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

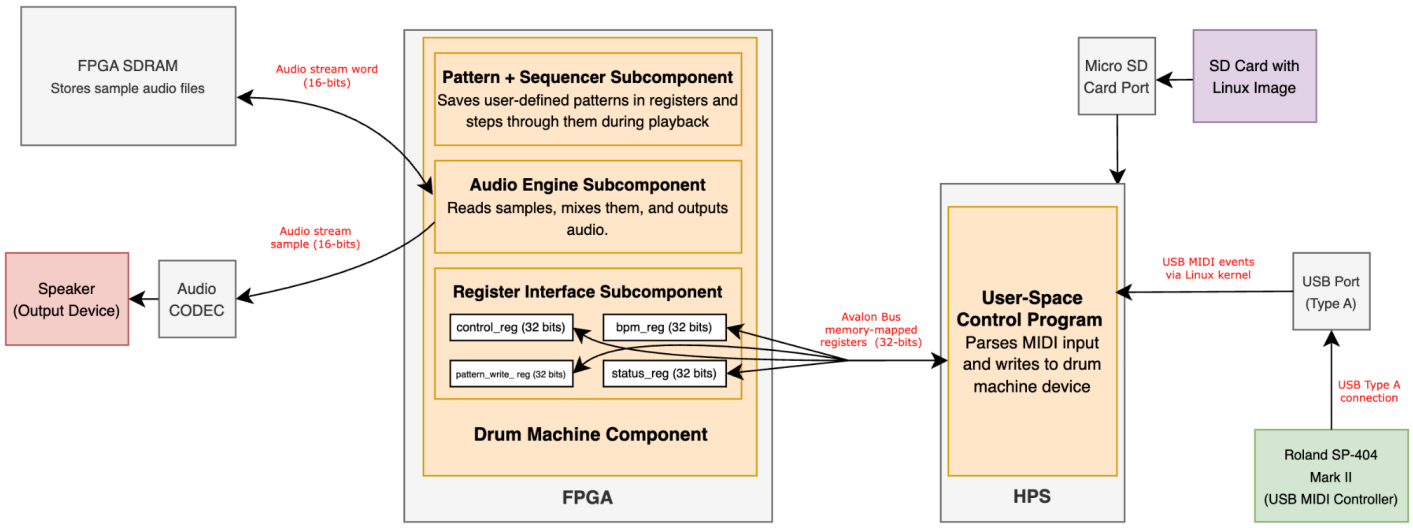
Contents

1. Introduction

The Roland TR-909 drum machine was released in 1983. It was a revolutionary analog/digital hybrid instrument—the first from Roland to use sampled sounds for cymbals/hi-hats alongside analog circuitry—and one of the first to feature MIDI. It was wildly influential to the growth of electronic music and the modern workflow of music production on DAWs like Ableton and FL Studio which today include the 909 sample bank as stock sounds.

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 Diagram



3. User Interface

3.1 Input Interface Device (Roland SP-404 MK II)

The drum machine system will allow the user to create, edit, and play back a rhythmic pattern known as a beat, by interacting with a Roland SP-404 MK II, a physical sampler device (Figure 1). A beat consists of up to 8 tracks arranged over 8 bars — units of musical time — where each bar is divided into 16 discrete, equal-sized time steps, for a total of 128 steps. For each track, the user may specify whether the instrument they selected for that track should sound — be active — at any of these steps.

To construct a beat, the user first selects one of the 8 tracks, then an instrument out of a sample bank of 16. Once they have selected an instrument and track, the user can then edit the track at one of its 8 bars. (All steps for all tracks are inactive at system startup.) Repeating this process across tracks and bars allows the user to define when each instrument plays throughout the beat. During this editing process, the user may also set the pitch of a note for a given track at a given step and solo the sample to hear it alone.

After constructing a beat, the user may switch to playback mode, in which the system sequences through all 128 steps in time and outputs the corresponding audio. The resulting sound is played through a speaker connected to the DE1-SoC board's audio codec. The user may adjust the global playback rate of the beat (BPM), which determines how quickly the system advances through the steps.

