCE is for toggling between playback and edit mode.

3.2 Output UI (Tilemap on VGA Display)

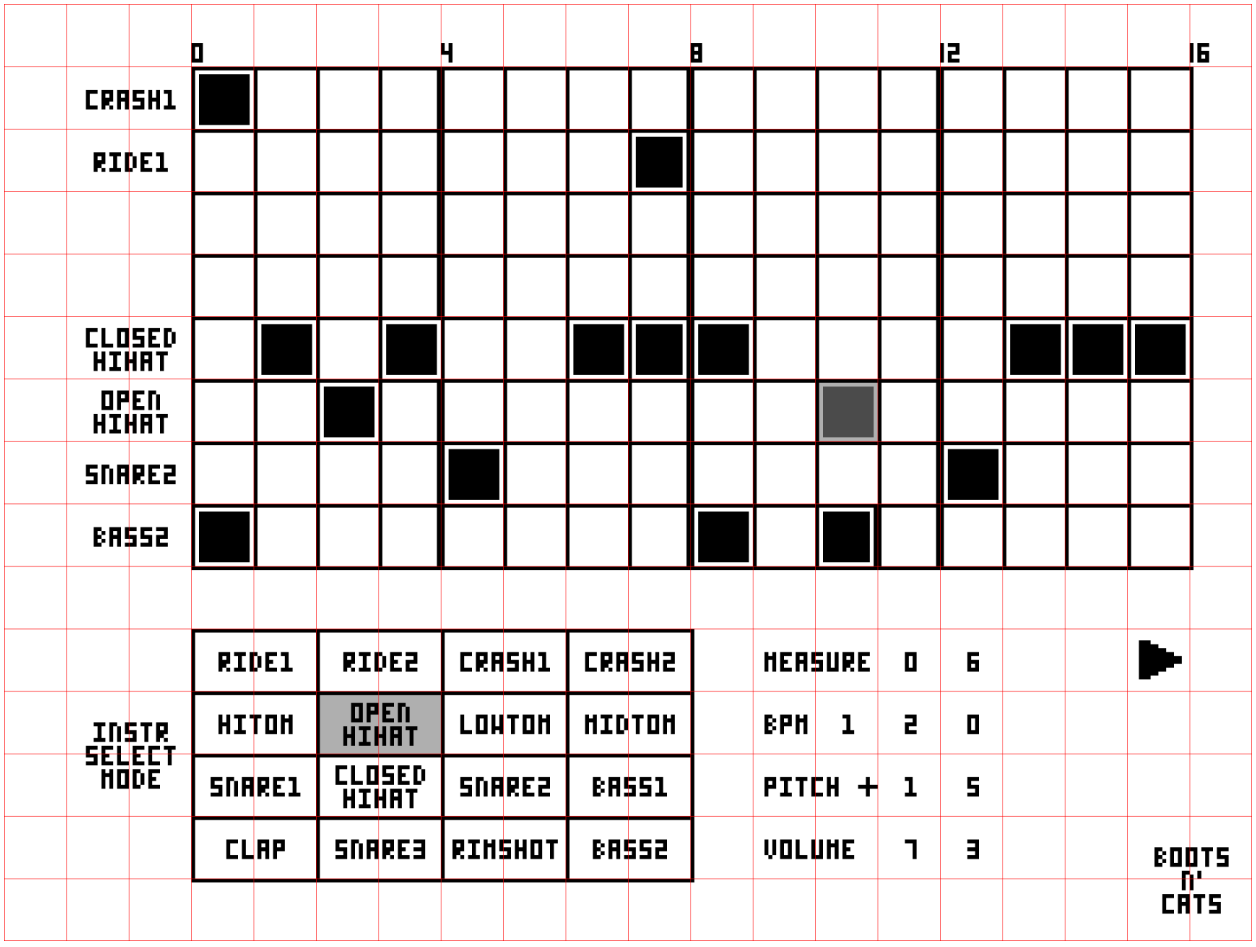
The output GUI is a tilemap on a VGA display that shows the 16-sample bank and the beat grid. The beat grid lets us program the beat sequence, determining which notes are active (filled in), along with their pitch, volume, and instrument (for the latter, on a per-track/horizontal basis), and we can also follow the playback cursor in real-time.

The on-screen UI reflects the two primary modes that are controlled by the Roland SP-404 MKII: Navigation mode and instrument selection mode. Namely, the 4×4 beneath the note grid maps to the 4×4 **Red** pads on the Roland.



*Example UI (in navigation mode) with red tile lines overlaid on top for visualization only. The current measure in measure select is highlighted, and pitch and volume of the current selected note (if there is a note) is shown. The 4×4 grid of measure numbers, arrow keys, and add/remove note and clear track mirrors the **Red** pads on the Roland.*

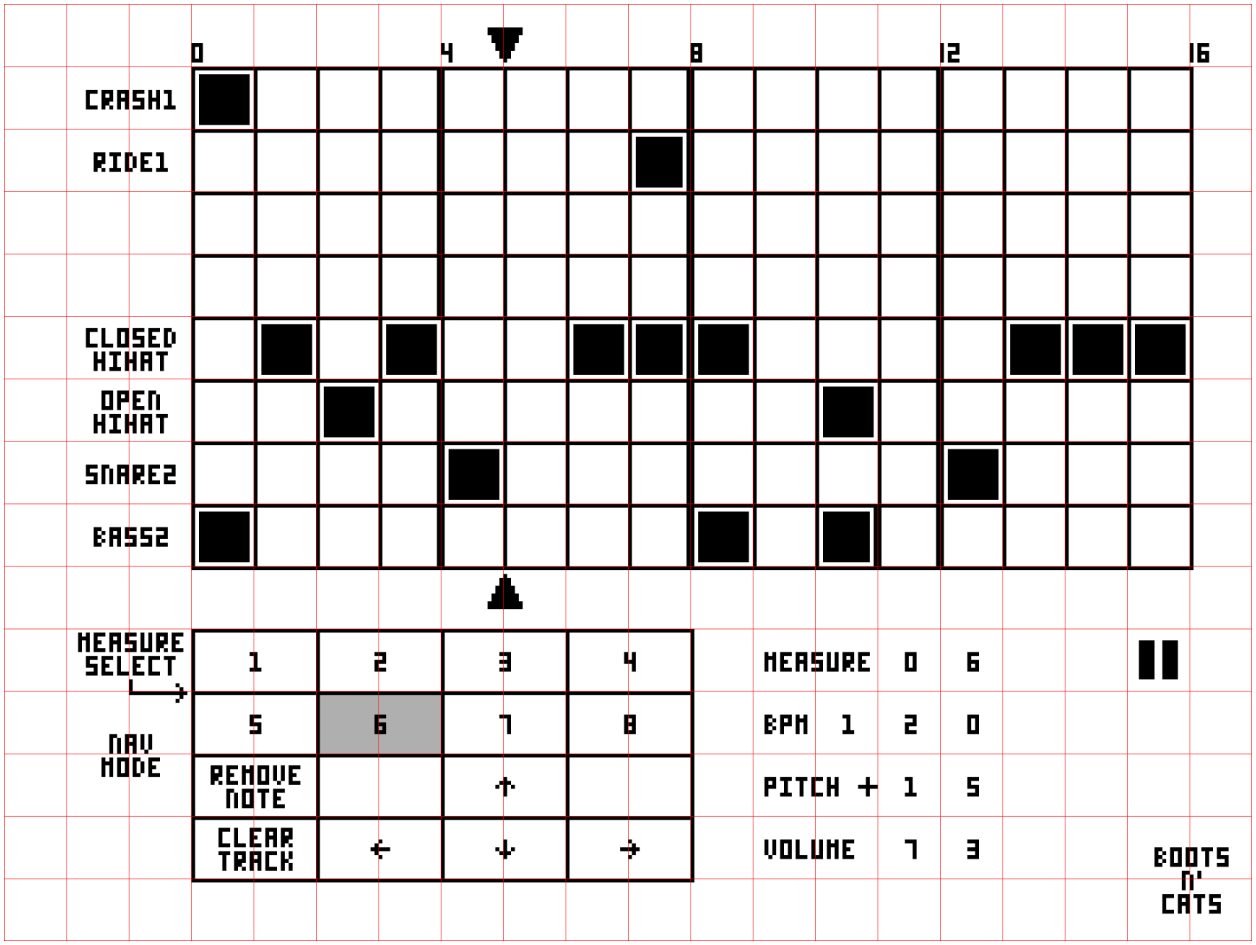
The primary way to navigate to different grid cells is with the arrow keys (i.e., the bottom-right of the Roland 4×4 grid of pads). The selected cell is highlighted with a light gray background, and the user can add/remove notes, along with changing the pitch and volume of the individual notes they added. Keep hitting right at the right edge of the measure goes to the next measure, and the measure wraps around after 8 (and vice-versa for left). Hitting the measure numbers jumps to the start of those measures. Selecting a cell with a note in it (or filling an empty cell with a note) will play that note, with all its effects applied.



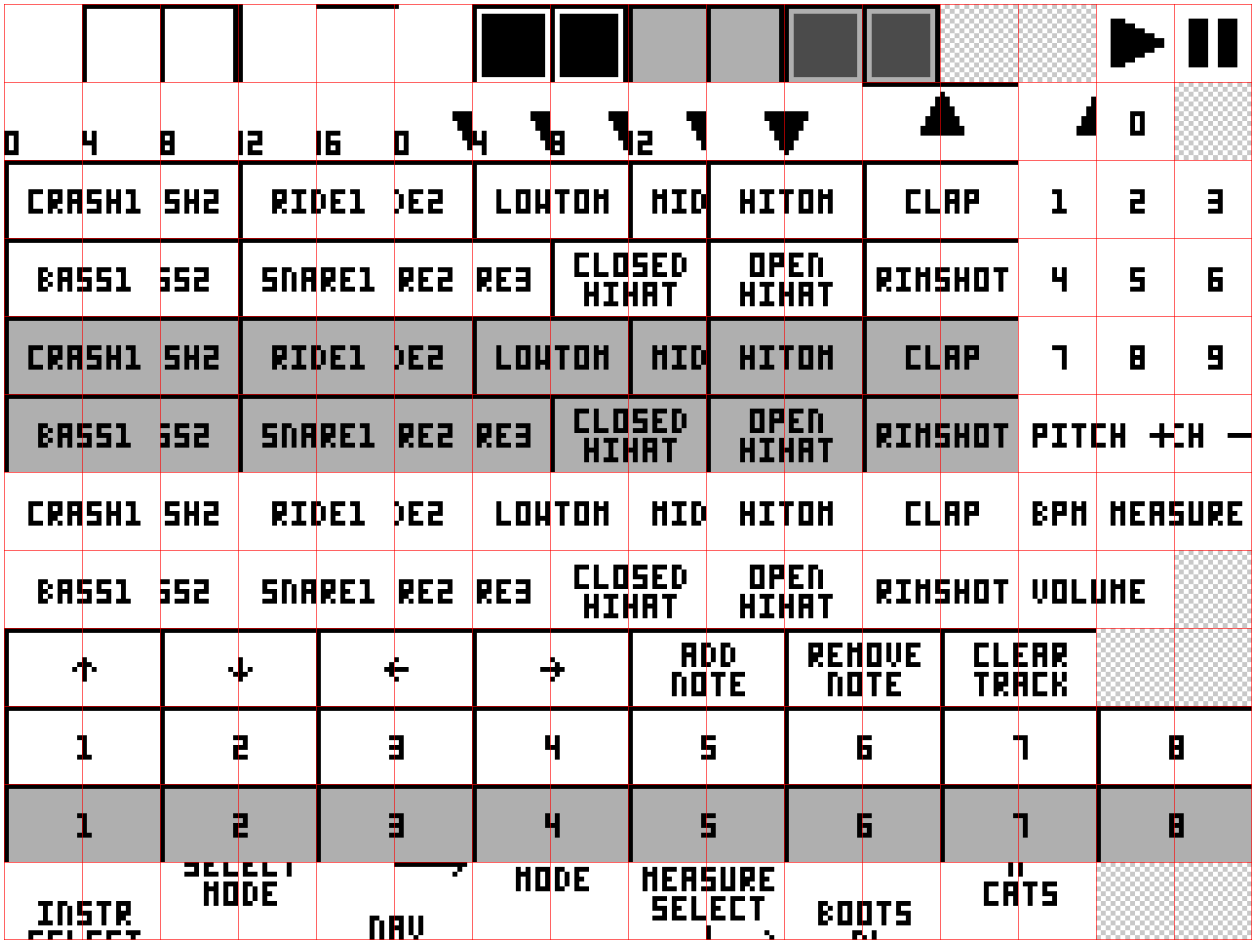
*Example UI (in instrument selection mode) with red tile lines overlaid on top for visualization only. The instrument of the current track is highlighted. The 4×4 grid of instruments mirrors the **Red** pads on the Roland.*

When in instrument selection mode, the user cannot change their selected cell with the arrow pads. Instead, the 4×4 grid of pads now selects the instrument for the track which the selected cell is on. By default (at startup), a track is empty and does not have an instrument (and thus also does not play a sound when a user fills a note or selects a

note), and no instrument is highlighted in instrument select mode. Clearing a track in navigation mode also removes the assigned instrument (along with all notes).



Example UI during playback. Two arrows at the top and bottom of the grid indicate where the playhead currently is and moves across the grid at the speed specified by the BPM. The playback starts on the column of the grid cell the user selected before playback begins. The playhead will automatically change measures at their ends, and measure 8 wraps back to measure 1. Pausing the playhead selects the grid cell of the instrument of the grid cell the user previously selected at the column/beat where the playhead was paused at.



The complete tileset for the UI (with red tile lines overlaid for visualization purposes). Each tile is 16×16 pixels with 4 possible colors (white, light gray, dark gray, black). There are 16×12 tiles in total (with some non-filled-in empty tiles represented with a checkerboard pattern here), though the current hardware implementation supports up to 16×16 tiles without needing to recompile and allocate additional BRAM.

Since we can easily run into BRAM memory issues if we attempt to display the UI with traditional display buffers (with the full 640×480 resolution of the VGA display), per Prof. Edwards's recommendation, we instead built the UI with tiles. For ease of design (since we are drawing individual pixels) and visibility, along with a fun retro aesthetic to our UI, we treat each 2×2 pixel on the VGA display as one large "macropixel", so we have an effective resolution of 320×240. With a tile size of 16×16 pixels, this leaves 20×15 visible tiles on the display. Since each pixel on any tile can only be one of four colors, each pixel only requires 2 bits of storage, which makes it fit within BRAM very comfortably.

4. Software Details

Our software is implemented in a single C program, `drum.c`, which manages all sequencer logic, hardware communication, and display rendering. The program maintains a global `sequencer_state` struct containing the 128×8 sequence grid, the active instrument assignment per track, the current cursor position, BPM, playback state, and edit mode. All access to this shared state is protected by a `pthread_mutex`.

After startup, the main program spawns two additional threads, for a total of three threads. The main thread loops on USB bulk transfers from the SP-404 MKII, decodes incoming MIDI packets, and dispatches them to update sequencer state. It also writes directly to the FPGA audio engine for single-note preview playback when the user toggles or hovers over an active cell. The playback thread sleeps on a condition variable and wakes when the user starts playback. It fires one Avalon-MM write to the audio engine per 16th note step, sleeping for a BPM-derived interval between steps using the Linux system call `nanosleep()`. The GUI thread runs at approximately 60 Hz, taking a snapshot of sequencer state via `get_sequencer_state()`, converting it to a tilemap via `state_to_tiles()`, and uploading the result to the FPGA display peripheral via `ioctl` on `/dev/ui`.

The program communicates with two hardware interfaces. The first is the FPGA audio engine, memory-mapped via `mmap()` over `/dev/mem` at physical address 0xFF200020. Triggering a voice requires a single 32-bit write to this register encoding the instrument ID, gain, and pitch ratio. The second is the VGA display peripheral, exposed as a character device at `/dev/ui` by a custom kernel module. The GUI thread writes tilemap, tileset, and palette data to this device via `ioctl`, which the FPGA peripheral reads to drive the VGA output.

Global Sequencer State Variable

```c
/* Complete sequencer state. The UI calls get_sequencer_state() to pull a snapshot
   and render; drum.c never pushes to the UI. */
typedef struct {
    cell       sequence[NUM_STEPS][MAX_TRACKS];   /* [step][track] */
    int          tracks[MAX_TRACKS];              /* instrument_id per slot, or -1 if unassigned */
    unsigned int measure;                         /* [1, 8]: current measure */
    unsigned int global_bpm;                      /* [64, 192] */
    unsigned int cursor_step;                     /* [0, 15]: step offset within current measure */
    unsigned int cursor_track;                    /* [0, 7]: current track slot */
    unsigned int playback_state;                  /* 0 = editing, 1 = playing back */
    unsigned int playback_step;                   /* [0, 15]: step within current measure during playback */
    unsigned int edit_mode;                       /* NAVIGATION, INSTRUMENT_SELECT */
} sequencer_state;
```

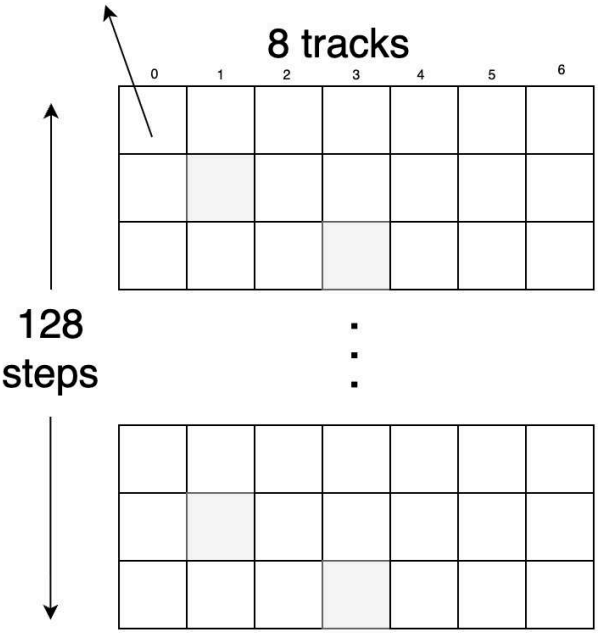
sequencer_state struct definition

The sequencer state is the central data structure of our software, shared across all three threads. It is initialized as a single global `sequencer_state` struct, protected by a `pthread_mutex` to prevent data races across the three threads. The main thread updates it in response to the user's MIDI input, the playback thread reads it to determine which notes to fire and advances the playback position within it, and the GUI thread takes a regular snapshot (~60Hz) of it each frame via `get_sequencer_state()` to render the display. All audio engine and GUI logic depends on this single struct; it serves as the source of truth for the state of the drum machine at any point in time.