To perform these actions, the user interacts with four color-coordinated groups of controls on the Roland SP-404MKII, as shown in Figure 1. Here is an overview of their functionality.

1. The red group of 16 pads are used to either select a track or specify which steps in a bar are active, depending on the editing mode.
2. The green group of buttons set the editing context. By pressing the first pad "A/F", the user selects which track to play or which instrument to select. By selecting the other four — the user selects which of the eight bars of the beat is being currently edited.
3. During editing, the yellow group of knobs control pitch for particular tracks at particular steps; during playback, they control the beat's BPM.
4. The pink group of pads manages playback: the top pad ("EXT SOURCE") allows the user to toggle switch between editing and playback and hear

the beat, and the bottom pad (“SUB PAD") allows the user to save their current work in the editing session.

Figure 1: Roland SP-404 MK II (used controls grouped by color)

For more details on the device's controls:

The buttons labeled “A/F”, “B/G”, “C/H”, “D/I”, and “E/J” allow the user to select the editing mode. These buttons determine how the numbered pads are interpreted by the system.

When the user presses “A/F”, the user selects between two related editing contexts depending on the button’s state. When the button is solid, the user enters track-selection mode. In this mode, each of the first 8 numbered pads corresponds directly to one of the 8 instrumental tracks, and pressing a pad selects the track whose steps will be edited. When the button is flashing, the user enters instrument-selection mode. In this mode, each of the 16 numbered pads corresponds directly to one of the 16 instruments in the sample bank, and pressing a pad assigns the selected instrument to the currently selected track.

By pressing one of the remaining buttons (“B/G”, “C/H”, “D/I”, or “E/J”), the user selects which bar to edit. The user can then press one of the 16 numbered pads to set the (already) selected instrument to be active at one of the 16 time steps within the bar. Each of these buttons has two lighting states: solid and flashing. The states represent pairs of consecutive bar numbers. The full mapping between button and lighting state and bar number is shown below, in Figure 2.

Button	Lighting State	Bar Number
B/G	solid	1
B/G	flashing	2
C/H	solid	3
C/H	flashing	4
D/I	solid	5
D/I	flashing	6
E/J	solid	7
E/J	flashing	8

Figure 2: Mapping of button and lighting state to bar number

For example, suppose the user starts a fresh session. The user first presses the “A/F” button and then presses pad 1, selecting Crash1 as the instrument to be edited. The user then presses the “D/I” button until it is solid, selecting measure 5 as the measure to be edited. Finally, the user presses pad 13, setting the selected instrument to play at the 13th time step within measure 5.

Finally, regarding the pink group of buttons: The user presses the “EXT SOURCE” pad to toggle between editing mode and playback mode. When the pad is lit, the user hears the beat played back in real time, and when it is unlit, the user returns to editing. The user presses the “SUB PAD” to save the current state of the session.

During editing, the user uses the knobs to control pitch for the currently selected instrument at the currently selected step. During playback, the user uses the same knobs to control the global tempo (BPM) of the beat.

3.2 Output Interface Device (Dell VGA Display)

On startup, the user sees the graphical user interface (GUI) for the drum machine on the connected display. It consists of a 4×4 sample bank labeled with the names of all 16 samples and an empty beat grid with 16 divisions per bar. The display shows one bar at a time and which bar/measure being shown can be controlled by changing the measure number. By default, we provide 8 tracks which allows the users to program up to 8 sounds. The sounds for each track are empty by default, and the user can customize the sound on each row from the 4x4 sample bank. During playback, a cursor will move across the grid to show what part of the beat is playing in realtime, while also automatically switching between measures.