Sequence Array and Tracks Array

The sequence array is a 128×8 2D array of cell structs, where the first dimension indexes the 128 steps across all 8 measures (16 steps per measure) and the second dimension indexes the 8 tracks. Each cell struct contains three fields: an active flag, a pitch value (−24 to +24 semitones), and a volume value (0–99), representing whether the step on that track is active (to be played), as well as its specified pitch and specified volume. The active tracks array is a parallel array of 8 integers, where each entry holds the instrument ID assigned to that track, or −1 if the track is unassigned.

```c
typedef struct {
    uint8_t active;  /* 0 or 1, default 0 */
    int8_t  pitch;   /* -24 to 24 semitones, default 0 */
    uint8_t volume;  /* 0-99, default 50 */
} cell;
```



0	1	2	3	4	5	6
-1	7	-1	13	-1	-1	-1

The 128x8 2D sequence array, comprising cell structs (left) and corresponding active tracks array, containing instrument IDs corresponding to each track (right)

Together, these two arrays fully describe what gets played and when. The sequence array determines at which steps and with what pitch and volume that instrument fires, and the active tracks array determines which instrument sound each track uses

Pre-Processing Pitch to Hardware

The SP-404 MKII's pitch knob (CTRL 2) outputs a 7-bit MIDI value in the range [0, 127]. We linearly map this range down to a semitone offset in [−24, +24] using the following formula:

```
semitone_offset = (int)value * 48 / 127 - 24;
```

This gives the user two octaves of pitch control in each direction from the default. We then convert this semitone offset into a frequency scaling ratio that the hardware can use directly.

```
None
float ratio = powf(2.0f, semitone_offset / 12.0f);
```

Then we convert that decimal ratio into fixed-point by multiplying by `2^10 = 1024`:

```
None
uint16_t pitch_fixed = (uint16_t)(ratio * 1024.0f + 0.5f);
```

So here are some examples:

```
None
-12 semitones -> ratio = 0.5  -> pitch_fixed = 512
  0 semitones -> ratio = 1.0  -> pitch_fixed = 1024
 12 semitones -> ratio = 2.0  -> pitch_fixed = 2048
 24 semitones -> ratio = 4.0  -> pitch_fixed = 4096
```

To transmit this value to the hardware, we pack the integer into bits [23:11] of the audio engine register:

```
REG[23:11] = pitch_fixed;
```

The FPGA peripheral interprets this field as a `Q3.10` fixed-point playback speed. At every 44.1kHz sample tick, the audio engine increments the voice's phase accumulator by this amount. A `pitch_fixed` value of `1024` corresponds to consuming exactly one sample per tick, resulting in standard playback.

Conversely, a `pitch_fixed` of `2048` doubles the advancement speed, shifting the audio one octave higher.

```
phase += pitch_fixed;
sample_index = phase >> 10;
```

Consequently, when `pitch_fixed` equals `2048` (`+12` semitones), the voice traverses its PCM data at twice the nominal rate, producing an output pitch that is one octave above the fundamental.

5. Software-Hardware Interface

Hardware-Software Interface for Playing Back User Sequence

The FPGA Avalon-MM slave exists as a memory-mapped entity within the HPS address space, represented by a single `uint32_t` register. To facilitate communication, our `drum.c` software utilizes `mmap()` to obtain a virtual address for this register, allowing the program to allocate voices and trigger playback by writing track data directly into the FPGA register.

The HPS is responsible for triggering FPGA playback in the following two scenarios:

1. **Sequence Playback**
   In standard sequencer playback mode, the HPS traverses the sequence grid stored in memory. For every step in the sequence, it writes the track data exclusively for active tracks to REG[0]. Every write to this register immediately allocates one voice and triggers playback, while inactive tracks are skipped entirely.
2. **Immediate Instrument Preview**
   When a user is editing in instrument select mode, the HPS performs a single write to REG[0] to initiate an instantaneous audio preview. To initialize the preview, the HPS assigns the following default parameters to the write: `instrument_id` set to the target instrument, `pitch ratio`=`1.0` (fixed-point `1024`), and `gain`=`64`.

Register Map

```
None
REG[0]   Voice/Track step data (Single Register)
```

Track Register Layout

The track register defines the layout for the playback parameters of a single voice or track.

```
None
bits [3:0]    instrument_id (4 bits, 0-15)
bits [10:4]   gain (7 bits, 0-127)
bits [23:11]  pitch ratio (13-bit fixed-point)
bits [31:24]  unused
```

The `instrument_id` field maps to one of the 16 drum samples. The `gain` field defines the per-step-and-track volume in a range of 0-127; the software is responsible for rescaling the user-facing 0-99 values to this hardware range. The `pitch ratio` field provides the playback scaling factor in fixed-point (Q3.10), meaning the real value is obtained by dividing by 1024. The `active` bit is no longer needed since the write operation itself acts as the trigger, leaving the remaining bits unused.

Upon receiving a write to REG[0], the FPGA immediately parses the playback parameters, allocates an available voice, and outputs the audio stream mixed with other active voices.

6. FPGA Peripheral Details

Audio Playback

The main audio_engine module that runs our hardware takes a 50MHz clock signal, write bit, reset bit, and [31:0] writedata as inputs from software and outputs a [15:0] audio_out to the DAC in the FPGA. A separate module called sample_clock also generates a sample_tick signal that pulses high at 44.1kHz for sample playback. At startup, the sampled amplitudes as hex values for our audio are loaded into [15:0] pcm_mem for the number of samples, 156395. We also hardcode a lookup table called sample_index with the starting indices of all 16 samples plus a final entry for the final index.

audio_engine has two main code blocks which deal with processing the parameters sent by writedata and mixing all actively playing notes on each cycle of sample_clock. To store and access all of the data for each note played, we allocate 64 voice_t structs as voices which have the following structure:

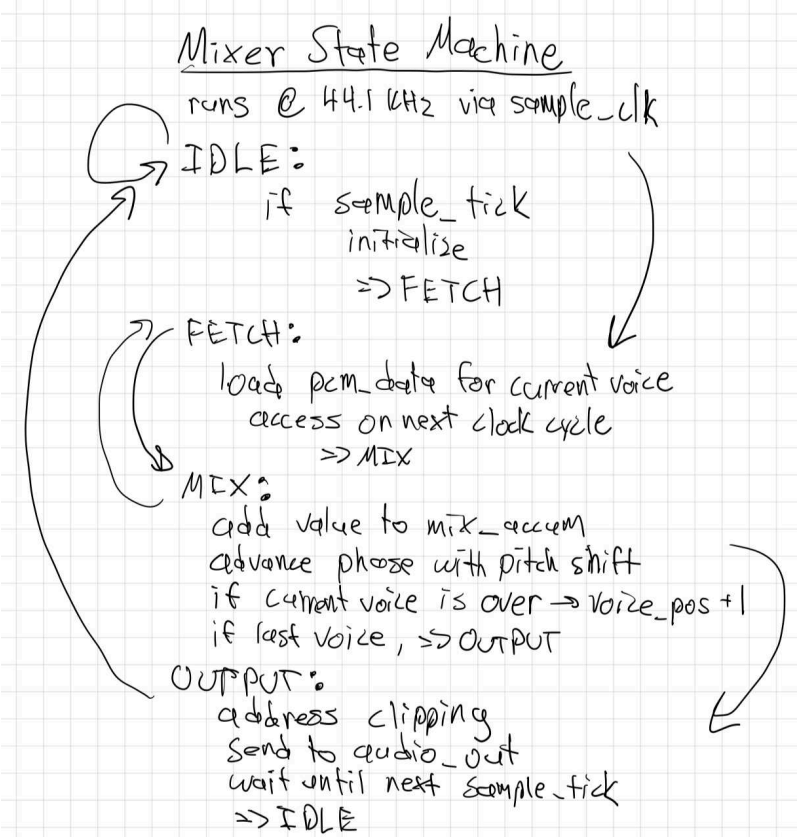
```
typedef struct packed {

    logic active;
    logic [27:0] phase;
    logic [17:0] end_addr;
    logic [12:0] ratio;
    logic [6:0] gain;

} voice_t;

voice_t voices[0:NUM_VOICES-1];
```

The active bit determines whether or not the note is still playing. The phase value is a 10bit shifted value of the current iterated index from pcm_mem to keep track of where the sound is in playback. End_addr stores the final index for the sample so we know when to end playback. The ratio value lets us know how much to scale our resampling to change pitch, and the gain value scales the amplitudes.

The first code block triggers when the write bit is high and handles storing the parameters given from writedata into the first inactive instance in voices.

```
    if (write) begin
        voice_found = 0;
        for (i=0; i<NUM_VOICES; i=i+1) begin
            if (!voices[i].active && !voice_found) begin
                voices[i].active <= 1'b1;
                voices[i].phase <= sample_index[writedata[4:1]] << 10;
                voices[i].end_addr <= sample_index[writedata[4:1]+1];
                voices[i].gain <= writedata[11:5];
                voices[i].ratio <= writedata[24:12];
                voice_found = 1;
            end
        end
    end
```

The second code block is a state machine with the following logic:



Because we want to play back our samples at 44.1kHz, we found that we have about 1100 system clock cycles to handle mixing samples for up to 64 voices. Our architecture triggers at every sample_tick and mixes voices one by one taking two cycles each time to read from pcm_mem and then add a value to a [31:0] mix_accum register. [5:0] voice_pos keeps track of where we are in the total 64 voices so that none get double counted and so that we can stop iterating after the last one. At this point, in the OUTPUT state, the magnitude of mix_accum is clipped if necessary and then output to audio_out. At this point the mixer remains idle until the next sample_tick.

We chose this architecture because it spread out processing among multiple clock cycles so as not to take up too many resources on the board and also because it has a latency of only 128 system clock cycles which is on the order of a few microseconds and imperceptible to the human ear. It's also easily scalable for additional features.

GUI Display

The GUI display hardware code is based on the sample VGA Tile Graphics code and tutorial as provided on the CSEE 4840 website.

The GUI display hardware lives in a single peripheral built around ui.sv, which wraps tiles.sv (the VGA pixel renderer), vga_counters.sv (the VGA timing generator), and three instances of a generic twoportbram.sv module that holds the tilemap, tileset, and palette. ui.sv exposes a single Avalon-MM slave port (8-bit data, 17-bit address) for memory-mapped writes from software, plus a conduit of VGA signals (R/G/B[7:0], HS, VS, BLANK_n, and the pixel clock) that drive the FPGA's VGA DAC directly. The peripheral works in-between two clock domains: a 50 MHz mem_clk from clk_0 driving the Avalon side, and a 25.175 MHz VGA_CLK from a dedicated vga_pll driving the pixel rendering on the VGA display.

The 17-bit address space is partitioned into three regions, each backed by its own dual-port BRAM. Addresses 0x00000–0x001FF hold the tilemap (one byte per cell of a 32×16 grid, only 20×15 visible on screen); 0x00200–0x0020F hold the palette (four 24-bit RGB colors); and 0x10000–0x1FFFF hold the tileset (256 tile slots × 16×16 pixels × 1 byte per pixel, with only the lower 2 bits of each byte actually stored in hardware). The address decoder is purely combinational: Bit 16 of the address selects the tileset versus everything else, and within the lower half, bit 9 selects palette versus tilemap. The corresponding write-enable signal (ts_we, tm_we, or palette_we) gates the destination BRAM's port-2 write input.

Because each palette entry is 24 bits but our Avalon bus is only 8 bits wide, writing one color requires four bus transactions. To avoid the VGA pipeline observing a half-finished

color mid-frame, ui.sv uses a 24-bit staging register creg plus a "commit-byte" protocol: Software writes the red, green, and blue bytes to the bottom three byte-offsets of a 4-byte palette slot, which load into creg[7:0], creg[15:8], and creg[23:16] respectively. Then, a subsequent write to the 4th byte (data ignored) asserts palette_we, which atomically copies all 24 bits of creg into palette[address[3:2]]. This way each color enters the palette as a single atomic update.
The track register defines the layout for the playback parameters of a single voice or track.

```
None
case (address[1:0])
  2'h 0: creg_write[0] = write;      // load creg[7:0]   <- red
  2'h 1: creg_write[1] = write;      // load creg[15:8]  <- green
  2'h 2: creg_write[2] = write;      // load creg[23:16] <- blue
  2'h 3: palette_we    = write;      // palette[i] <- creg
endcase
```

The three BRAMs are instantiated as true dual-port memories (M10K blocks for tilemap and tileset, distributed RAM for the tiny palette, as determined by Quartus), each with two independent ports. Port 2 is clocked on mem_clk and serves Avalon writes; port 1 is clocked on VGA_CLK and is read-only by the pixel pipeline (rendering pixels to the VGA display). Because port 1 never writes, port-1/port-2 collisions are impossible regardless of clock skew, and Quartus infers the cross-clock memory directly from this code pattern with no synchronizers or FIFOs.

Drawing each pixel requires three dependent BRAM lookups: Tilemap → tileset → palette. Since each feeds into the next, they cannot be parallelized. tiles.sv accordingly implements a 3-stage pixel pipeline clocked at VGA_CLK. Stage 1 reads the tilemap at address {vcount[8:5], hcount[9:5]} to fetch the tile ID at the current screen cell. Stage 2 reads the tileset at {tilenumber, vcount[4:1], hcount1} to fetch the 2-bit palette index for the current in-tile pixel. Stage 3 reads the palette at colorindex to fetch the final 24-bit RGB color out of the four color options. The VGA timing signals VGA_HS and VGA_BLANK_n are flopped in parallel at each stage so they stay aligned with the pixel they describe at the output:

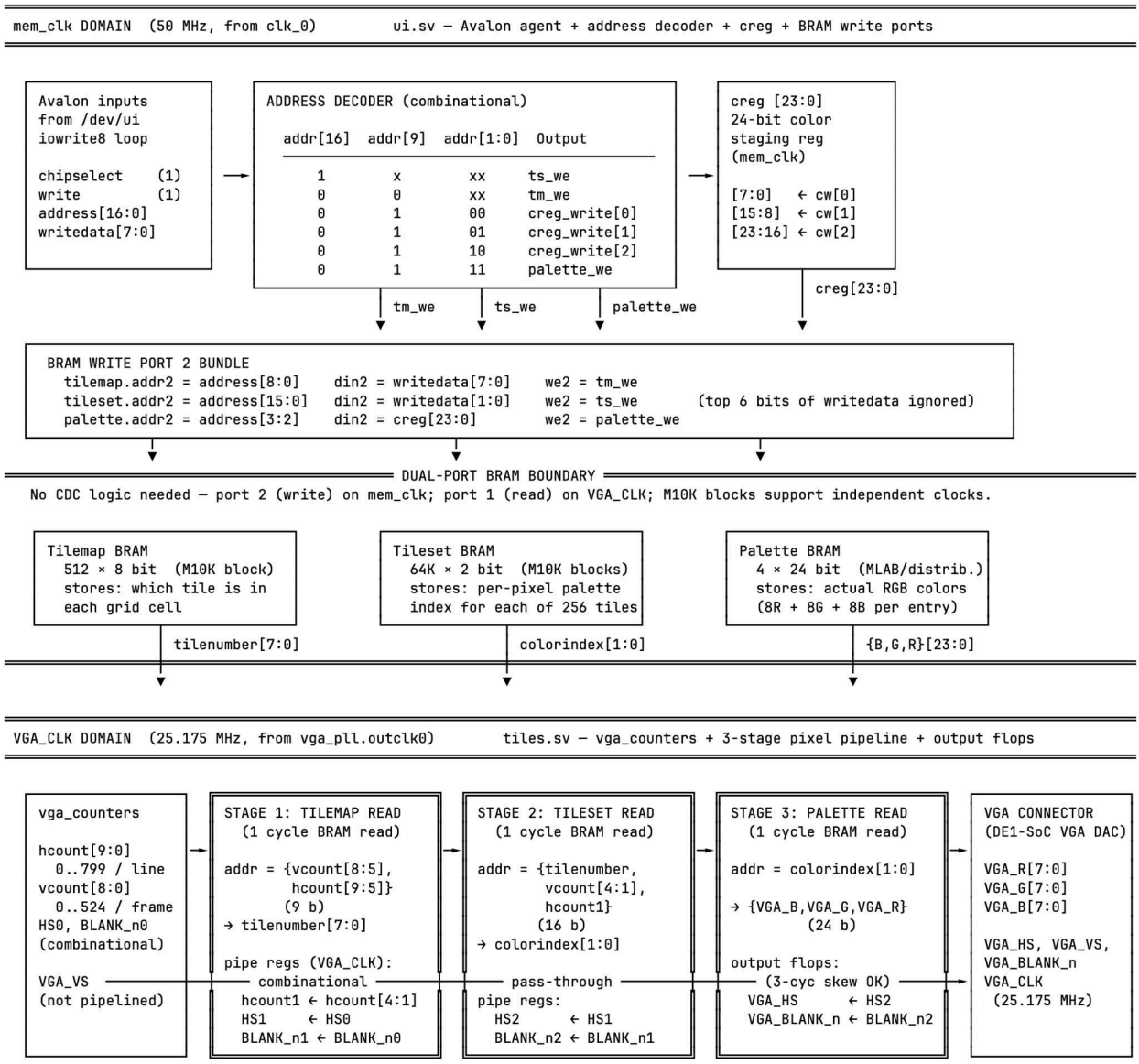
```
None
always_ff @(posedge VGA_CLK)
  { hcount1, VGA_BLANK_n1, VGA_HS1 } <=
```

```
{ hcount[4:1], VGA_BLANK_n0, VGA_HS0 };
```

VGA_VS is the one exception: It passes combinationally from vga_counters straight to the output, because it changes only twice per frame (during lines 490–491) so a 3-cycle pipeline skew is imperceptible.

vga_counters generates standard 640×480 @ 60 Hz timing (800 horizontal × 525 vertical pixel counts per frame, with active video in the upper-left 640×480 region) driven by the 25.175 MHz pixel clock. Even though our logical framebuffer is 320×240, every BRAM address computed above ignores hcount[0] and vcount[0], which doubles every logical pixel into a 2×2 block of physical pixels on screen. This pixel-doubling comes from bit-slicing for free with no upscaling logic, and gives us the chunky retro aesthetic we wanted while keeping the tileset memory four times smaller than a true-640×480 implementation would have required. This also made initially hand-designing and drawing the tileset more straightforward, as the tiles are often designed pixel-by-pixel, especially those with numbers and characters where we were basically doing font design.

We chose this architecture for three reasons. First, using the dual-port BRAM as the synchronization element between two clock domains removes any need for explicit synchronizers or FIFOs between the 50 MHz Avalon side and the 25.175 MHz VGA side, eliminating a major source of timing bugs. (Past labs approximated the VGA clock with a 25 MHz clock that was possible to be manually synced with the 50 MHz Avalon clock, but the dual-port BRAM setup just made it so that we didn’t have to even think about this.) Second, even though each pixel takes 3 cycles to propagate through the BRAM lookups, the pipeline is fully filled. This means a new RGB value emerges every VGA_CLK period, so the renderer produces one pixel per clock with no stalls. Third, separating the static tileset (uploaded once at boot) from the live tilemap (rewritten every frame at 60 Hz) means software only has to push around 300 bytes per frame to keep the screen current, while the bulk of the visual data sits in BRAM untouched after the initial setup, which is why tiles are so much more efficient than full framebuffers.



Detailed block diagram of how the GUI is displayed. The state of the GUI is stored as three BRAMs: Tilemap (current tile IDs of the tiles being displayed/rendered), tileset (set of all tiles that can be displayed), and palette (RGB codes for the 4 available colors). These are dual-port BRAMs which is written to by the mem_clk domain (i.e., hardware controlled by the 50 MHz memory clock) and read by the VGA_CLK domain (i.e., hardware controlled by the 25.175 MHz VGA display clock) so we can have "asynchronous" updates that come from the software and instantaneous-ish updates shown on the VGA display.

7. Resource Budget

Name	Notes	Memory (bit)
pcm_samples.hex	3.546s, 44.1kHz, loaded in 16b pcm_mem registers	156,395
[31:0] sample_index [0:16]	Indexes for the starting samples of each instrument and final index of pcm_mem	544
[64:0] voices [0:63]	Registers to store data and parameters for each playing note. Up to 64 at a time. logic active; logic [27:0] phase; logic [17:0] end_addr; logic [12:0] ratio; logic [6:0] gain;	4160
Mixing regs	Keeping track of logic for the main mixer FSM logic [5:0] voice_pos; logic signed [31:0] pcm_data; logic signed [31:0] mix_accum; mixer_state_t state [1:0];	62
Tileset	256 tile slots, 16×16 pixels per slot, 2 bits per pixel (4 colors total) The actual tileset.bin uses 8 bits (1 byte) each **only at load time** for ease of addressing with (only slightly slows down startup), actual FPGA BRAM usage is 2 bits each with ts_din(writedata[1:0]) In practice we only use about 192 tile slots	131,072
Palette	8 bits per R/G/B (24 bits total), 4 colors total	96
(Live) Tilemap	32×16 tiles, 8 bits per tile (to address all 256 possible tile slots) Only 20×15 tiles are visible. (Actual displayed resolution is 320×240, we display each pixel as a 2×2 pixel on screen.) The extra tiles allow easy address conversion by bit-slicing VGA counters	4096
	Total memory (bit)	296425

Appendix A: Device MIDI to Button Mapping

The HPS receives input from the SP-404MKII as a stream of 4-byte USB-MIDI packets of the form (`cin, status, data1, data2`). The first byte (`cin`) encodes the USB MIDI packet type and is ignored. The second byte (status) encodes both the event type and the channel: the upper four bits specify the message type (e.g., Note On, Note Off, Control Change), and the lower four bits specify the channel. The third byte (`data1`) identifies the control (pad, button, or knob), and the fourth byte (`data2`) encodes the velocity or control value.

The system decodes each packet by extracting the channel from the status byte to determine the current editing mode, and then using `data1` to identify the specific control. The mappings used are defined below.

Note that the editing mode buttons (“A/F”, “B/G”, “C/H”, “D/I”, “E/J”) do not generate independent MIDI events. Instead, they modify the channel encoded in the status byte of subsequent pad events, thereby changing the interpretation of numbered pad presses. In contrast, the numbered pads, knobs, and the remaining pads (“EXT SOURCE” and “SUB PAD”) always generate explicit MIDI events when pressed or adjusted.

Channel Mapping

Button	State	`status` (press)	`status` (release)	Channel
A/F	A (solid)	0x95	0x85	5
A/F	F (flash)	0x90	0x80	0
B/G	B (solid)	0x91	0x81	1
B/G	G (flash)	0x96	0x86	6
C/H	C (solid)	0x92	0x82	2
C/H	H (flash)	0x97	0x87	7

D/I	D (solid)	0x93	0x83	3
D/I	I (flash)	0x98	0x88	8
E/J	E (solid)	0x94	0x84	4
E/J	J (flash)	0x99	0x89	9

Pad Mapping

`data1`	Pad Number
0x30	1
0x31	2
0x32	3
0x33	4
0x2c	5
0x2d	6
0x2e	7
0x2f	8
0x28	9
0x29	10
0x2a	11
0x2b	12
0x24	13
0x25	14
0x26	15

0x27	16

EXT SOURCE

```
status (press)   = 0x90
status (release) = 0x80
data1            = 0x23
data2            = 0x7f (press), 0x40 (release)
```

The “EXT SOURCE” button behaves as a toggle at the user interface level, but generates a standard press/release pair for each interaction.

Knobs Mapping

Control	`data1` (controller ID)	`data2` (value)
CTRL 1	0x10	0x00–0x7f
CTRL 2	0x11	0x00–0x7f

Appendix B: Hardware Files

audio_engine.sv, Noel Gomez

```
None
module audio_engine (
    input  logic clk,
    input  logic reset,
    input  logic write,
    input  logic [2:0] address,
    input  logic [31:0] writedata,

    output logic signed [15:0] audio_out
);

    localparam NUM_VOICES = 64;
    localparam TOTAL_PCM_SAMPLES = 156395;

(* ramstyle = "M10K" *)
    logic signed [15:0] pcm_mem [0:TOTAL_PCM_SAMPLES-1];

    initial begin
        $readmemh("pcm_samples.hex", pcm_mem);
    end

// ============================================================
// SAMPLE START TABLE
// ============================================================

logic [31:0] sample_index [0:16];

initial begin

    sample_index[0]  = 0;
    sample_index[1]  = 15744;
    sample_index[2]  = 31487;
    sample_index[3]  = 47231;
    sample_index[4]  = 63019;
    sample_index[5]  = 72192;
    sample_index[6]  = 86039;
    sample_index[7]  = 94727;
    sample_index[8]  = 106634;
    sample_index[9]  = 115983;
    sample_index[10] = 120481;
    sample_index[11] = 128331;
    sample_index[12] = 129478;
    sample_index[13] = 138121;
    sample_index[14] = 148220;
    sample_index[15] = 149234;
    sample_index[16] = TOTAL_PCM_SAMPLES;
```

```
end

logic sample_tick;

sample_clock clkgen (

    .clk(clk),
    .reset(reset),
    .sample_tick(sample_tick)

);

typedef struct packed {

    logic active;
    logic [27:0] phase;
    logic [17:0] end_addr;
    logic [12:0] ratio;
    logic [6:0] gain;

} voice_t;

voice_t voices[0:NUM_VOICES-1];

// ============================================================
// MIXER STATE
// ============================================================

typedef enum logic [1:0] {

    IDLE,
    FETCH,
    MIX,
    OUTPUT

} mixer_state_t;

mixer_state_t state;

// ============================================================
// MIXER REGISTERS
// ============================================================

logic [5:0] voice_pos;
logic signed [31:0] pcm_data;
logic signed [31:0] mix_accum;

logic voice_found;
integer i;
```

```
always_ff @(posedge clk) begin

    if(reset) begin
        state <= IDLE;
        voice_pos <= 0;
        mix_accum <= 0;
        audio_out <= 0;

        for(i = 0; i < NUM_VOICES; i = i + 1) begin
            voices[i].active <= 0;
        end
    end
    else begin

        if (write) begin
            if (address == 3'd1) begin
                // Runtime sample-index table write (Avalon address 1).
                //   writedata[4:0]  = entry index (0..16)
                //   writedata[31:5] = sample offset (up to 27 bits)
                // Lets software override the initial-block defaults
                // without recompiling.  Address 0 (below) still triggers
                // voices exactly as before.
                sample_index[writedata[4:0]] <= writedata[31:5];
            end else begin
                // Address 0 (default): trigger a voice.
                voice_found = 0;
                for (i=0; i<NUM_VOICES; i=i+1) begin
                    if (!voices[i].active && !voice_found) begin
                        voices[i].active <= 1'b1;
                        voices[i].phase <= sample_index[writedata[3:0]] << 10;
                        voices[i].end_addr <= sample_index[writedata[3:0]+1];
                        voices[i].gain <= writedata[10:4];
                        voices[i].ratio <= writedata[23:11];
                        voice_found = 1;
                    end
                end
            end
        end

    // ============================================================
    // MAIN MIXER FSM
    // ============================================================

        case(state)

        // ====================================================
        // WAIT FOR NEXT AUDIO SAMPLE PERIOD
        // ====================================================

        IDLE: begin
```

```
            if(sample_tick) begin
                mix_accum <= 0;
                voice_pos <= 0;

                state <= FETCH;
            end
        end

        // ====================================================
        // SEND ADDRESS TO BRAM
        // ====================================================

        FETCH: begin
            pcm_data <= pcm_mem[voices[voice_pos].phase >> 10];
            state <= MIX;
        end

        // ====================================================
        // BRAM DATA ARRIVES THIS CYCLE
        // ====================================================

        MIX: begin
            if(voices[voice_pos].active) begin
                //!!!NOTE: MAKE SURE SOFTWARE SENDS 0-127 GAIN VALUE AND NOT 0-99 SO BIT
SHIFT IS ACCURATE!!!
                // apply gain + accumulate
                mix_accum <= mix_accum + (($signed(pcm_data) *
$signed(voices[voice_pos].gain)) >>> 7);
                // advance playback phase
                voices[voice_pos].phase <= voices[voice_pos].phase + voices[voice_pos].ratio;
                // deactivate finished voice
                if(((voices[voice_pos].phase + voices[voice_pos].ratio) >> 10) >=
voices[voice_pos].end_addr) begin
                    voices[voice_pos].active <= 1'b0;
                end
            end
            // move to next voice
            if(voice_pos == NUM_VOICES-1) begin
                state <= OUTPUT;
            end
            else begin
                voice_pos <= voice_pos + 1;
                state <= FETCH;
            end
        end

        // ====================================================
        // OUTPUT FINAL MIXED SAMPLE
        // ====================================================
```

```
        OUTPUT: begin
            // clipping
            if(mix_accum > 32767) begin
                audio_out <= 16'sd32767;
            end
            else if(mix_accum < -32768) begin
                audio_out <= -16'sd32768;
            end
            else begin
                audio_out <= mix_accum[15:0];
            end

            state <= IDLE;
        end

        endcase
    end
end

endmodule
```

i2s_tx.sv, Aaron Zhu

```
None
module i2s_tx (
    input  logic        clk,
    input  logic        reset,
    input  logic [15:0] audio_in,
    output logic        aud_xck,
    output logic        aud_bclk,
    output logic        aud_daclrck,
    output logic        aud_dacdat
);

    // XCK = 50 MHz / 4 = 12.5 MHz
    logic xck_cnt;
    always_ff @(posedge clk)
        if (reset) begin xck_cnt <= 0; aud_xck <= 0; end
        else begin
            xck_cnt <= ~xck_cnt;
            if (xck_cnt) aud_xck <= ~aud_xck;
        end

    // BCLK = 50 MHz / 18 ≈ 2.78 MHz (toggle every 9 cycles)
    logic [3:0] bclk_cnt;
    logic       bclk_r, bclk_prev;
    always_ff @(posedge clk)
        if (reset) begin bclk_cnt <= 0; bclk_r <= 0; end
        else if (bclk_cnt == 8) begin bclk_cnt <= 0; bclk_r <= ~bclk_r; end
```

```
        else bclk_cnt <= bclk_cnt + 1;

    assign aud_bclk = bclk_r;
    always_ff @(posedge clk) bclk_prev <= reset ? 1'b0 : bclk_r;
    wire bclk_fall = bclk_prev & ~bclk_r;

    // 64-bit frame: pos 0-31 = left channel, 32-63 = right channel
    // Standard I2S: LRCK changes 1 BCLK before MSB, then 16 data bits, then zeros
    logic [5:0]  pos;
    logic [31:0] sreg;
    logic [5:0]  next_pos;
    assign next_pos = (pos == 63) ? 6'd0 : pos + 1'b1;

    always_ff @(posedge clk) begin
        if (reset) begin
            pos <= 0; sreg <= 0;
            aud_daclrck <= 0; aud_dacdat <= 0;
        end else if (bclk_fall) begin
            pos         <= next_pos;
            aud_daclrck <= next_pos[5];  // 0=left (pos 0-31), 1=right (32-63)
            aud_dacdat  <= sreg[31];

            // Reload shift reg at frame boundary and at channel midpoint;
            // left-justify 16-bit sample in the 32-bit slot (zeros for bits 16-31)
            if (pos == 63 || pos == 31)
                sreg <= {audio_in, 16'h0000};
            else
                sreg <= {sreg[30:0], 1'b0};
        end
    end

endmodule
```

sample_clock.sv, Noel Gomez

```
None
module sample_clock (
    input  logic clk,
    input  logic reset,
    output logic sample_tick
);

    localparam DIV = 1134;

    logic [10:0] counter;

    always_ff @(posedge clk) begin

        if(reset) begin
```

```
            counter <= 0;
            sample_tick <= 0;
        end

        else begin

            if(counter == DIV - 1) begin
                counter <= 0;
                sample_tick <= 1;
            end

            else begin
                counter <= counter + 1;
                sample_tick <= 0;
            end

        end

    end

endmodule
```

tiles.sv, Jordan Lin

```
None
module tiles
  (input logic         VGA_CLK, VGA_RESET,
   output logic [7:0]  VGA_R, VGA_G, VGA_B,
   output logic        VGA_HS, VGA_VS, VGA_BLANK_n,

   input logic            mem_clk,         // Clock for memory ports

   input logic [8:0]   tm_address,      // Tilemap memory port  (32x16 = 512)
   input logic            tm_we,
   input logic [7:0]   tm_din,
   output logic [7:0]  tm_dout,

   input logic [15:0]  ts_address,      // Tileset memory port  (256 tiles x 16x16 px)
   input logic            ts_we,
   input logic [1:0]   ts_din,
   output logic [1:0]  ts_dout,

   input logic [1:0]   palette_address, // Palette memory port  (4 colors)
   input logic            palette_we,
   input logic [23:0]  palette_din,
   output logic [23:0] palette_dout);

   logic [9:0]            hcount;          // From counters
   logic [8:0]            vcount;
```

```
logic [3:0]           hcount1;         // Pipeline registers (4 bits for 16-px tiles)
logic            VGA_HS0, VGA_HS1, VGA_HS2;
logic            VGA_BLANK_n0, VGA_BLANK_n1, VGA_BLANK_n2;

logic [7:0]           tilenumber;      // Memory outputs
logic [1:0]           colorindex;

/* verilator lint_off UNUSED */
logic               unconnected;     // Extra vcount bit from counters
/* verilator lint_on UNUSED */

vga_counters cntrs(.vcount( {unconnected, vcount} ),
           .VGA_BLANK_n( VGA_BLANK_n0 ),
           .VGA_HS( VGA_HS0 ),
           .*);

// Tilemap: 32 cols x 16 rows = 512 entries; addr = {row[3:0], col[4:0]}
twoportbram #(.DATA_BITS(8), .ADDRESS_BITS(9))
tilemap(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
    .addr1 ( { vcount[8:5], hcount[9:5] } ),
    .we1   ( 1'b0 ), .din1( 8'h X ), .dout1( tilenumber ),
    .addr2 ( tm_address ),
    .we2   ( tm_we ), .din2( tm_din ), .dout2( tm_dout ));

always_ff @(posedge VGA_CLK)                     // Pipeline registers
  { hcount1, VGA_BLANK_n1, VGA_HS1 } <=
    { hcount[4:1], VGA_BLANK_n0, VGA_HS0 };

// Tileset: 256 tiles x 16x16 px = 64K entries x 2 bits
//   addr = {tilenum[7:0], j[3:0], i[3:0]}
twoportbram #(.DATA_BITS(2), .ADDRESS_BITS(16))
tileset(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
    .addr1 ( { tilenumber, vcount[4:1], hcount1 } ),
    .we1   ( 1'b0 ), .din1( 2'h X ), .dout1( colorindex ),
    .addr2 ( ts_address ),
    .we2   ( ts_we ), .din2( ts_din ), .dout2( ts_dout ));

always_ff @(posedge VGA_CLK)                     // Pipeline registers
  { VGA_BLANK_n2, VGA_HS2 } <= { VGA_BLANK_n1, VGA_HS1 };

// Palette: 4 colors x 24 bits
twoportbram #(.DATA_BITS(24), .ADDRESS_BITS(2))
palette(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
    .addr1 ( colorindex ),
    .we1   ( 1'b0 ), .din1( 24'h X ), .dout1( { VGA_B, VGA_G, VGA_R } ),
    .addr2 ( palette_address ),
    .we2   ( palette_we ), .din2( palette_din ), .dout2( palette_dout ));

always_ff @(posedge VGA_CLK)                     // Pipeline registers
```

```
    { VGA_BLANK_n, VGA_HS } <= { VGA_BLANK_n2, VGA_HS2 };

endmodule
```

twoportbram.sv, Jordan Lin, mostly from class website

```
None
module twoportbram
  #(parameter int DATA_BITS = 8,  ADDRESS_BITS = 10)
   (input logic            clk1,  clk2,
    input logic [ADDRESS_BITS-1:0] addr1, addr2,
    input logic [DATA_BITS-1:0]    din1,  din2,
    input logic            we1,   we2,
    output logic [DATA_BITS-1:0]   dout1, dout2);

   localparam WORDS = 1 << ADDRESS_BITS;

   /* verilator lint_off MULTIDRIVEN */
   logic [DATA_BITS-1:0]        mem [WORDS-1:0];
   /* verilator lint_on MULTIDRIVEN */

   always_ff @(posedge clk1)
     if (we1) begin
    mem[addr1] <= din1;
    dout1 <= din1;
     end else dout1 <= mem[addr1];

   always_ff @(posedge clk2)
     if (we2) begin
    mem[addr2] <= din2;
    dout2 <= din2;
     end else dout2 <= mem[addr2];

endmodule
```