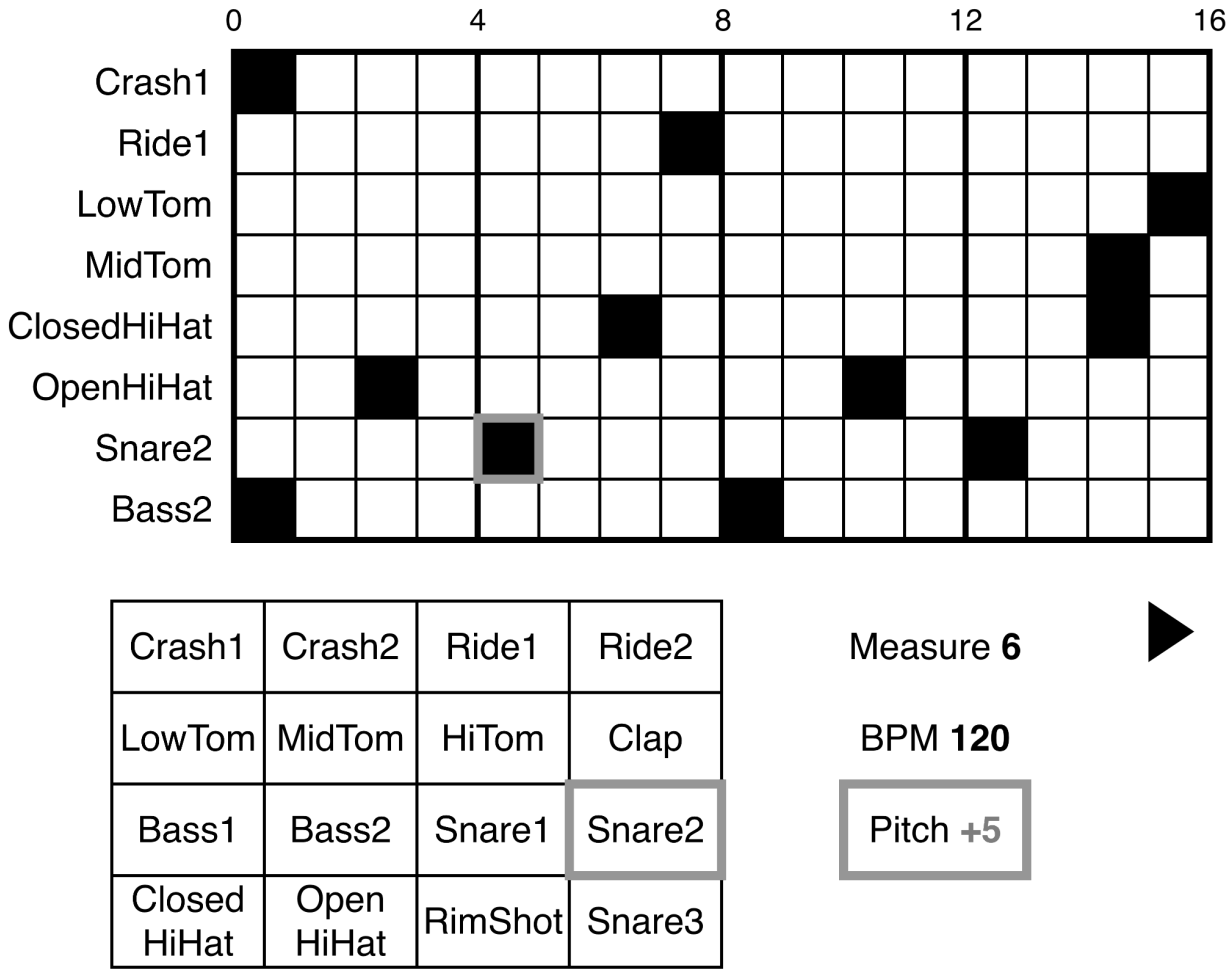


Figure 3: Diagram of the system GUI. Currently, the user has selected a note in Snare 2 and shifted its pitch up by +5. Gray outline indicates selection.