ui.sv, Jordan Lin

```
None
/*
 * Avalon memory-mapped agent peripheral that produces the drum-machine
 * UI tile display.  Mirrors labs/tiles/vga_tiles.sv but with our memory map.
 *
 * Memory map (17-bit byte address):
 *
 *   0x00000 - 0x001FF   Tilemap   (512 bytes, 1 byte per tile, only 20x15 used)
 *   0x00200 - 0x0020F   Palette   (4 colors x 4 bytes, byte 3 commits)
 *   0x10000 - 0x1FFFF   Tileset   (64K bytes, lower 2 bits = color index)
```

```
 *
 *   1xxxx xxxx xxxx xxxx   Tileset (address[16]=1)
 *   0xxxx xxx0 xxxx xxxx   Tilemap (address[16]=0, address[9]=0)
 *   0xxxx xxx1 xxxx xxbb   Palette (address[16]=0, address[9]=1)
 *
 * Palette commit (mirrors lab pattern): writing the lower 3 bytes of each
 * 4-byte group fills a 24-bit color register; writing the 4th byte commits
 * that register into palette memory.
 *
 *   | Offset (low 2 bits) | On Write              | On Read           |
 *   +---------------------+-----------------------+-------------------+
 *   |  0  (red)           | creg[7:0]   <- data   | palette[i].red    |
 *   |  1  (green)         | creg[15:8]  <- data   | palette[i].green  |
 *   |  2  (blue)          | creg[23:16] <- data   | palette[i].blue   |
 *   |  3  (commit)        | palette[i] <- creg    | always 0          |
 *
 * Color index i = address[3:2].
 */
module ui
  (input logic           clk, reset,                    // Avalon MM Agent port
   input logic           chipselect, write,             // read == chipselect & !write
   input logic [16:0] address,                        // 128K window
   input logic [7:0]  writedata,                      // 8-bit interface
   output logic [7:0] readdata,

   input logic        vga_clk_in, VGA_RESET,          // VGA signals
   output logic [7:0] VGA_R, VGA_G, VGA_B,
   output logic       VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n);

   logic [2:0]           creg_write;                    // Latch enable per byte
   logic           tm_we, ts_we, palette_we;      // Memory write enables
   logic [7:0]           tm_dout;                       // Data from tilemap
   logic [1:0]           ts_dout;                       // Data from tileset
   logic [23:0]       creg, palette_dout;            // Data to/from palette

   tiles tiles(.mem_clk        ( clk             ),
           .tm_address     ( address[8:0]    ), .tm_din     ( writedata      ),
           .ts_address     ( address[15:0]   ), .ts_din     ( writedata[1:0] ),
           .palette_address ( address[3:2]   ), .palette_din( creg           ), .*);
   assign VGA_CLK = vga_clk_in;

   always_comb begin                                    // Address Decoder
      {tm_we, ts_we, palette_we, creg_write, readdata } = { 6'b 0, 8'h xx };
      if (chipselect)
    if (address[16] == 1'b 1) begin                 // Tileset 1----------------
       ts_we    = write;                            //  Write to tileset mem
       readdata = { 6'h 0, ts_dout };               //  Read lower 2 bits; pad upper
    end else if (address[9] == 1'b 0) begin         // Tilemap 0------0--------
       tm_we    = write;                            //  Write to tilemap mem
       readdata = tm_dout;                         //  Read 8 bits
```

```
    end else                                        // Palette 0------1xxxxxxbb
       case (address[1:0])
         2'h 0 : begin readdata = palette_dout[7:0];   // Read red byte
               creg_write[0] = write;          // creg <- red
             end
         2'h 1 : begin readdata = palette_dout[15:8];  // Read green byte
               creg_write[1] = write;          // creg <- green
             end
         2'h 2 : begin readdata = palette_dout[23:16]; // Read blue byte
               creg_write[2] = write;          // creg <- blue
             end
         2'h 3 : begin readdata = 8'h 00;              // Always reads as 00
               palette_we = write;             // mem <- creg
             end
       endcase
   end

   always_ff @(posedge clk or posedge reset)
     if (reset) creg <= 24'b 0; else begin
    if (creg_write[0]) creg[7:0]   <= writedata;    // Write byte (color)
    if (creg_write[1]) creg[15:8]  <= writedata;    // to creg according to
    if (creg_write[2]) creg[23:16] <= writedata;    // creg_write bits
     end
endmodule
```

vga_counters.sv, Jordan Lin, mostly from class website

```
None
module vga_counters(
   input  logic       VGA_CLK, VGA_RESET,
   output logic [9:0] hcount, // 0-639 active, 640-799 blank/sync
   output logic [9:0] vcount, // 0-479 active, 480-524 blank/sync
   output logic       VGA_HS, VGA_VS, VGA_BLANK_n);

   logic endOfLine;
   assign endOfLine = hcount == 10'd 799;

   always_ff @(posedge VGA_CLK or posedge VGA_RESET)
     if (VGA_RESET)      hcount <= 10'd 797;
     else if (endOfLine) hcount <= 0;
     else                hcount <= hcount + 10'd 1;

   logic endOfFrame;
   assign endOfFrame = vcount == 10'd 524;

   always_ff @(posedge VGA_CLK or posedge VGA_RESET)
     if (VGA_RESET)      vcount <= 10'd 524;
     else if (endOfLine)
       if (endOfFrame)   vcount <= 10'd 0;
```

```
       else              vcount <= vcount + 10'd 1;

   // 656 <= hcount <= 751
   assign VGA_HS = !( hcount[9:7] == 3'b101 &
                      hcount[6:4] != 3'b000 & hcount[6:4] != 3'b111 );
   assign VGA_VS = !( vcount[9:1] == 9'd 245 ); // Lines 490 and 491

   // hcount < 640 && vcount < 480
   assign VGA_BLANK_n = !( hcount[9] & (hcount[8] | hcount[7]) ) &
                        !( vcount[9] | (vcount[8:5] == 4'b1111) );
endmodule
```

Appendix C: Software Files

drum.c, Aaron Zhu

None

```
#include <assert.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#ifndef I2C_SLAVE
#define I2C_SLAVE 0x0703
#endif
#include <sys/mman.h>
#include <sys/stat.h>
#include <pthread.h>
#include <time.h>
#include <math.h>
#include <libusb-1.0/libusb.h>
#include "drum.h"
#include "seq_load.h"
#include "ui.h"
#include "state_to_tiles.h"

#define VID               0x0582
#define PID               0x0281
#define EP_IN             0x84
#define IFACE             3

#define UI_DEV            "/dev/ui"
#define UI_DIR            "../ui"
#define GUI_INTERVAL_US   16666   /* ~60 Hz */

#define FPGA_BASE         0xFF200000u  /* LWH2F bridge base (must be page-aligned) */
#define FPGA_OFFSET       0x20u        /* audio_engine_0 at LWH2F + 0x20 */
#define FPGA_SPAN         0x00001000u  /* one page */

typedef struct {
    unsigned char note;
    int pad;
} pad_map;

static const pad_map pad_maps[] = {
    {0x30,  1}, {0x31,  2}, {0x32,  3}, {0x33,  4},
    {0x2c,  5}, {0x2d,  6}, {0x2e,  7}, {0x2f,  8},
    {0x28,  9}, {0x29, 10}, {0x2a, 11}, {0x2b, 12},
```

```
    {0x24, 13}, {0x25, 14}, {0x26, 15}, {0x27, 16},
};

static const char *channel_name(int ch) {
    switch (ch) {
        case 0: return "F";
        case 1: return "B";
        case 2: return "C";
        case 3: return "D";
        case 4: return "E";
        case 5: return "A";
        case 6: return "G";
        case 7: return "H";
        case 8: return "I";
        case 9: return "J";
        default: return "UNKNOWN";
    }
}


static int data1_to_pad(unsigned char data1) {
    size_t i;
    for (i = 0; i < sizeof(pad_maps) / sizeof(pad_maps[0]); i++) {
        if (pad_maps[i].note == data1)
            return pad_maps[i].pad;
    }
    return -1;
}

static volatile uint32_t *fpga_regs = NULL;

static sequencer_state state;
static pthread_mutex_t state_mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t  playback_cond = PTHREAD_COND_INITIALIZER;
static volatile int running = 1;  /* set to 0 on shutdown to exit all threads */

/* Returns a consistent snapshot of sequencer state for the UI to render. */
sequencer_state get_sequencer_state(void) {
    sequencer_state snapshot;
    pthread_mutex_lock(&state_mutex);
    snapshot = state;
    pthread_mutex_unlock(&state_mutex);
    return snapshot;
}

static void init_sequence(void) {
    int i;
    int j;
    for (i = 0; i < NUM_STEPS; i++)
        for (j = 0; j < MAX_TRACKS; j++)
```

```
            state.sequence[i][j] = (cell){0, 0, 50};
    for (i = 0; i < MAX_TRACKS; i++)
        state.tracks[i] = -1;
    state.measure        = 1;
    state.global_bpm     = 120;
    state.cursor_step    = 0;
    state.cursor_track   = 0;
    state.playback_state = 0;
    state.playback_step  = 0;
    state.edit_mode      = NAVIGATION;
}

/* Assigns instrument_id to track_slot, clearing any existing sequence on that track. */
static void assign_instrument(int track_slot, int instrument_id) {
    assert(track_slot >= 0 && track_slot < MAX_TRACKS);
    assert(instrument_id >= 0 && instrument_id < NUM_VOICES);
    state.tracks[track_slot] = instrument_id;
}

/* Clears all steps on track_slot and unassigns its instrument. */
static void clear_track(int track_slot) {
    int i;
    assert(track_slot >= 0 && track_slot < MAX_TRACKS);
    for (i = 0; i < NUM_STEPS; i++)
        state.sequence[i][track_slot] = (cell){0, 0, 50};
    state.tracks[track_slot] = -1;
}

/* Single write to REG[0] allocates one voice and triggers immediate playback. */
static void fpga_play_preview(int instrument_id, int8_t pitch, uint8_t volume) {
    uint32_t pitch_fixed = (uint32_t)(powf(2.0f, (float)pitch / 12.0f) * 1024.0f + 0.5f);
    uint32_t reg;
    if (!fpga_regs) return;
    reg = ((uint32_t)instrument_id & 0xf)
        | ((uint32_t)(volume * 63 / 99) & 0x7f) << 4   /* bits 10:4:  gain (0-127) */
        | (pitch_fixed & 0x1fff) << 11;                  /* bits 23:11: pitch ratio (Q3.10)
*/
    fpga_regs[0] = reg;
}

/* Writes one REG[0] per active track at the current playback step. Called with state_mutex
held. */
static void fpga_play_sequence_step(void) {
    unsigned int abs_step;
    uint32_t pitch_fixed, reg;
    int t;
    if (!fpga_regs) return;
    abs_step = (state.measure - 1) * STEPS_PER_MEASURE + state.playback_step;
    for (t = 0; t < MAX_TRACKS; t++) {
        const cell *c = &state.sequence[abs_step][t];
```

```
        if (!c->active || state.tracks[t] < 0) continue;
        pitch_fixed = (uint32_t)(powf(2.0f, (float)c->pitch / 12.0f) * 1024.0f + 0.5f);
        reg = ((uint32_t)state.tracks[t] & 0xf)
            | ((uint32_t)(c->volume * 63 / 99) & 0x7f) << 4
            | (pitch_fixed & 0x1fff) << 11;
        fpga_regs[0] = reg;
    }
}

static void save_sequence(void) {
    char path[256];
    time_t now = time(NULL);
    struct tm *tm = localtime(&now);
    mkdir("saves", 0755);
    strftime(path, sizeof(path), "saves/%Y-%m-%d_%H-%M-%S.json", tm);
    if (save_sequence_file(path, &state) == 0)
        printf("Saved: %s\n", path);
}

/* Routes pad press/release by edit_mode: navigation or instrument select. */
static void handle_pad_event(unsigned char status, unsigned char data1, unsigned char value)
{
    int ch = status & 0x0f;
    int type = status & 0xf0;
    int pad = data1_to_pad(data1);

    /* EXT SOURCE always arrives on ch 0, data1 0x23; press toggles edit/playback state. */
    if (ch == 0 && data1 == 0x23) {
        if (type == 0x90 && value > 0) {
            state.playback_state ^= 1;
            printf("EXT SOURCE pressed: playback_state=%u\n", state.playback_state);
            if (state.playback_state == 1) {
                /* Starting playback: begin from current cursor position. */
                state.playback_step = state.cursor_step;
                pthread_cond_signal(&playback_cond);
            } else {
                /* playback_thread advances playback_step right after firing each step,
                   so it already points to the next position; land cursor there.
                   measure was kept up to date by playback_thread as it advanced. */
                state.cursor_step = state.playback_step;
                /* cursor_track unchanged */
            }
        } else if (type == 0x80 || (type == 0x90 && value == 0)) {
            state.playback_state ^= 1;
            printf("EXT SOURCE released: playback_state=%u\n", state.playback_state);
            if (state.playback_state == 1) {
                state.playback_step = state.cursor_step;
                pthread_cond_signal(&playback_cond);
            } else {
                state.cursor_step = state.playback_step;
```

```
            }
        }
        return;
    }

    if (state.playback_state == 1) {
        return; /* pad input ignored during playback; playback_thread drives FPGA writes */
    }

    /* Numbered pads 1–16: channel encodes the active mode button (A/F, B/G, C/H). */
    if (pad != -1) {
        static unsigned int prev_edit_mode = NAVIGATION;
        unsigned int new_mode;
        int mode_changed;

        if (ch == 5 || ch == 0)      new_mode = NAVIGATION;
        else if (ch == 1 || ch == 6) new_mode = INSTRUMENT_SELECT;
        else return; /* C/H and beyond: no-op */

        mode_changed       = (new_mode != prev_edit_mode);
        prev_edit_mode     = new_mode;
        state.edit_mode    = new_mode;

        if (type == 0x90 && value > 0) {
            if (mode_changed) return; /* first press after mode switch: UI updates, no action
*/
            printf("Pad %d pressed (edit_mode=%u, data1=0x%02x, velocity=%u)\n",
                   pad, state.edit_mode, data1, value);
            if (state.edit_mode == NAVIGATION) {
                unsigned int abs_step = (state.measure - 1) * STEPS_PER_MEASURE +
state.cursor_step;

                if (pad <= 8) {
                    /* Top 8 buttons: jump to measure */
                    state.measure      = (unsigned int)pad;
                    state.cursor_step  = 0;
                    state.cursor_track = 0;
                } else if (pad == 9) {
                    /* Add / remove note */
                    state.sequence[abs_step][state.cursor_track].active ^= 1;
                } else if (pad == 11) {
                    /* Up: previous track */
                    if (state.cursor_track > 0)
                        state.cursor_track--;
                } else if (pad == 13) {
                    /* Clear track */
                    clear_track((int)state.cursor_track);
                } else if (pad == 14) {
                    /* Left: previous step, wraps across measures */
                    if (state.cursor_step > 0) {
```

```
                    state.cursor_step--;
                } else {
                    state.measure = (state.measure == 1) ? 8 : state.measure - 1;
                    state.cursor_step = STEPS_PER_MEASURE - 1;
                }
            } else if (pad == 15) {
                /* Down: next track */
                if (state.cursor_track < MAX_TRACKS - 1)
                    state.cursor_track++;
            } else if (pad == 16) {
                /* Right: next step, wraps across measures */
                if (state.cursor_step < STEPS_PER_MEASURE - 1) {
                    state.cursor_step++;
                } else {
                    state.measure = (state.measure == 8) ? 1 : state.measure + 1;
                    state.cursor_step = 0;
                }
            } else if (pad == 12) {
                save_sequence();
            }
            /* pad 10: no-op */

            {
                unsigned int cur = (state.measure - 1) * STEPS_PER_MEASURE +
state.cursor_step;
                const cell *c = &state.sequence[cur][state.cursor_track];
                if (c->active && state.tracks[state.cursor_track] >= 0)
                    fpga_play_preview(state.tracks[state.cursor_track], c->pitch,
c->volume);
            }

        } else if (state.edit_mode == INSTRUMENT_SELECT) {
            assign_instrument((int)state.cursor_track, pad - 1);
            fpga_play_preview(pad - 1, 0, 50);
        }
    } else if (type == 0x80 || (type == 0x90 && value == 0)) {
        printf("Pad %d released (edit_mode=%u, data1=0x%02x)\n",
               pad, state.edit_mode, data1);
    }
    return;
}

printf("Unknown pad/button event (status=0x%02x, channel=%s, data1=0x%02x,
value=0x%02x)\n",
       status, channel_name(ch), data1, value);
}

/* Routes CTRL 1 (bpm), CTRL 2 (pitch), CTRL 3 (volume) to global/cell state. */
static void handle_knob_event(unsigned char status, unsigned char controller, unsigned char
value) {
```

```
    int ch = status & 0x0f;

    (void)ch;

    if (controller == 0x10) {
        /* Scale 0-127 → 64-192 */
        state.global_bpm = 64 + (unsigned int)(value * 128 / 127);
        printf("CTRL 1: global_bpm=%u\n", state.global_bpm);
        return;
    }

    if (controller == 0x11 || controller == 0x12) {
        if (state.playback_state == 0) {
            unsigned int abs_step = (state.measure - 1) * STEPS_PER_MEASURE +
state.cursor_step;
            cell *cell = &state.sequence[abs_step][state.cursor_track];
            if (cell->active) {
                if (controller == 0x11) {
                    /* Scale 0-127 → -24 to 24 */
                    cell->pitch = (int8_t)((int)value * 48 / 127 - 24);
                    printf("CTRL 2: measure=%u step=%u track=%u pitch=%d\n",
                           state.measure, state.cursor_step, state.cursor_track,
cell->pitch);
                } else {
                    /* Scale 0-127 → 0-99 */
                    cell->volume = (uint8_t)(value * 99 / 127);
                    printf("CTRL 3: measure=%u step=%u track=%u volume=%u\n",
                           state.measure, state.cursor_step, state.cursor_track,
cell->volume);
                }
            }
        }
        return;
    }

    printf("Unknown knob event (controller=0x%02x, value=%u)\n", controller, value);
}

static void decode_packet(const unsigned char *buf, int transferred) {
    int i;

    if (transferred <= 0 || transferred % 4 != 0) {
        printf("Unexpected packet length: %d\n", transferred);
        return;
    }

    for (i = 0; i < transferred; i += 4) {
        unsigned char cin    = buf[i + 0];
        unsigned char status = buf[i + 1];
        unsigned char data1  = buf[i + 2];
```

```
        unsigned char data2  = buf[i + 3];
        unsigned char type   = status & 0xf0;

        (void)cin;

        if (type == 0x90 || type == 0x80) {
            handle_pad_event(status, data1, data2);
        } else if (type == 0xb0) {
            handle_knob_event(status, data1, data2);
        } else {
            printf("Unhandled MIDI packet (status=0x%02x, data1=0x%02x, data2=0x%02x)\n",
                   status, data1, data2);
        }
    }
}

#define HPS_GPIO1_BASE   0xFF709000u
#define HPS_I2C_CTRL_BIT 19            /* HPS GPIO48 = bit 19 of GPIO1 */
#define WM8731_ADDR      0x1a
#define WM8731_I2C_BUS   "/dev/i2c-0"

static void wm8731_reg(int fd, uint8_t reg, uint16_t data) {
    uint8_t buf[2];
    buf[0] = (reg << 1) | ((data >> 8) & 1);
    buf[1] = data & 0xFF;
    if (write(fd, buf, 2) != 2)
        perror("wm8731_reg");
}

static int wm8731_init(void) {
    static const struct { uint8_t reg; uint16_t data; } seq[] = {
        { 0x0F, 0x000 }, /* software reset */
        { 0x06, 0x000 }, /* power: all blocks on */
        { 0x00, 0x097 },
        { 0x01, 0x097 },
        { 0x02, 0x079 },
        { 0x03, 0x079 },
        { 0x04, 0x010 }, /* analog: DAC select */
        { 0x05, 0x000 },
        { 0x07, 0x002 }, /* interface: I2S, 16-bit, slave */
        { 0x08, 0x001 }, /* sampling: USB mode, 48 kHz */
        { 0x09, 0x001 }, /* activate */
    };
    volatile uint32_t *gpio1;
    int memfd, i2cfd, ret = 0;
    size_t i;

    memfd = open("/dev/mem", O_RDWR | O_SYNC);
    if (memfd < 0) { perror("wm8731_init: /dev/mem"); return -1; }
    gpio1 = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED, memfd, HPS_GPIO1_BASE);
```

```
    close(memfd);
    if (gpio1 == MAP_FAILED) { perror("wm8731_init: mmap GPIO1"); return -1; }

    gpio1[1] |= (1u << HPS_I2C_CTRL_BIT);  /* DDR: set as output */
    gpio1[0] |= (1u << HPS_I2C_CTRL_BIT);  /* drive high: HPS I2C → codec */

    i2cfd = open(WM8731_I2C_BUS, O_RDWR);
    if (i2cfd < 0) { perror("wm8731_init: open i2c"); ret = -1; goto restore; }
    if (ioctl(i2cfd, I2C_SLAVE, WM8731_ADDR) < 0) {
        perror("wm8731_init: I2C_SLAVE"); close(i2cfd); ret = -1; goto restore;
    }
    for (i = 0; i < sizeof(seq) / sizeof(seq[0]); i++)
        wm8731_reg(i2cfd, seq[i].reg, seq[i].data);
    close(i2cfd);

restore:
    gpio1[0] &= ~(1u << HPS_I2C_CTRL_BIT); /* restore: FPGA I2C back in control */
    munmap((void *)gpio1, 0x1000);
    return ret;
}

static int fpga_init(void) {
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd < 0) { perror("open /dev/mem"); return -1; }
    void *base = mmap(NULL, FPGA_SPAN, PROT_READ | PROT_WRITE, MAP_SHARED, fd, FPGA_BASE);
    close(fd);
    if (base == MAP_FAILED) { perror("mmap FPGA"); return -1; }
    fpga_regs = (volatile uint32_t *)((char *)base + FPGA_OFFSET);
    return 0;
}

/* Upload 17 sample-boundary offsets to audio_engine_0 via Avalon address 1.
   Packs (index, offset) into one 32-bit write per entry. Returns 0 on success,
   -1 on file error / bad format. If this fails, the audio engine still works
   off its initial-block defaults; this is just a non-fatal best-effort load. */
static int load_sample_indices(const char *path) {
    FILE *f = fopen(path, "r");
    if (!f) { fprintf(stderr, "load_sample_indices: open %s: ", path); perror(NULL); return
-1; }

    if (!fpga_regs) { fclose(f); return -1; }

    char line[256];
    int idx = 0;
    while (idx < 17 && fgets(line, sizeof(line), f)) {
        char *p = line;
        while (*p && isspace((unsigned char)*p)) p++;
        if (*p == '\0' || *p == '#') continue;
        char *end;
        unsigned long val = strtoul(p, &end, 10);
```

```
        if (end == p) {
            fprintf(stderr, "load_sample_indices: %s: parse error near '%s'\n", path, p);
            fclose(f);
            return -1;
        }
        if (val > 0x07FFFFFFu) {  /* writedata[31:5] = 27-bit field */
            fprintf(stderr, "load_sample_indices: offset %lu exceeds 27-bit field\n", val);
            fclose(f);
            return -1;
        }
        /* Avalon address 1: writedata[4:0] = idx, writedata[31:5] = offset. */
        fpga_regs[1] = ((uint32_t)val << 5) | (uint32_t)(idx & 0x1f);
        idx++;
    }
    fclose(f);
    if (idx != 17) {
        fprintf(stderr, "load_sample_indices: %s has %d entries, expected 17\n", path, idx);
        return -1;
    }
    printf("Uploaded 17 sample indices from %s\n", path);
    return 0;
}

/* Advances playback one step per BPM-derived interval, writing each step to the FPGA. */
static void *playback_thread_func(void *arg) {
    struct timespec ts;
    long step_ns;
    unsigned int bpm;

    (void)arg;

    pthread_mutex_lock(&state_mutex);
    while (running) {
        while (state.playback_state == 0 && running)
            pthread_cond_wait(&playback_cond, &state_mutex);

        if (!running)
            break;

        fpga_play_sequence_step();
        bpm = state.global_bpm;

        state.playback_step++;
        if (state.playback_step >= STEPS_PER_MEASURE) {
            state.playback_step = 0;
            state.measure = (state.measure % 8) + 1;
        }

        pthread_mutex_unlock(&state_mutex);
```

```
        /* 15,000,000,000 ns / BPM = ns per 16th note (4 steps per beat) */
        step_ns        = 15000000000L / (long)bpm;
        ts.tv_sec      = step_ns / 1000000000L;
        ts.tv_nsec     = step_ns % 1000000000L;
        nanosleep(&ts, NULL);

        pthread_mutex_lock(&state_mutex);
    }
    pthread_mutex_unlock(&state_mutex);
    return NULL;
}

static unsigned char *gui_load_file(const char *path, size_t size) {
    FILE *f = fopen(path, "rb");
    if (!f) { fprintf(stderr, "open %s: ", path); perror(NULL); return NULL; }
    unsigned char *buf = malloc(size);
    if (!buf || fread(buf, 1, size, f) != size) {
        free(buf); fclose(f); return NULL;
    }
    fclose(f);
    return buf;
}

static int gui_upload(int fd, unsigned int offset,
                      const unsigned char *buf, unsigned int len) {
    ui_write_arg_t arg = { .offset = offset, .length = len, .buf = buf };
    if (ioctl(fd, UI_WRITE, &arg) != 0) {
        fprintf(stderr, "ioctl UI_WRITE @0x%x len %u: ", offset, len);
        perror(NULL);
        return -1;
    }
    return 0;
}

/* Renders the full GUI at ~60 Hz by snapshotting sequencer state, converting to
   a tilemap, and uploading it to the FPGA display peripheral via /dev/ui. */
static void *gui_thread_func(void *arg) {
    char path[256];
    unsigned char tilemap[UI_TILEMAP_SIZE];
    unsigned char *tileset, *palette;

    (void)arg;

    int fd = open(UI_DEV, O_RDWR);
    if (fd < 0) { perror("open " UI_DEV); return NULL; }

    snprintf(path, sizeof(path), "%s/tileset.bin", UI_DIR);
    tileset = gui_load_file(path, UI_TILESET_SIZE);
    if (!tileset) { close(fd); return NULL; }
```

```
    snprintf(path, sizeof(path), "%s/palette.bin", UI_DIR);
    palette = gui_load_file(path, UI_PALETTE_SIZE);
    if (!palette) { free(tileset); close(fd); return NULL; }

    gui_upload(fd, UI_TILESET_OFFSET, tileset, UI_TILESET_SIZE);
    gui_upload(fd, UI_PALETTE_OFFSET, palette, UI_PALETTE_SIZE);
    free(tileset);
    free(palette);

    while (running) {
        sequencer_state s = get_sequencer_state();
        state_to_tiles(&s, tilemap);
        gui_upload(fd, UI_TILEMAP_OFFSET, tilemap, UI_TILEMAP_SIZE);
        usleep(GUI_INTERVAL_US);
    }

    close(fd);
    return NULL;
}

/* drum.c owns all internal sequencer state (sequencer_state). In the main thread, we open
the
   SP-404 MKII via libusb, then loop reading USB-MIDI bulk packets and dispatching pad/knob
events to
   update state. In the main thread, we spawn 2 additional threads that run concurrently:
     - gui_thread: pulls state via get_sequencer_state() and renders to the FPGA display at
~60 Hz.
     - playback_thread: wakes on playback_cond when playback starts, then fires one step per
       BPM-derived interval by writing track registers to the FPGA and advancing
playback_step. */
int main(int argc, char *argv[]) {
    libusb_context *ctx = NULL;
    libusb_device_handle *h = NULL;
    pthread_t gui_thread;
    pthread_t playback_thread;
    int r;

    init_sequence();

    if (argc > 1 && load_sequence_file(argv[1], &state) != 0)
        return 1;

    if (wm8731_init() != 0)
        fprintf(stderr, "Warning: WM8731 init failed; audio may be silent\n");

    if (fpga_init() != 0) {
        fprintf(stderr, "Warning: FPGA init failed; playback writes disabled\n");
    } else if (load_sample_indices("../media/samples.txt") != 0) {
        fprintf(stderr, "Warning: sample-index upload failed; "
                        "audio engine will use its hardcoded defaults\n");
```

```
}

r = libusb_init(&ctx);
if (r < 0) {
    fprintf(stderr, "libusb_init failed: %s\n", libusb_error_name(r));
    return 1;
}

h = libusb_open_device_with_vid_pid(ctx, VID, PID);
if (h == NULL) {
    fprintf(stderr, "Could not open Roland device %04x:%04x\n", VID, PID);
    libusb_exit(ctx);
    return 1;
}

libusb_set_auto_detach_kernel_driver(h, 1);

r = libusb_claim_interface(h, IFACE);
if (r < 0) {
    fprintf(stderr, "Claim interface %d failed: %s\n", IFACE, libusb_error_name(r));
    libusb_close(h);
    libusb_exit(ctx);
    return 1;
}

pthread_create(&gui_thread, NULL, gui_thread_func, NULL);
pthread_create(&playback_thread, NULL, playback_thread_func, NULL);

printf("Listening for SP-404 USB events...\n");
fflush(stdout);

while (1) {
    unsigned char buf[64];
    int transferred = 0;

    r = libusb_bulk_transfer(h, EP_IN, buf, sizeof(buf), &transferred, 0);
    if (r == 0) {
        pthread_mutex_lock(&state_mutex);
        decode_packet(buf, transferred);
        pthread_mutex_unlock(&state_mutex);
        fflush(stdout);
    } else {
        fprintf(stderr, "bulk_transfer failed: %s\n", libusb_error_name(r));
        break;
    }
}

pthread_mutex_lock(&state_mutex);
running = 0;
pthread_cond_signal(&playback_cond);
```

```
    pthread_mutex_unlock(&state_mutex);

    pthread_join(gui_thread, NULL);
    pthread_join(playback_thread, NULL);

    if (fpga_regs)
        munmap((void *)fpga_regs, FPGA_SPAN);

    libusb_release_interface(h, IFACE);
    libusb_close(h);
    libusb_exit(ctx);
    return 0;
}
```