4. Hardware-Software Interface

The top-level FPGA component of the system exposes a set of memory-mapped control registers through the Avalon Memory-Mapped (Avalon-MM) slave interface. The HPS writes to these registers to communicate user actions, which it decodes from USB input. Each register is 32 bits wide and occupies a distinct address in the memory map. The top-level FPGA component reads these registers and updates its internal sequencing state accordingly.

From the perspective of software, the interface consists of five 32-bit control registers, which may be accessed as an array of five 32-bit words. Each register encodes a specific component of the user's interaction: track selection, instrument selection, measure selection, and step activation. The software should write to exactly one register at a time, with each write corresponding to a single user action and updating only one component of this state. The FPGA stores the current values of all registers internally and combines them to determine how the sequencing state is updated.

Within each 32-bit register, bit 0 denotes the least significant bit (LSB) and bit 31 denotes the most significant bit (MSB). Bit ranges are specified using the notation `[high:low]`, inclusive. The FPGA reads the `writedata[31:0]` signal and extracts fields according to these bit positions.

For the software, the offsets of the registers in the memory-mapped region for the component are as follows:

```
0x00 → Register 0
0x04 → Register 1
0x08 → Register 2
0x0C → Register 3
0x10 → Register 4
```

The registers are defined as follows:

Register 0: Track Select Register
Selects which of the 8 tracks is currently active.

Bits	Name	Description

[2:0]	track_id	Track index (0–7)
[31:3]	reserved	Must be written as 0

Register 1: Instrument Select Register
Selects which of the 16 instruments is assigned to the currently selected track.

Bits	Name	Description
[3:0]	instrument_id	Instrument index (0–15)
[31:4]	reserved	Must be written as 0

Register 2: Measure Select Register
Selects which of the 8 measures (bars) is currently active.

Bits	Name	Description
[2:0]	measure_id	Measure index (0–7)
[31:3]	reserved	Must be written as 0

Register 3: Step Write Register
Encodes a step activation/deactivation event for the currently selected track and measure.

Bits	Name	Description
[3:0]	step_id	Step index (0–15)
[4]	step_value	1 = activate step, 0 = deactivate step
[31:5]	reserved	Must be written as 0

Register 4: Control Register
Encodes a value for either pitch manipulation (knob CTRL1) or BPM (knob CTRL2), depending on the current mode and `ctrl_sel`.

Bits	Name	Description
[6:0]	ctrl_value	Value (0–127)
[7]	ctrl_sel	0 = CTRL1 (pitch), 1 = CTRL2 (BPM)
[31:8]	reserved	Must be written as 0

5. FPGA Component Details

The top-level FPGA component operates on the 50 MHz system clock (clk_50) and interfaces with the HPS through an Avalon Memory-Mapped (Avalon-MM) slave interface. The interface exposes the following signals:

```
clk              (input)  - system clock (50 MHz)
reset            (input)  - synchronous reset
address          (input)  - register address (2-3 bits)
write            (input)  - write enable
read             (input)  - read enable (unused)
writedata[31:0]  (input)  - data written by HPS
readdata[31:0]   (output) - data returned to HPS (optional)
```

On each rising edge of the `clk`, when write is asserted, the component decodes address and updates the corresponding internal control register using `writedata[31:0]`. The component maintains these register values internally and uses them to drive sequencing and playback behavior.

The top-level component instantiates two primary subcomponents: the Pattern + Sequencer Subcomponent and the Audio Engine Subcomponent.

5.1 Pattern + Sequencer Subcomponent

The system stores sequencing data at the resolution of 128 sixteenth-note steps (8 measures × 16 steps per measure). Each step stores which of the 16 samples in the current bank are active at that step.

This can be represented as 128 registers of 16 bits each, where each bit corresponds to one of the 16 possible samples. A value of 1 indicates that the sample is active at that step, and 0 indicates that it is inactive.

When the user programs a step, the system uses the current values of the track, measure, and step registers to determine which sixteenth-note register to update. The selected sample is then written into that step's register.