drum.h, Aaron Zhu

None

```
#ifndef DRUM_H
#define DRUM_H

#include <stdint.h>

#define NUM_STEPS         128
#define NUM_VOICES        16
#define MAX_TRACKS        8
#define STEPS_PER_MEASURE 16

#define NAVIGATION        0
#define INSTRUMENT_SELECT 1

typedef struct {
    uint8_t active;  /* 0 or 1, default 0 */
    int8_t  pitch;   /* -24 to 24 semitones, default 0 */
    uint8_t volume;  /* 0-99, default 50 */
} cell;

/* Look up an instrument's display name by id.  The single source of
 * truth for instrument order, names, and palette tiles lives in
 * state_to_tiles.c (instrument_defs[]); reorder/edit that table to
 * change the palette layout (and the name returned here). */
const char *instrument_name(int instrument_id);

/* Complete sequencer state. The UI calls get_sequencer_state() to pull a snapshot
   and render; drum.c never pushes to the UI. */
typedef struct {
    cell       sequence[NUM_STEPS][MAX_TRACKS];   /* [step][track] */
    int          tracks[MAX_TRACKS];              /* instrument_id per slot, or -1 if
unassigned */
    unsigned int measure;                         /* [1, 8]: current measure */
```

```
    unsigned int global_bpm;                    /* [64, 192] */
    unsigned int cursor_step;                   /* [0, 15]: step offset within current
measure */
    unsigned int cursor_track;                  /* [0, 7]: current track slot */
    unsigned int playback_state;                /* 0 = editing, 1 = playing back */
    unsigned int playback_step;                 /* [0, 15]: step within current measure
during playback */
    unsigned int edit_mode;                     /* NAVIGATION, INSTRUMENT_SELECT */
} sequencer_state;

sequencer_state get_sequencer_state(void);

#endif /* DRUM_H */
```

drum.sh (compiles and starts main drum machine program, optionally takes an argument to read in existing program states as .json files, see folder sequencer_states), Aaron Zhu

```
None
#!/bin/sh
# Build the ui kernel module + userspace tools, load the module,
# verify /dev/ui exists, then run the drum machine.

set -e

rmmod ui 2>/dev/null || true   # ignore "module not loaded"
make
insmod ui.ko
ls -l /dev/ui
./drum "$@"
```

Makefile, Jordan Lin, Aaron Zhu

```
None
ifneq (${KERNELRELEASE},)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
        obj-m := drum_hello.o ui.o

else

# We are being compiled as a module: use the Kernel build system

    KERNEL_SOURCE := /usr/src/linux-headers-$(shell uname -r)
        PWD := $(shell pwd)

CC = gcc
```

```
CFLAGS = -Wall -Wextra -O2 -I$(HOME)/.local/include
LDFLAGS_USB = -lusb-1.0 -lpthread -lm

default: module hello drum ui_demo

module:
    ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules

hello: hello.c drum_hello.h
    $(CC) $(CFLAGS) -o hello hello.c

drum: drum.c drum.h seq_load.c seq_load.h state_to_tiles.c state_to_tiles.h
    $(CC) $(CFLAGS) -o drum drum.c seq_load.c state_to_tiles.c $(LDFLAGS_USB)

ui_demo: ui_demo.c ui.h
    $(CC) $(CFLAGS) -o ui_demo ui_demo.c

clean:
    ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
    ${RM} hello drum ui_demo

.PHONY: default module clean

endif
```

seq_load.c, Aaron Zhu

None

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "seq_load.h"

typedef struct {
    int     index;
    int     pitch;   /* -24 to 24 */
    int     volume;  /* 0-99 */
} step_entry;

static const char *skip_ws(const char *p) {
    while (*p == ' ' || *p == '\t' || *p == '\n' || *p == '\r') p++;
    return p;
}

static const char *expect_char(const char *p, char c) {
    p = skip_ws(p);
    return (*p == c) ? p + 1 : NULL;
}
```

```
static const char *parse_jstring(const char *p, char *out, int max) {
    int i = 0;
    p = skip_ws(p);
    if (*p++ != '"') return NULL;
    while (*p && *p != '"' && i < max - 1) out[i++] = *p++;
    out[i] = '\0';
    return (*p == '"') ? p + 1 : NULL;
}

static const char *parse_jint(const char *p, int *out) {
    char *end;
    p = skip_ws(p);
    *out = (int)strtol(p, &end, 10);
    return (end == p) ? NULL : end;
}

/* Skip any JSON value (string, number, object, array, true/false/null). */
static const char *skip_jvalue(const char *p) {
    int depth;
    p = skip_ws(p);
    if (*p == '"') {
        p++;
        while (*p && *p != '"') { if (*p == '\\' && *(p+1)) p++; p++; }
        return *p ? p + 1 : NULL;
    }
    if (*p == '{' || *p == '[') {
        depth = 1; p++;
        while (*p && depth > 0) {
            if (*p == '"') {
                p++;
                while (*p && *p != '"') { if (*p == '\\' && *(p+1)) p++; p++; }
                if (*p) p++;
            } else if (*p == '{' || *p == '[') { depth++; p++;
            } else if (*p == '}' || *p == ']') { depth--; p++;
            } else { p++; }
        }
        return depth == 0 ? p : NULL;
    }
    while (*p && *p != ',' && *p != '}' && *p != ']' &&
           *p != ' ' && *p != '\t' && *p != '\n' && *p != '\r') p++;
    return p;
}

/* Parse one step: plain integer index or { "index": N, "pitch": P, "volume": V }. */
static const char *parse_step(const char *p, step_entry *out) {
    char key[32];
    out->pitch  = 0;
    out->volume = 50;
    out->index  = -1;
```