Pitch data is also associated with each sample. To avoid excessive memory usage, pitch values are stored per sample rather than per step. This allows each sample to maintain a consistent pitch across all steps, rather than requiring separate pitch storage for every step.

The sequencer continuously tracks the current playback position and provides the corresponding step register to the audio engine during playback.

5.2 Audio Engine Subcomponent

The playback rate is determined by the BPM value. The system derives a base frequency:

```
CLK_BPM = BPM / 60 Hz
```

During playback, the system generates a playback clock:

```
CLK_PLAY = 4 × CLK_BPM
```

Each rising edge of CLK_PLAY corresponds to one sixteenth-note step. On each edge, the system reads the corresponding step register and determines which samples are active.

For each active sample, the system fetches the corresponding audio data from memory, applies the stored pitch value, and mixes all active samples together. The combined signal is normalized by dividing by the number of active samples to avoid clipping.

The resulting audio signal is sent to the Audio CODEC once per `CLK_PLAY` cycle. After reaching step 127, the sequencer wraps back to step 0.

The system allows updates to sequencing data during playback. Writes from the HPS immediately update the corresponding step registers, allowing the user to modify the pattern in real time.

Pressing play enables the playback clock and audio output; pressing play again disables playback and returns the system to sequencing mode.

6. Algorithms

6.1 Preprocessing Audio-Sample Address Mapping

```python
import wave

FILENAME = "909samplebank.wav"
FS = 44100

# --- Load WAV file ---
with wave.open(FILENAME, "rb") as f:
    n_channels = f.getnchannels()
    sampwidth = f.getsampwidth()
    framerate = f.getframerate()
    n_frames = f.getnframes()

    print("Channels:", n_channels)
    print("Sample width (bytes):", sampwidth)
    print("Sample rate:", framerate)
    print("Frames:", n_frames)

    raw_bytes = f.readframes(n_frames)

# --- Convert 24-bit stereo → 16-bit mono (averaged) ---
samples = []

for i in range(0, len(raw_bytes), sampwidth * n_channels):
    # --- Left channel (first 3 bytes) ---
    b0 = raw_bytes[i]
    b1 = raw_bytes[i+1]
    b2 = raw_bytes[i+2]

    L = b0 | (b1 << 8) | (b2 << 16)
    if L & 0x800000:
        L -= 1 << 24

    # --- Right channel (next 3 bytes) ---
    b3 = raw_bytes[i+3]
    b4 = raw_bytes[i+4]
    b5 = raw_bytes[i+5]

    R = b3 | (b4 << 8) | (b5 << 16)
    if R & 0x800000:
        R -= 1 << 24

    # --- Average to mono ---
    mono = (L + R) // 2
    # --- Downscale 24-bit → 16-bit ---
    mono_16 = mono >> 8

    samples.append(mono_16)

print("Converted samples:", len(samples))

# --- OPTIONAL: Trim for testing (uncomment if needed) ---
# samples = samples[:44100 * 3]  # first 3 seconds

# --- Generate sample start indices ---
start_times = [0.119 + 1.786 * n for n in range(16)]
start_indices = [int(t * FS) for t in start_times]

print("Sample start indices:", start_indices)

# --- Write PCM samples to MIF ---
with open("pcm_samples.mif", "w") as f:
    f.write(f"DEPTH = {len(samples)};\n")
    f.write("WIDTH = 16;\n")
    f.write("ADDRESS_RADIX = DEC;\n")
    f.write("DATA_RADIX = DEC;\n")
    f.write("CONTENT BEGIN\n")

    for i, s in enumerate(samples):
        f.write(f"{i} : {s};\n")

    f.write("END;\n")

# --- Write sample start indices to MIF ---
with open("sample_starts.mif", "w") as f:
    f.write("DEPTH = 16;\n")
    f.write("WIDTH = 32;\n")
    f.write("ADDRESS_RADIX = DEC;\n")
    f.write("DATA_RADIX = DEC;\n")
    f.write("CONTENT BEGIN\n")

    for i, s in enumerate(start_indices):
        f.write(f"{i} : {s};\n")

    f.write("END;\n")

print("Done. Generated pcm_samples.mif and sample_starts.mif")
```