```
    p = skip_ws(p);
    if (*p == '{') {
        p++;
        while (*(p = skip_ws(p)) != '}' && *p) {
            if (*p == ',') { p++; continue; }
            if (!(p = parse_jstring(p, key, sizeof(key)))) return NULL;
            if (!(p = expect_char(p, ':'))) return NULL;
            int val;
            if (!(p = parse_jint(p, &val))) return NULL;
            if      (strcmp(key, "index")  == 0) out->index  = val;
            else if (strcmp(key, "pitch")  == 0) out->pitch  = val;
            else if (strcmp(key, "volume") == 0) out->volume = val;
        }
        return expect_char(p, '}');
    }
    if (!(p = parse_jint(p, &out->index))) return NULL;
    return p;
}

int load_sequence_file(const char *path, sequencer_state *state) {
    FILE *f;
    long size;
    char *buf;
    const char *p;
    char key[64];
    int val;

    f = fopen(path, "r");
    if (!f) { fprintf(stderr, "%s: ", path); perror(NULL); return -1; }
    fseek(f, 0, SEEK_END);
    size = ftell(f);
    rewind(f);
    buf = malloc(size + 1);
    if (!buf) { fclose(f); return -1; }
    if ((long)fread(buf, 1, size, f) != size) { free(buf); fclose(f); return -1; }
    buf[size] = '\0';
    fclose(f);

    p = buf;
    if (!(p = expect_char(p, '{'))) goto fail;

    while (*(p = skip_ws(p)) != '}' && *p) {
        if (*p == ',') { p++; continue; }
        if (!(p = parse_jstring(p, key, sizeof(key)))) goto fail;
        if (!(p = expect_char(p, ':'))) goto fail;

        if (strcmp(key, "bpm") == 0) {
            if (!(p = parse_jint(p, &val))) goto fail;
            if (val >= 64 && val <= 192) state->global_bpm = (unsigned int)val;
```

```
} else if (strcmp(key, "tracks") == 0) {
    if (!(p = expect_char(p, '['))) goto fail;

    while (*(p = skip_ws(p)) != ']' && *p) {
        if (*p == ',') { p++; continue; }
        if (!(p = expect_char(p, '{'))) goto fail;

        int slot = -1, instrument = -1;
        step_entry steps[NUM_STEPS];
        int nsteps = 0;

        while (*(p = skip_ws(p)) != '}' && *p) {
            if (*p == ',') { p++; continue; }
            if (!(p = parse_jstring(p, key, sizeof(key)))) goto fail;
            if (!(p = expect_char(p, ':'))) goto fail;

            if (strcmp(key, "slot") == 0) {
                if (!(p = parse_jint(p, &slot))) goto fail;
            } else if (strcmp(key, "instrument") == 0) {
                if (!(p = parse_jint(p, &instrument))) goto fail;
            } else if (strcmp(key, "measures") == 0) {
                if (!(p = expect_char(p, '['))) goto fail;

                while (*(p = skip_ws(p)) != ']' && *p) {
                    if (*p == ',') { p++; continue; }
                    if (!(p = expect_char(p, '{'))) goto fail;

                    int measure_num = -1;
                    step_entry msteps[STEPS_PER_MEASURE];
                    int nmsteps = 0;

                    while (*(p = skip_ws(p)) != '}' && *p) {
                        if (*p == ',') { p++; continue; }
                        if (!(p = parse_jstring(p, key, sizeof(key)))) goto fail;
                        if (!(p = expect_char(p, ':'))) goto fail;

                        if (strcmp(key, "measure") == 0) {
                            if (!(p = parse_jint(p, &measure_num))) goto fail;
                        } else if (strcmp(key, "steps") == 0) {
                            if (!(p = expect_char(p, '['))) goto fail;
                            while (*(p = skip_ws(p)) != ']' && *p) {
                                if (*p == ',') { p++; continue; }
                                step_entry se;
                                if (!(p = parse_step(p, &se))) goto fail;
                                if (se.index >= 0 && se.index < STEPS_PER_MEASURE
                                        && nmsteps < STEPS_PER_MEASURE)
                                    msteps[nmsteps++] = se;
                            }
                            if (!(p = expect_char(p, ']'))) goto fail;
                        } else {
```

```
                                if (!(p = skip_jvalue(p))) goto fail;
                            }
                        }
                        if (!(p = expect_char(p, '}'))) goto fail;

                        if (measure_num >= 1 &&
                                measure_num <= NUM_STEPS / STEPS_PER_MEASURE) {
                            int base = (measure_num - 1) * STEPS_PER_MEASURE;
                            for (int i = 0; i < nmsteps && nsteps < NUM_STEPS; i++) {
                                steps[nsteps].index  = base + msteps[i].index;
                                steps[nsteps].pitch  = msteps[i].pitch;
                                steps[nsteps].volume = msteps[i].volume;
                                nsteps++;
                            }
                        }
                    }
                    if (!(p = expect_char(p, ']'))) goto fail;
                } else {
                    if (!(p = skip_jvalue(p))) goto fail;
                }
            }
            if (!(p = expect_char(p, '}'))) goto fail;

            if (slot >= 0 && slot < MAX_TRACKS &&
                instrument >= 0 && instrument < NUM_VOICES) {
                state->tracks[slot] = instrument;
                for (int i = 0; i < nsteps; i++) {
                    int idx = steps[i].index;
                    state->sequence[idx][slot].active = 1;
                    state->sequence[idx][slot].pitch  = (int8_t) (steps[i].pitch  < -24 ?
-24 : steps[i].pitch  > 24 ? 24 : steps[i].pitch);
                    state->sequence[idx][slot].volume = (uint8_t)(steps[i].volume <   0 ?
0 : steps[i].volume > 99 ? 99 : steps[i].volume);
                }
            }
        }
        if (!(p = expect_char(p, ']'))) goto fail;
    } else {
        if (!(p = skip_jvalue(p))) goto fail;
    }
}

free(buf);
printf("Loaded sequence: %s\n", path);
return 0;

fail:
    fprintf(stderr, "Parse error in sequence file: %s\n", path);
    free(buf);
    return -1;
```

```
}

int save_sequence_file(const char *path, const sequencer_state *state) {
    int num_measures = NUM_STEPS / STEPS_PER_MEASURE;
    int t, m, s, first_track, first_measure, first_step;
    FILE *f = fopen(path, "w");
    if (!f) { fprintf(stderr, "%s: ", path); perror(NULL); return -1; }

    fprintf(f, "{\n  \"bpm\": %u,\n  \"tracks\": [\n", state->global_bpm);

    first_track = 1;
    for (t = 0; t < MAX_TRACKS; t++) {
        if (state->tracks[t] < 0) continue;
        if (!first_track) fprintf(f, ",\n");
        first_track = 0;

        fprintf(f, "    {\n      \"slot\": %d, \"instrument\": %d,\n      \"measures\": [\n",
                t, state->tracks[t]);

        first_measure = 1;
        for (m = 0; m < num_measures; m++) {
            if (!first_measure) fprintf(f, ",\n");
            first_measure = 0;
            fprintf(f, "        { \"measure\": %d, \"steps\": [", m + 1);

            first_step = 1;
            for (s = 0; s < STEPS_PER_MEASURE; s++) {
                const cell *c = &state->sequence[m * STEPS_PER_MEASURE + s][t];
                if (!c->active) continue;
                if (!first_step) fprintf(f, ", ");
                first_step = 0;
                fprintf(f, "{ \"index\": %d, \"pitch\": %d, \"volume\": %u }",
                        s, (int)c->pitch, (unsigned)c->volume);
            }
            fprintf(f, "] }");
        }
        fprintf(f, "\n      ]\n    }");
    }

    fprintf(f, "\n  ]\n}\n");
    fclose(f);
    return 0;
}
```

seq_load.h, Aaron Zhu

```
None
#ifndef SEQ_LOAD_H
#define SEQ_LOAD_H
```

```
#include "drum.h"

/*
 * Parse a JSON sequence file into *state. Format:
 *   {
 *     "bpm": 120,
 *     "tracks": [
 *       {
 *         "slot": 0, "instrument": 8,
 *         "measures": [
 *           {
 *             "measure": 1,
 *             "steps": [
 *               0, 4, 8,
 *               { "index": 12, "pitch": -5, "volume": 80 }
 *             ]
 *           },
 *           {
 *             "measure": 2,
 *             "steps": [ 0, 8 ]
 *           }
 *         ]
 *       }
 *     ]
 *   }
 * measure: [1, 8], step index: [0, 15] within that measure.
 * A plain integer step uses defaults (pitch=0, volume=50).
 * pitch: [-24, 24], volume: [0, 99].
 * Returns 0 on success, -1 on error.
 */
int load_sequence_file(const char *path, sequencer_state *state);
int save_sequence_file(const char *path, const sequencer_state *state);

#endif /* SEQ_LOAD_H */
```

state_to_tiles.c, Jordan Lin

None

```
#include <string.h>
#include "state_to_tiles.h"
#include "ui.h"

#define TILEMAP_STRIDE  32   /* Matches hardware tilemap addressing */

static inline void put(unsigned char *tilemap, int row, int col, tile_id_t t)
{
    tilemap[row * TILEMAP_STRIDE + col] = (unsigned char)t;
}
```

```
/* Shared 2-tile-pair struct, used for the measure-select buttons below.
 * (instrument_def_t lives in state_to_tiles.h so other modules can read
 * the table.) */
typedef struct
{
    tile_id_t left;
    tile_id_t right;
} tile_pair_t;

/* Definition of the table declared in state_to_tiles.h.  Reorder rows
 * here to reorder palette positions, track-row labels, and audio-voice
 * mapping in lockstep — the index is instrument_id throughout the
 * project. */
const instrument_def_t instrument_defs[NUM_VOICES] =
{
    [0]  = { "Ride1",       TILE_INS_RIDE_L,     TILE_INS_RIDE1_R,     TILE_INS_RIDE_SEL_L,
TILE_INS_RIDE1_SEL_R,     TILE_INS_RIDE_NB_L,     TILE_INS_RIDE1_NB_R     },
    [1]  = { "Ride2",       TILE_INS_RIDE_L,     TILE_INS_RIDE2_R,     TILE_INS_RIDE_SEL_L,
TILE_INS_RIDE2_SEL_R,     TILE_INS_RIDE_NB_L,     TILE_INS_RIDE2_NB_R     },
    [2]  = { "Crash1",      TILE_INS_CRASH_L,    TILE_INS_CRASH1_R,    TILE_INS_CRASH_SEL_L,
TILE_INS_CRASH1_SEL_R,    TILE_INS_CRASH_NB_L,    TILE_INS_CRASH1_NB_R    },
    [3]  = { "Crash2",      TILE_INS_CRASH_L,    TILE_INS_CRASH2_R,    TILE_INS_CRASH_SEL_L,
TILE_INS_CRASH2_SEL_R,    TILE_INS_CRASH_NB_L,    TILE_INS_CRASH2_NB_R    },
    [4]  = { "HiTom",       TILE_INS_HITOM_L,    TILE_INS_HITOM_R,     TILE_INS_HITOM_SEL_L,
TILE_INS_HITOM_SEL_R,     TILE_INS_HITOM_NB_L,    TILE_INS_HITOM_NB_R     },
    [5]  = { "OpenHiHat",   TILE_INS_OPHAT_L,    TILE_INS_OPHAT_R,     TILE_INS_OPHAT_SEL_L,
TILE_INS_OPHAT_SEL_R,     TILE_INS_OPHAT_NB_L,    TILE_INS_OPHAT_NB_R     },
    [6]  = { "MidTom",      TILE_INS_MIDTOM_L,   TILE_INS_TOM_R,       TILE_INS_MIDTOM_SEL_L,
TILE_INS_TOM_SEL_R,       TILE_INS_MIDTOM_NB_L,   TILE_INS_TOM_NB_R       },
    [7]  = { "LowTom",      TILE_INS_LOWTOM_L,   TILE_INS_TOM_R,       TILE_INS_LOWTOM_SEL_L,
TILE_INS_TOM_SEL_R,       TILE_INS_LOWTOM_NB_L,   TILE_INS_TOM_NB_R       },
    [8]  = { "Snare1",      TILE_INS_SNARE_L,    TILE_INS_SNARE1_R,    TILE_INS_SNARE_SEL_L,
TILE_INS_SNARE1_SEL_R,    TILE_INS_SNARE_NB_L,    TILE_INS_SNARE1_NB_R    },
    [9]  = { "ClosedHiHat", TILE_INS_CLHAT_L,    TILE_INS_CLHAT_R,     TILE_INS_CLHAT_SEL_L,
TILE_INS_CLHAT_SEL_R,     TILE_INS_CLHAT_NB_L,    TILE_INS_CLHAT_NB_R     },
    [10] = { "Snare2",      TILE_INS_SNARE_L,    TILE_INS_SNARE2_R,    TILE_INS_SNARE_SEL_L,
TILE_INS_SNARE2_SEL_R,    TILE_INS_SNARE_NB_L,    TILE_INS_SNARE2_NB_R    },
    [11] = { "Bass1",       TILE_INS_BASS_L,     TILE_INS_BASS1_R,     TILE_INS_BASS_SEL_L,
TILE_INS_BASS1_SEL_R,     TILE_INS_BASS_NB_L,     TILE_INS_BASS1_NB_R     },
    [12] = { "Clap",        TILE_INS_CLAP_L,     TILE_INS_CLAP_R,      TILE_INS_CLAP_SEL_L,
TILE_INS_CLAP_SEL_R,      TILE_INS_CLAP_NB_L,     TILE_INS_CLAP_NB_R      },
    [13] = { "Snare3",      TILE_INS_SNARE_L,    TILE_INS_SNARE3_R,    TILE_INS_SNARE_SEL_L,
TILE_INS_SNARE3_SEL_R,    TILE_INS_SNARE_NB_L,    TILE_INS_SNARE3_NB_R    },
    [14] = { "RimShot",     TILE_INS_RIMSHOT_L,  TILE_INS_RIMSHOT_R,
TILE_INS_RIMSHOT_SEL_L,  TILE_INS_RIMSHOT_SEL_R,   TILE_INS_RIMSHOT_NB_L,
TILE_INS_RIMSHOT_NB_R   },
    [15] = { "Bass2",       TILE_INS_BASS_L,     TILE_INS_BASS2_R,     TILE_INS_BASS_SEL_L,
TILE_INS_BASS2_SEL_R,     TILE_INS_BASS_NB_L,     TILE_INS_BASS2_NB_R     },
};
```

```
const char *instrument_name(int instrument_id)
{
    if (instrument_id < 0 || instrument_id >= NUM_VOICES)
        return "?";
    return instrument_defs[instrument_id].name;
}

/* Digit-to-tile lookup. TILE_NUM_X enum values are not sequential
 * (numbers occupy the unused tail of rows 1-4 in the tileset), so
 * indexing by digit is the cleanest option. */
static const tile_id_t num_tiles[10] = {
    TILE_NUM_0, TILE_NUM_1, TILE_NUM_2, TILE_NUM_3, TILE_NUM_4,
    TILE_NUM_5, TILE_NUM_6, TILE_NUM_7, TILE_NUM_8, TILE_NUM_9,
};

/* Measure-select button tile pairs (used in NAV mode), indexed 0-7 for
 * measures 1-8, _sel variant is shown for the currently-selected
 * measure (state->measure) */
static const tile_pair_t measure_tiles[8] =
{
    [0] = { TILE_DNUM_1_L, TILE_DNUM_1_R },
    [1] = { TILE_DNUM_2_L, TILE_DNUM_2_R },
    [2] = { TILE_DNUM_3_L, TILE_DNUM_3_R },
    [3] = { TILE_DNUM_4_L, TILE_DNUM_4_R },
    [4] = { TILE_DNUM_5_L, TILE_DNUM_5_R },
    [5] = { TILE_DNUM_6_L, TILE_DNUM_6_R },
    [6] = { TILE_DNUM_7_L, TILE_DNUM_7_R },
    [7] = { TILE_DNUM_8_L, TILE_DNUM_8_R },
};

static const tile_pair_t measure_sel_tiles[8] =
{
    [0] = { TILE_DNUM_1_SEL_L, TILE_DNUM_1_SEL_R },
    [1] = { TILE_DNUM_2_SEL_L, TILE_DNUM_2_SEL_R },
    [2] = { TILE_DNUM_3_SEL_L, TILE_DNUM_3_SEL_R },
    [3] = { TILE_DNUM_4_SEL_L, TILE_DNUM_4_SEL_R },
    [4] = { TILE_DNUM_5_SEL_L, TILE_DNUM_5_SEL_R },
    [5] = { TILE_DNUM_6_SEL_L, TILE_DNUM_6_SEL_R },
    [6] = { TILE_DNUM_7_SEL_L, TILE_DNUM_7_SEL_R },
    [7] = { TILE_DNUM_8_SEL_L, TILE_DNUM_8_SEL_R },
};

void state_to_tiles(const sequencer_state *state, unsigned char *buf)
{
    memset(buf, TILE_EMPTY, UI_TILEMAP_SIZE);

    /* STATIC: UI BACKGROUND, does not depend on state */

    /* Row 0: Beat number labels (labels 0, 4, 8, and 12 can be covered by the
```

```
     * playback bar. We draw it here and let the dynamic layer overwrite). */
    put(buf, 0,  3, TILE_BEAT_0);
    put(buf, 0,  7, TILE_BEAT_4);
    put(buf, 0, 11, TILE_BEAT_8);
    put(buf, 0, 15, TILE_BEAT_12);
    put(buf, 0, 19, TILE_BEAT_16);

    /* Rows 1-8: Empty step grid, step 3, 7, 11 use TILE_GRID_4 (carries an extra right line
     * marking 4-step group boundaries), step 15 uses plain TILE_GRID, the right border comes
from
     * TILE_GRID_R_EDGE at col 19 */
    for (int track = 0; track < MAX_TRACKS; track++)
    {
        int row = track + 1;
        for (int step = 0; step < STEPS_PER_MEASURE; step++)
        {
            int col = 3 + step;
            tile_id_t t = ((step % 4) == 3 && step != 15) ? TILE_GRID_4 : TILE_GRID;
            put(buf, row, col, t);
        }
        put(buf, row, 19, TILE_GRID_R_EDGE);
    }

    /* Row 9: Bottom border of the grid (parts may be overwritten by the playback-bar bottom)
*/
    for (int col = 3; col <= 18; col++) put(buf, 9, col, TILE_GRID_B_EDGE);
    put(buf, 9, 19, TILE_GRID_BR_CORNER);  // Bottom-right of grid, single top-left pixel

    /* Rows 10-13 cols 3-10: empty 4x4 palette frame (each "cell" is 2 tiles wide, TILE_GRID
on
     * the left, TILE_GRID_B_EDGE on the right), mode-specific code later overwrites
individual
     * cells with buttons, and cells left untouched stay looking empty */
    for (int row = 10; row <= 13; row++)
    {
        for (int cell = 0; cell < 4; cell++)
        {
            int col = 3 + cell * 2;
            put(buf, row, col,     TILE_GRID);
            put(buf, row, col + 1, TILE_GRID_B_EDGE);
        }
    }

    /* Col 11: Right border of palette */
    for (int row = 10; row <= 13; row++) put(buf, row, 11, TILE_GRID_R_EDGE);

    /* Row 14: Bottom border of the palette */
    for (int col = 3; col <= 10; col++) put(buf, 14, col, TILE_GRID_B_EDGE);
    put(buf, 14, 11, TILE_GRID_BR_CORNER);  // Single top-left pixel for bottom-right
```

```
    /* Cols 12-13: Value labels (digits go in cols 13-15, right-justified) */
    put(buf, 10, 12, TILE_VAL_MEASURE_L);  put(buf, 10, 13, TILE_VAL_MEASURE_R);
    put(buf, 11, 12, TILE_VAL_BPM);
    put(buf, 12, 12, TILE_VAL_PITCH_L);  /* No TILE_VAL_PITCH_PLUS_R because it depends on
+/- of pitch */
    put(buf, 13, 12, TILE_VAL_VOLUME_L);   put(buf, 13, 13, TILE_VAL_VOLUME_R);

    /* Boots N' Cats logo :) at rows 13-14 cols 18-19 (2x2, always shown) */
    put(buf, 13, 18, TILE_BOOTSNCATS_TL);  put(buf, 13, 19, TILE_BOOTSNCATS_TR);
    put(buf, 14, 18, TILE_BOOTSNCATS_BL);  put(buf, 14, 19, TILE_BOOTSNCATS_BR);

    /* DYNAMIC: UI ELEMENTS, depends on state */

    /* For each track t, if state->tracks[t] != -1, place that instrument's
     * NB (no-border) instrument label in cols 1-2 of row t+1 */
    for (int t = 0; t < MAX_TRACKS; ++t)
    {
        int id = state->tracks[t];
        if (id == -1) continue;
        put(buf, t + 1, 1, instrument_defs[id].nb_left);
        put(buf, t + 1, 2, instrument_defs[id].nb_right);
    }

    /* For each (track, step) where state->sequence[base+step][track].active,
     * overwrite the empty grid cell with a note tile. Use TILE_GRID_NOTE_4
     * on 4-boundary steps (3, 7, 11) to keep the extra right line */
    unsigned int base = (state->measure - 1) * STEPS_PER_MEASURE;
    for (int track = 0; track < MAX_TRACKS; track++)
    {
        for (int step = 0; step < STEPS_PER_MEASURE; step++)
        {
            if (!state->sequence[base + step][track].active) continue;
            tile_id_t t = ((step % 4) == 3 && step != 15) ? TILE_GRID_NOTE_4 :
TILE_GRID_NOTE;
            put(buf, track + 1, 3 + step, t);
        }
    }

    /* Cursor's (x, y) on the grid is (state->cursor_step, state->cursor_track),
     * read what's already there in buf and replace with the matching SEL variant:
     *   TILE_GRID        -> TILE_GRID_SEL
     *   TILE_GRID_4      -> TILE_GRID_SEL_4
     *   TILE_GRID_NOTE   -> TILE_GRID_NOTE_SEL
     *   TILE_GRID_NOTE_4 -> TILE_GRID_NOTE_SEL_4
     * Only drawn when not playing back, during playback there's no editing cursor */
    if (state->playback_state == 0)
    {
        int cur_row = state->cursor_track + 1;
        int cur_col = 3 + state->cursor_step;
        switch (buf[cur_row * TILEMAP_STRIDE + cur_col])
```

```
        {
            case TILE_GRID:        put(buf, cur_row, cur_col, TILE_GRID_SEL);        break;
            case TILE_GRID_4:      put(buf, cur_row, cur_col, TILE_GRID_SEL_4);      break;
            case TILE_GRID_NOTE:   put(buf, cur_row, cur_col, TILE_GRID_NOTE_SEL);   break;
            case TILE_GRID_NOTE_4: put(buf, cur_row, cur_col, TILE_GRID_NOTE_SEL_4); break;
        }
    }

    /* Playback bar (only when state->playback_state == 1), bar is at
     * cols (2 + step, 3 + step) on rows 0 and 9. Taking care of special cases:
     *   Row 0 left  = TILE_BEAT_X_PH if 2+step lands on a beat-number col
     *                 (3, 7, 11, 15) else TILE_PH_T_LEFT
     *   Row 0 right = TILE_PH_T_RIGHT (covers whatever's underneath, no need for special
case)
     *   Row 9 left  = TILE_PH_BL if step == 0 (col 2 is before the
     *                 bottom border line) else TILE_PH_B_LEFT
     *   Row 9 right = TILE_PH_B_RIGHT */
    if (state->playback_state == 1)
    {
        unsigned int step = state->playback_step;
        int left_col  = 2 + step;
        int right_col = 3 + step;

        tile_id_t top_left;
        switch (left_col) {
            case 3:  top_left = TILE_BEAT_0_PH;  break;
            case 7:  top_left = TILE_BEAT_4_PH;  break;
            case 11: top_left = TILE_BEAT_8_PH;  break;
            case 15: top_left = TILE_BEAT_12_PH; break;
            default: top_left = TILE_PH_T_LEFT;  break;
        }
        put(buf, 0, left_col,  top_left);
        put(buf, 0, right_col, TILE_PH_T_RIGHT);

        put(buf, 9, left_col,  step == 0 ? TILE_PH_BL : TILE_PH_B_LEFT);
        put(buf, 9, right_col, TILE_PH_B_RIGHT);
    }

    /* TILE_PLAY when state->playback_state == 0, TILE_PAUSE when 1 */
    put(buf, 10, 18, state->playback_state == 0 ? TILE_PLAY : TILE_PAUSE);

    /* Convert state->measure, state->global_bpm, and pitch/volume
     * of the cursor cell into their digit tiles using TILE_NUM_X,
     * BPM is 3 digits (cols 13-15); the others are 2 digits (cols 14-15) */
    unsigned int bpm = state->global_bpm;
    put(buf, 11, 13, num_tiles[(bpm / 100) % 10]);
    put(buf, 11, 14, num_tiles[(bpm /  10) % 10]);
    put(buf, 11, 15, num_tiles[ bpm        % 10]);

    unsigned int meas = state->measure;
```

```
put(buf, 10, 14, num_tiles[(meas / 10) % 10]);
put(buf, 10, 15, num_tiles[ meas       % 10]);

const cell *c = &state->sequence[base + state->cursor_step][state->cursor_track];

int pitch_disp = (int)c->pitch;
unsigned int pitch_mag = (pitch_disp >= 0) ? (unsigned)pitch_disp
                                           : (unsigned)(-pitch_disp);
put(buf, 12, 13, pitch_disp >= 0 ? TILE_VAL_PITCH_PLUS_R
                                 : TILE_VAL_PITCH_MINUS_R);
put(buf, 12, 14, num_tiles[(pitch_mag / 10) % 10]);
put(buf, 12, 15, num_tiles[ pitch_mag       % 10]);

put(buf, 13, 14, num_tiles[(c->volume / 10) % 10]);
put(buf, 13, 15, num_tiles[ c->volume       % 10]);

if (state->edit_mode == NAVIGATION)
{
    /* Mode labels at cols 1-2:
     *   Row 10: MEASURE SELECT (1 row x 2 cols)
     *   Rows 11-12: NAV MODE   (2 rows x 2 cols) */
    put(buf, 10, 1, TILE_MODE_MEA_SEL_L);  put(buf, 10, 2, TILE_MODE_MEA_SEL_R);
    put(buf, 11, 1, TILE_MODE_NAV_TL);     put(buf, 11, 2, TILE_MODE_NAV_TR);
    put(buf, 12, 1, TILE_MODE_NAV_BL);     put(buf, 12, 2, TILE_MODE_NAV_BR);

    /* Measure-select buttons 1-8 across rows 10-11 (4 per row, 2 tiles each),
     * the currently-selected measure uses the SEL variant. Also, since
     * state->measure is, 1-indexed, subtract 1 for the table lookup */
    for (int m = 0; m < 8; m++)
    {
        int row = 10 + (m / 4);
        int col = 3  + (m % 4) * 2;
        int sel = ((unsigned)(m + 1) == state->measure);
        const tile_pair_t *t = sel ? &measure_sel_tiles[m]
                                   : &measure_tiles[m];
        put(buf, row, col,     t->left);
        put(buf, row, col + 1, t->right);
    }

    /* Row 12 palette:   NOTE_ADD/REM   [empty]   ARR_UP   [empty]
     * If the current cursor cell is active, show REM; otherwise ADD */
    if (c->active)
    {
        put(buf, 12, 3, TILE_NOTE_REM_L);  put(buf, 12, 4, TILE_NOTE_REM_R);
    }
    else
    {
        put(buf, 12, 3, TILE_NOTE_ADD_L);  put(buf, 12, 4, TILE_NOTE_ADD_R);
    }
    put(buf, 12, 7, TILE_ARR_UP_L);    put(buf, 12, 8, TILE_ARR_UP_R);
```

```
        /* Row 13 palette:   TRACK_CLEAR   ARR_LEFT   ARR_DOWN   ARR_RIGHT */
        put(buf, 13, 3, TILE_TRACK_CLEAR_L);  put(buf, 13, 4, TILE_TRACK_CLEAR_R);
        put(buf, 13, 5, TILE_ARR_LEFT_L);     put(buf, 13, 6, TILE_ARR_LEFT_R);
        put(buf, 13, 7, TILE_ARR_DOWN_L);     put(buf, 13, 8, TILE_ARR_DOWN_R);
        put(buf, 13, 9, TILE_ARR_RIGHT_L);    put(buf, 13, 10, TILE_ARR_RIGHT_R);
    }
    else  /* INSTRUMENT_SELECT */
    {
        /* Mode label at cols 1-2:
         *   Rows 11-12: INSTR SELECT MODE (2 rows x 2 cols) */
        put(buf, 11, 1, TILE_MODE_INS_SEL_TL);  put(buf, 11, 2, TILE_MODE_INS_SEL_TR);
        put(buf, 12, 1, TILE_MODE_INS_SEL_BL);  put(buf, 12, 2, TILE_MODE_INS_SEL_BR);

        /* 16 instruments laid out 4 rows x 4 instruments (2 tiles each),
         * in instrument_defs[] order so palette position == instrument_id */
        for (int id = 0; id < NUM_VOICES; id++)
        {
            int row = 10 + (id / 4);  // Putting them into a 4x4 grid
            int col = 3  + (id % 4) * 2;
            put(buf, row, col,     instrument_defs[id].left);
            put(buf, row, col + 1, instrument_defs[id].right);
        }

        /* For the selected track (state->cursor_track), if an instrument has been
         * set for it (state->tracks[t] != -1), highlight/select the instrument in
         * the instrument palette */
        int cur_id = state->tracks[state->cursor_track];
        if (cur_id != -1)
        {
            int pal_row = 10 + (cur_id / 4);
            int pal_col = 3  + (cur_id % 4) * 2;
            put(buf, pal_row, pal_col,     instrument_defs[cur_id].sel_left);
            put(buf, pal_row, pal_col + 1, instrument_defs[cur_id].sel_right);
        }
    }
}
```

state_to_tiles.h, Jordan Lin

```
None
#ifndef _STATE_TO_TILES_H
#define _STATE_TO_TILES_H

#include <stdint.h>

/*
 * Symbolic names for tile IDs in our tileset.  192 tiles defined
 * (0x00-0xbd); the tileset PNG is 16x12 (16 cols x 12 rows of 16x16
```

```
 * tiles), and the hardware supports up to 256 tiles (8-bit tile
 * number).  This file is the single source of truth for the mapping,
 * so once a tile gets a name here, code throughout the project
 * should use the name and never the literal hex.
 *
 * Naming convention: ALL_CAPS, prefixed with TILE_, organized by
 * function. Group related tiles (numbers, bar pieces, etc.) and
 * leave a blank line between groups.
 */
typedef enum {
    TILE_EMPTY              = 0x00,  // Empty tile (completely white)

    /* Grid background, cell separators, & notes */
    TILE_GRID               = 0x01,  // Normal grid cell (lines top and left)
    TILE_GRID_4             = 0x02,  // Grid cell with extra right line (for every four grid
in a row)
    TILE_GRID_R_EDGE        = 0x03,  // Only left line, for the right edge of the entire grid
    TILE_GRID_B_EDGE        = 0x04,  // Only top line, for the bottom edge of the entire grid
    TILE_GRID_BR_CORNER     = 0x05,  // Only top-left pixel, for bottom-right corner of the
entire grid
    TILE_GRID_NOTE          = 0x06,  // TILE_GRID with note filled in
    TILE_GRID_NOTE_4        = 0x07,  // TILE_GRID_4 with note filled in
    TILE_GRID_SEL           = 0x08,  // TILE_GRID, selected
    TILE_GRID_SEL_4         = 0x09,  // TILE_GRID_4, selected
    TILE_GRID_NOTE_SEL      = 0x0a,  // TILE_GRID_NOTE, selected
    TILE_GRID_NOTE_SEL_4    = 0x0b,  // TILE_GRID_NOTE_4, selected

    /* Play & pause */
    TILE_PLAY               = 0x0e,  // Play symbol
    TILE_PAUSE              = 0x0f,  // Pause symbol

    /* Beat numbers & playhead */
    TILE_BEAT_0             = 0x10,  // Beat number on top of grid, 0
    TILE_BEAT_4             = 0x11,  // Beat number on top of grid, 4
    TILE_BEAT_8             = 0x12,  // Beat number on top of grid, 8
    TILE_BEAT_12            = 0x13,  // Beat number on top of grid, 12
    TILE_BEAT_16            = 0x14,  // Beat number on top of grid, 16
    TILE_BEAT_0_PH          = 0x15,  // TILE_BEAT_0, with left half of top playhead
    TILE_BEAT_4_PH          = 0x16,  // TILE_BEAT_4, with left half of top playhead
    TILE_BEAT_8_PH          = 0x17,  // TILE_BEAT_8, with left half of top playhead
    TILE_BEAT_12_PH         = 0x18,  // TILE_BEAT_12, with left half of top playhead
    TILE_PH_T_LEFT          = 0x19,  // Top playhead, left half
    TILE_PH_T_RIGHT         = 0x1a,  // Top playhead, right half
    TILE_PH_B_LEFT          = 0x1b,  // Bottom playhead, left half (has top line for bottom
of grid)
    TILE_PH_B_RIGHT         = 0x1c,  // Bottom playhead, right half (has top line for bottom
of grid)
    TILE_PH_BL              = 0x1d,  // TILE_PH_B_LEFT but without top line, for playhead at
bottom-left corner of grid
```

```
/* Numbers (0-9) */
TILE_NUM_0              = 0x1e,  // Number, 0
TILE_NUM_1              = 0x2d,  // Number, 1
TILE_NUM_2              = 0x2e,  // Number, 2
TILE_NUM_3              = 0x2f,  // Number, 3
TILE_NUM_4              = 0x3d,  // Number, 4
TILE_NUM_5              = 0x3e,  // Number, 5
TILE_NUM_6              = 0x3f,  // Number, 6
TILE_NUM_7              = 0x4d,  // Number, 7
TILE_NUM_8              = 0x4e,  // Number, 8
TILE_NUM_9              = 0x4f,  // Number, 9

/* Adjustable values with knobs (measure, BPM, pitch, & volume) */
TILE_VAL_PITCH_L        = 0x5d,  // Pitch +/- text, left
TILE_VAL_PITCH_PLUS_R   = 0x5e,  // Pitch + text, right
TILE_VAL_PITCH_MINUS_R  = 0x5f,  // Pitch - text, right
TILE_VAL_BPM            = 0x6d,  // BPM text
TILE_VAL_MEASURE_L      = 0x6e,  // Measure text, left
TILE_VAL_MEASURE_R      = 0x6f,  // Measure text, right
TILE_VAL_VOLUME_L       = 0x7d,  // Volume text, left
TILE_VAL_VOLUME_R       = 0x7e,  // Volume text, right

/* Instruments/sounds, with border (for changing instruments, unselected) */
TILE_INS_CRASH_L        = 0x20,  // Instrument, Crash1/2, left
TILE_INS_CRASH1_R       = 0x21,  // Instrument, Crash1, right
TILE_INS_CRASH2_R       = 0x22,  // Instrument, Crash2, right
TILE_INS_RIDE_L         = 0x23,  // Instrument, Ride1/2, left
TILE_INS_RIDE1_R        = 0x24,  // Instrument, Ride1, right
TILE_INS_RIDE2_R        = 0x25,  // Instrument, Ride2, right
TILE_INS_LOWTOM_L       = 0x26,  // Instrument, LowTom, left
TILE_INS_TOM_R          = 0x27,  // Instrument, Low/MidTom, right
TILE_INS_MIDTOM_L       = 0x28,  // Instrument, MidTom, left
TILE_INS_HITOM_L        = 0x29,  // Instrument, HiTom, left
TILE_INS_HITOM_R        = 0x2a,  // Instrument, Hitom, right
TILE_INS_CLAP_L         = 0x2b,  // Instrument, Clap, left
TILE_INS_CLAP_R         = 0x2c,  // Instrument, Clap, right
TILE_INS_BASS_L         = 0x30,  // Instrument, Bass1/2, left
TILE_INS_BASS1_R        = 0x31,  // Instrument, Bass1, right
TILE_INS_BASS2_R        = 0x32,  // Instrument, Bass2, right
TILE_INS_SNARE_L        = 0x33,  // Instrument, Snare1/2/3, left
TILE_INS_SNARE1_R       = 0x34,  // Instrument, Snare1, right
TILE_INS_SNARE2_R       = 0x35,  // Instrument, Snare2, right
TILE_INS_SNARE3_R       = 0x36,  // Instrument, Snare3, right
TILE_INS_CLHAT_L        = 0x37,  // Instrument, ClosedHiHat, left
TILE_INS_CLHAT_R        = 0x38,  // Instrument, ClosedHiHat, right
TILE_INS_OPHAT_L        = 0x39,  // Instrument, OpenHiHat, left
TILE_INS_OPHAT_R        = 0x3a,  // Instrument, OpenHiHat, right
TILE_INS_RIMSHOT_L      = 0x3b,  // Instrument, RimShot, left
TILE_INS_RIMSHOT_R      = 0x3c,  // Instrument, RimShot, right
```