The audio samples used in the drum machine are stored as a single contiguous waveform in a .wav file. The samples are organized at one every 1.786sec with the first at 0.119sec in this order: Crash1, Crash2, Ride1, Ride2, LowTom, MidTom, OpenHiHat, HiTom, Bass1, Snare1, ClosedHiHat, Snare2, Bass2, RimShot, Snare3, Clap.To simplify hardware implementation, preprocessing is performed offline to convert this file into a format suitable for FPGA-based playback.

First, the .wav file is parsed in software to extract raw PCM data. The input file is converted from stereo 24-bit format to mono 16-bit PCM by averaging the left and right channels and downscaling each 24-bit sample to 16 bits. This produces a one-dimensional array of signed 16-bit samples:

$$pcm\_samples[n], \quad n = 0, 1, \ldots, N-1$$

This array is exported as a memory initialization file and loaded into external SDRAM accessible by the FPGA.

To enable efficient access to individual drum sounds within this contiguous buffer, a deterministic mapping from sample index to memory address is precomputed. Each drum sound is placed at a known time offset within the waveform, defined by:

$$t_n = t_0 + \Delta t \cdot n$$

where $t_0 = 0.119\text{sec}$ and $\Delta t = 1.786\text{sec}$. These time offsets are converted into sample indices using the sampling frequency $f_s = 44.1\ \text{kHz}$.

$$S_n = t_n \cdot f_s$$

Each Sn represents the starting address of sample n within the PCM memory. These values are precomputed in software and hardcoded into a lookup table in hardware:

$$\text{sample\_start}[n] = S_n$$

This approach eliminates the need for runtime file parsing or arithmetic, enabling constant-time retrieval of sample start addresses during playback.

6.2 MIDI Input Parsing

The Roland SP-404MKII communicates with the system over USB. MIDI messages are processed in software running on the HPS (Linux on ARM) using libusb hexadecimal inputs. For buttons, the input stream consists of data for On/Off, Channel, and Button ID (ignoring velocity). For knobs, the input MIDI message contains a knob ID byte and a value byte (from 0 to 128).

Our userspace program continuously polls on incoming MIDI events. On message with nonzero velocity, the note value is mapped to a corresponding drum sample ID using a predefined mapping: note → sample instrument or pad number → step number in bar.

Once a valid mapping is identified, the software writes the sample ID to a memory-mapped FPGA register depending on state. This interface provides a low-latency control path from user input to hardware execution, allowing real-time triggering of audio samples.

6.3 Step Sequencer

The step sequencer is implemented in hardware as a timing-driven finite state machine that iterates through a fixed-length pattern grid and generates sample

trigger events. The sequencer operates on a discretized time base derived from the user-defined tempo (BPM). A programmable clock divider converts the system clock into a step clock, where each tick corresponds to one step in the sequence, given by sixteenth notes. The step frequency is given by:

$$F_{step} = BPM \cdot 4/60$$

A step counter increments on each step clock tick where N=128 is the total number of steps in the pattern:

$$step = (step + 1) \bmod N$$

Pattern data is stored in memory as a two-dimensional structure indexed by instrument and step: `pattern[step][instrument]`.

At each step, the sequencer reads all instrument entries corresponding to the current step. If a value is active (e.g., 1), a trigger signal is generated for that instrument. These triggers are sent to the sample playback engine. This design allows multiple instruments to be triggered simultaneously and ensures consistent timing independent of software execution.

7. Resource Budgets

Name	Time (s)	f_s (kHz)	Memory (bit)
pcm_samples.mif	30.48	44.1	139,188,192
sample_starts.mif	-	-	1120
		Total memory (bit)	139,189,312

Because all 16 samples are in one file, we only list one audio file pcm_samples.mif. sample_starts.mif indicates the starting indices of each audio sample.