```
/* Instruments/sounds, with border (for changing instruments, selected) */
TILE_INS_CRASH_SEL_L    = 0x40,  // TILE_INS_CRASH_L, selected
TILE_INS_CRASH1_SEL_R   = 0x41,  // TILE_INS_CRASH1_R, selected
TILE_INS_CRASH2_SEL_R   = 0x42,  // TILE_INS_CRASH2_R, selected
TILE_INS_RIDE_SEL_L     = 0x43,  // TILE_INS_RIDE_L, selected
TILE_INS_RIDE1_SEL_R    = 0x44,  // TILE_INS_RIDE1_R, selected
TILE_INS_RIDE2_SEL_R    = 0x45,  // TILE_INS_RIDE2_R, selected
TILE_INS_LOWTOM_SEL_L   = 0x46,  // TILE_INS_LOWTOM_L, selected
TILE_INS_TOM_SEL_R      = 0x47,  // TILE_INS_TOM_R, selected
TILE_INS_MIDTOM_SEL_L   = 0x48,  // TILE_INS_MIDTOM_L, selected
TILE_INS_HITOM_SEL_L    = 0x49,  // TILE_INS_HITOM_L, selected
TILE_INS_HITOM_SEL_R    = 0x4a,  // TILE_INS_HITOM_R, selected
TILE_INS_CLAP_SEL_L     = 0x4b,  // TILE_INS_CLAP_L, selected
TILE_INS_CLAP_SEL_R     = 0x4c,  // TILE_INS_CLAP_R, selected
TILE_INS_BASS_SEL_L     = 0x50,  // TILE_INS_BASS_L, selected
TILE_INS_BASS1_SEL_R    = 0x51,  // TILE_INS_BASS1_R, selected
TILE_INS_BASS2_SEL_R    = 0x52,  // TILE_INS_BASS2_R, selected
TILE_INS_SNARE_SEL_L    = 0x53,  // TILE_INS_SNARE_L, selected
TILE_INS_SNARE1_SEL_R   = 0x54,  // TILE_INS_SNARE1_R, selected
TILE_INS_SNARE2_SEL_R   = 0x55,  // TILE_INS_SNARE2_R, selected
TILE_INS_SNARE3_SEL_R   = 0x56,  // TILE_INS_SNARE3_R, selected
TILE_INS_CLHAT_SEL_L    = 0x57,  // TILE_INS_CLHAT_L, selected
TILE_INS_CLHAT_SEL_R    = 0x58,  // TILE_INS_CLHAT_R, selected
TILE_INS_OPHAT_SEL_L    = 0x59,  // TILE_INS_OPHAT_L, selected
TILE_INS_OPHAT_SEL_R    = 0x5a,  // TILE_INS_OPHAT_R, selected
TILE_INS_RIMSHOT_SEL_L  = 0x5b,  // TILE_INS_RIMSHOT_L, selected
TILE_INS_RIMSHOT_SEL_R  = 0x5c,  // TILE_INS_RIMSHOT_R, selected

/* Instruments/sounds, without border (for track instrument indication) */
TILE_INS_CRASH_NB_L     = 0x60,  // TILE_INS_CRASH_L
TILE_INS_CRASH1_NB_R    = 0x61,  // TILE_INS_CRASH1_R
TILE_INS_CRASH2_NB_R    = 0x62,  // TILE_INS_CRASH2_R
TILE_INS_RIDE_NB_L      = 0x63,  // TILE_INS_RIDE_L
TILE_INS_RIDE1_NB_R     = 0x64,  // TILE_INS_RIDE1_R
TILE_INS_RIDE2_NB_R     = 0x65,  // TILE_INS_RIDE2_R
TILE_INS_LOWTOM_NB_L    = 0x66,  // TILE_INS_LOWTOM_L
TILE_INS_TOM_NB_R       = 0x67,  // TILE_INS_TOM_R
TILE_INS_MIDTOM_NB_L    = 0x68,  // TILE_INS_MIDTOM_L
TILE_INS_HITOM_NB_L     = 0x69,  // TILE_INS_HITOM_L
TILE_INS_HITOM_NB_R     = 0x6a,  // TILE_INS_HITOM_R
TILE_INS_CLAP_NB_L      = 0x6b,  // TILE_INS_CLAP_L
TILE_INS_CLAP_NB_R      = 0x6c,  // TILE_INS_CLAP_R
TILE_INS_BASS_NB_L      = 0x70,  // TILE_INS_BASS_L
TILE_INS_BASS1_NB_R     = 0x71,  // TILE_INS_BASS1_R
TILE_INS_BASS2_NB_R     = 0x72,  // TILE_INS_BASS2_R
TILE_INS_SNARE_NB_L     = 0x73,  // TILE_INS_SNARE_L
TILE_INS_SNARE1_NB_R    = 0x74,  // TILE_INS_SNARE1_R
TILE_INS_SNARE2_NB_R    = 0x75,  // TILE_INS_SNARE2_R
TILE_INS_SNARE3_NB_R    = 0x76,  // TILE_INS_SNARE3_R
TILE_INS_CLHAT_NB_L     = 0x77,  // TILE_INS_CLHAT_L
```

```
TILE_INS_CLHAT_NB_R     = 0x78,  // TILE_INS_CLHAT_R
TILE_INS_OPHAT_NB_L     = 0x79,  // TILE_INS_OPHAT_L
TILE_INS_OPHAT_NB_R     = 0x7a,  // TILE_INS_OPHAT_R
TILE_INS_RIMSHOT_NB_L   = 0x7b,  // TILE_INS_RIMSHOT_L
TILE_INS_RIMSHOT_NB_R   = 0x7c,  // TILE_INS_RIMSHOT_R

/* Navigation mode (arrows, note add/remove, & clear track) */
TILE_ARR_UP_L           = 0x80,
TILE_ARR_UP_R           = 0x81,
TILE_ARR_DOWN_L         = 0x82,
TILE_ARR_DOWN_R         = 0x83,
TILE_ARR_LEFT_L         = 0x84,
TILE_ARR_LEFT_R         = 0x85,
TILE_ARR_RIGHT_L        = 0x86,
TILE_ARR_RIGHT_R        = 0x87,
TILE_NOTE_ADD_L         = 0x88,
TILE_NOTE_ADD_R         = 0x89,
TILE_NOTE_REM_L         = 0x8a,
TILE_NOTE_REM_R         = 0x8b,
TILE_TRACK_CLEAR_L      = 0x8c,
TILE_TRACK_CLEAR_R      = 0x8d,

/* Measure select (two-tile numbers) */
TILE_DNUM_1_L           = 0x90,
TILE_DNUM_1_R           = 0x91,
TILE_DNUM_2_L           = 0x92,
TILE_DNUM_2_R           = 0x93,
TILE_DNUM_3_L           = 0x94,
TILE_DNUM_3_R           = 0x95,
TILE_DNUM_4_L           = 0x96,
TILE_DNUM_4_R           = 0x97,
TILE_DNUM_5_L           = 0x98,
TILE_DNUM_5_R           = 0x99,
TILE_DNUM_6_L           = 0x9a,
TILE_DNUM_6_R           = 0x9b,
TILE_DNUM_7_L           = 0x9c,
TILE_DNUM_7_R           = 0x9d,
TILE_DNUM_8_L           = 0x9e,
TILE_DNUM_8_R           = 0x9f,

/* Measure select (two-tile numbers, selected) */
TILE_DNUM_1_SEL_L       = 0xa0,
TILE_DNUM_1_SEL_R       = 0xa1,
TILE_DNUM_2_SEL_L       = 0xa2,
TILE_DNUM_2_SEL_R       = 0xa3,
TILE_DNUM_3_SEL_L       = 0xa4,
TILE_DNUM_3_SEL_R       = 0xa5,
TILE_DNUM_4_SEL_L       = 0xa6,
TILE_DNUM_4_SEL_R       = 0xa7,
TILE_DNUM_5_SEL_L       = 0xa8,
```

```
    TILE_DNUM_5_SEL_R       = 0xa9,
    TILE_DNUM_6_SEL_L       = 0xaa,
    TILE_DNUM_6_SEL_R       = 0xab,
    TILE_DNUM_7_SEL_L       = 0xac,
    TILE_DNUM_7_SEL_R       = 0xad,
    TILE_DNUM_8_SEL_L       = 0xae,
    TILE_DNUM_8_SEL_R       = 0xaf,

    /* Mode selection text & Boots N' Cats */
    TILE_MODE_INS_SEL_TL    = 0xb0,
    TILE_MODE_INS_SEL_TR    = 0xb1,
    TILE_MODE_INS_SEL_BL    = 0xb2,
    TILE_MODE_INS_SEL_BR    = 0xb3,
    TILE_MODE_NAV_TL        = 0xb4,
    TILE_MODE_NAV_TR        = 0xb5,
    TILE_MODE_NAV_BL        = 0xb6,
    TILE_MODE_NAV_BR        = 0xb7,
    TILE_MODE_MEA_SEL_L     = 0xb8,
    TILE_MODE_MEA_SEL_R     = 0xb9,
    TILE_BOOTSNCATS_TL      = 0xba,
    TILE_BOOTSNCATS_TR      = 0xbb,
    TILE_BOOTSNCATS_BL      = 0xbc,
    TILE_BOOTSNCATS_BR      = 0xbd,
} tile_id_t;

#include "drum.h"

/*
 * Single source of truth for instrument order, names, and palette tiles.
 * Index is the instrument_id; reorder this table (defined in
 * state_to_tiles.c) to reorder the palette, track-row labels, and the
 * names returned by instrument_name() in lockstep.  The audio engine's
 * voice/sample layout must also match this order.
 */
typedef struct {
    const char *name;
    tile_id_t left,     right;       /* with border, palette default */
    tile_id_t sel_left, sel_right;   /* with border, palette cursor */
    tile_id_t nb_left,  nb_right;    /* no border, track-row label */
} instrument_def_t;

extern const instrument_def_t instrument_defs[NUM_VOICES];

/*
 * Render the current program state into a 512-byte tilemap buffer
 * laid out for the hardware (row stride 32, 16 rows; only rows 0-14
 * cols 0-19 are visible).
 *
 * The buffer is what drum.c (or any caller) hands to ioctl(UI_WRITE)
 * at UI_TILEMAP_OFFSET with length UI_TILEMAP_SIZE.  Caller owns the
```

```
 * buffer.  This function is pure: no I/O, no globals, no statics.
 */
void state_to_tiles(const sequencer_state *state, unsigned char *tilemap);

#endif
```

ui.c, Jordan Lin

None

```
/*
 * Device driver for the ui tile-renderer peripheral.
 *
 * Exposes /dev/ui.  A single ioctl (UI_WRITE) takes an offset, length,
 * and userspace buffer, and writes each byte to the peripheral's iomem
 * region.  Userspace uses this to upload the tilemap, tileset, and
 * palette in three batched calls.
 *
 * "make" to build
 * insmod ui.ko
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "ui.h"

#define DRIVER_NAME "ui"

/* Peripheral iomem window is 128KB (17-bit address, 8-bit data). */
#define UI_IOMEM_SIZE  0x20000

struct ui_dev {
    struct resource res;
    void __iomem *virtbase;
} dev;

/*
 * Copy a userspace buffer into the iomem region one byte at a time.
 * Avalon's 8-bit slave handles each iowrite8 as a single bus cycle,
```

```
 * which is exactly how the tilemap/tileset/palette commits expect
 * to be written.
 */
static long ui_write_region(const ui_write_arg_t *arg)
{
    unsigned int i;
    unsigned char *kbuf;

    if (arg->length == 0)
        return 0;
    if (arg->offset >= UI_IOMEM_SIZE)
        return -EINVAL;
    if (arg->length > UI_IOMEM_SIZE)
        return -EINVAL;
    if (arg->offset + arg->length > UI_IOMEM_SIZE)
        return -EINVAL;

    kbuf = kmalloc(arg->length, GFP_KERNEL);
    if (!kbuf)
        return -ENOMEM;

    if (copy_from_user(kbuf, arg->buf, arg->length)) {
        kfree(kbuf);
        return -EACCES;
    }

    for (i = 0; i < arg->length; i++)
        iowrite8(kbuf[i], dev.virtbase + arg->offset + i);

    kfree(kbuf);
    return 0;
}

static long ui_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    ui_write_arg_t warg;

    switch (cmd) {
    case UI_WRITE:
        if (copy_from_user(&warg, (ui_write_arg_t *) arg,
                   sizeof(ui_write_arg_t)))
            return -EACCES;
        return ui_write_region(&warg);

    default:
        return -EINVAL;
    }
}

static const struct file_operations ui_fops = {
```

```
    .owner        = THIS_MODULE,
    .unlocked_ioctl = ui_ioctl,
};

static struct miscdevice ui_misc_device = {
    .minor        = MISC_DYNAMIC_MINOR,
    .name        = DRIVER_NAME,
    .fops        = &ui_fops,
};

static int __init ui_probe(struct platform_device *pdev)
{
    int ret;

    ret = misc_register(&ui_misc_device);

    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }

    return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&ui_misc_device);
    return ret;
}

static int ui_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&ui_misc_device);
    return 0;
}
```

```
#ifdef CONFIG_OF
static const struct of_device_id ui_of_match[] = {
    { .compatible = "csee4840,ui-1.0" },
    {},
};
MODULE_DEVICE_TABLE(of, ui_of_match);
#endif

static struct platform_driver ui_driver = {
    .driver    = {
        .name    = DRIVER_NAME,
        .owner    = THIS_MODULE,
        .of_match_table = of_match_ptr(ui_of_match),
    },
    .remove    = __exit_p(ui_remove),
};

static int __init ui_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&ui_driver, ui_probe);
}

static void __exit ui_exit(void)
{
    platform_driver_unregister(&ui_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(ui_init);
module_exit(ui_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("CSEE 4840 Drum Machine Team");
MODULE_DESCRIPTION("Tile-renderer driver for drum machine UI");
```

ui.h, Jordan Lin

None

```
#ifndef _UI_H
#define _UI_H

#include <linux/ioctl.h>

/*
 * Userspace passes a buffer + offset; the kernel writes each byte to
 * the peripheral's iomem region at offset+i.  One ioctl uploads an
 * entire region (tilemap, tileset, or palette) in a single call.
```

```
 *
 * Offsets follow the hardware memory map in ui.sv:
 *   0x00000 - 0x001FF   Tilemap   (512 bytes)
 *   0x00200 - 0x0020F   Palette   (16 bytes: 4 colors x R,G,B,commit)
 *   0x10000 - 0x1FFFF   Tileset   (65536 bytes, lower 2 bits used)
 */
typedef struct {
    unsigned int  offset;   /* byte offset within peripheral, 0..0x1FFFF */
    unsigned int  length;   /* number of bytes to write */
    const unsigned char *buf;  /* userspace buffer */
} ui_write_arg_t;

#define UI_MAGIC 'u'

#define UI_WRITE  _IOW(UI_MAGIC, 1, ui_write_arg_t)

/* Convenience offsets matching the hardware memory map */
#define UI_TILEMAP_OFFSET   0x00000
#define UI_TILEMAP_SIZE     512
#define UI_PALETTE_OFFSET   0x00200
#define UI_PALETTE_SIZE     16
#define UI_TILESET_OFFSET   0x10000
#define UI_TILESET_SIZE     65536

#endif
```

Appendix D: Contributions & Learning

Our project naturally divided into three major components that was quite cleanly divided between the three of us: Audio playing (Noel Gomez), GUI design and display (Jordan Lin), and main drum machine program and connecting everything together (Aaron Zhu).

Noel Gomez

I did the preprocessing steps for our audio samples, designed and wrote the FPGA hardware audio architecture (audio_engine.sv and sample_clock.sv), and helped my partners optimize our user interface design in a way that was easy to use and build with synergy between the various modules. One of the biggest things I learned was to draw everything out first on pencil and paper. This was surprisingly helpful to catch bugs and pitfalls early and help me see architecture from a bigger lens than I could trying to analyze in my head. This was my first time using SystemVerilog so I learned a lot about navigating clock cycles which was a new way of thinking in code from an electrical engineering background. Considering BRAM storage capacity and hardware resource allocation was a fun hurdle too. I'm also very happy that this project deepened my understanding of audio sampling and signal processing.

Aaron Zhu

On the software side, I wrote the main program in drum.c, which handles everything from reading USB MIDI input via libusb, to updating the global sequencer state, to writing to the FPGA audio engine for playback. I also handled the high-level logic for rendering the tilemap onto our VGA display, but it was really Jordan who worked out all the nitty-gritty details on rendering the tilemap (as well as hand drawing the tiles!). I also worked on the i2s_tx.sv module, which required me to carefully learn how I2S works and how audio CODECs like the WM8731 are interfaced, none of which I had any prior knowledge of coming into this project.

Another thing I worked on was just making sure everything was wired together correctly and debugging issues as they came up. One of the more interesting problems was that when we were running *make quartus,* it would error out and tell us our design was too big. We hypothesized that it was potentially due to a large number of multiple operations in parallel, but even after reducing the parallelism, we were running into the error. Then, I saw that in actuality it was that a huge number of LUTs were being used: 47K, which is 15K over the 32K budget. The M10K blocks we intended to use for the audio engine's sample storage was, on the flip side, were being under-utilized, inferred as LUTs instead. I traced it down to a conditional assignment in a single always block that was

preventing Quartus from recognizing the memory structure as a true dual-port RAM. Once I fixed that, the M10K inference worked correctly and the design fit.

Overall, I was very happy to work on this project with my partners and I think it came out fantastic in the end! Thank you for everything, CSEE 4840 Spring 2026.

Jordan Lin

I designed the UI layout and tileset by hand, implemented the FPGA hardware tilemap display architecture (ui.sv and tiles.sv) based on the VGA Tile Graphics guide from the class website (vga_counters.sv and twoportbram.sv are used essentially as-is from the guide), wrote the abstraction for all the tiles and converting the program state to the tile IDs for display (state_to_tiles.c and state_to_tiles.h), and the helper functions for calling the UI rendering peripherals in software (ui.c and ui.h). One of the biggest things I learned is the benefits of modularization. Unlike a lot of my past collaborative group projects, we were very clear with the division of work at the beginning and emphasized writing code which had minimal assumptions from the other components (besides our agreed upon hardware-software and software-software interfaces). This modularization not only enabled us to develop (and test) a big chunk of the project asynchronously, it also enabled us to relatively quickly locate bugs and make localized changes that required little or no recompilation. One of my biggest hurdles was just the mental hurdle of figuring out how the dual-port BRAM worked and why it was necessary, but once I saw how neatly (and modularized) the two sides (with different clockspeeds) was linked together by the BRAM I was convinced of this architecture's effectiveness. Designing the tiles was surprisingly time-consuming (and fun) as well. As I knew the audio would take up a lot of memory, I tried to design the UI to have as much tile reuse as possible, while having something that looks visually coherent and appealing. As I leaned into the retro aesthetic, I became very happy with the final UI I came up with, and I am very happy to have toyed around with a method so commonly used in early video games.