WaveSURFER: Wave Synthesis Using Real FPGA EngineeRing

Spring 2026 CSEE-W4840 Embedded Systems Final Report

Professor Stephen Edwards

Harry Minsky (hm3121), Opalina Khanna (ok2373), Sunny Fang (yf2610)

May 13, 2026

Contents

1 Introduction

This document describes the design of a wave table synthesizer modeled in reference to the Ensoniq 5503 Digital Oscillator Chip (DOC), with a register-based control interface inspired by the original architecture [1]. Unlike Frequency modulation (FM) synthesis, which generates audio by modulating a carrier waveform (such as a sine, square, or sawtooth wave) with a secondary modulator signal to produce new timbers, wave table synthesis operates by storing single-cycle waveforms directly in memory and playing them back at variable rates. While FM synthesis offers a highly exploratory, programmatic approach to sound design, wave table synthesis provides a more direct and predictable path to a wide variety of pleasing timbers, since waveforms with known sonic characteristics can be loaded into hardware and recalled on demand.

While the Ensoniq 5503 DOC serves as the architectural reference for this design, the oscillator count, register control, and memory allocation were tweaked to suit a more convenient and intuitive design based on more modern hardware standards. We present WaveSURFER (Wave Synthesis Using Real FPGA EngineeRing).

Our synthesizer, which, to be explicit, is a MIDI controller hooked up to the DE1-SoC with our custom design running on the FPGA to generate the audio, accepts 16-bit signed PCM samples and plays them back at varying pitches. It can play as interesting—or uninteresting—a sound wave that can fit into its wave table.

2 Project Overview

Figure 1 shows a high-level overview of our project. The core idea is simple: on a key press of the piano, the signal from the MIDI keyboard gets parsed and processed in software (on the Linux kernel loaded on the HPS), and these signals get driven into the FPGA via the Lightweight bridge. More processing happens before the signals get sent to the Audio CODEC to produce sound, and the seven-segment LED display on the DE1-SoC board to display the last note played, respectively.

To start our synthesizer, run the following on the DE1-SoC Board (after necessary Quartus build, etc):

```
root@de1-soc:~# # fill samples.txt with wav files of the form 'file.wav,basestep,resolution.'
root@de1-soc:~# python sw/tools/build_wavetable.py samples.txt # gens wavetable.bin
root@de1-soc:~# ./midi_to_fpga wavetable.bin
```

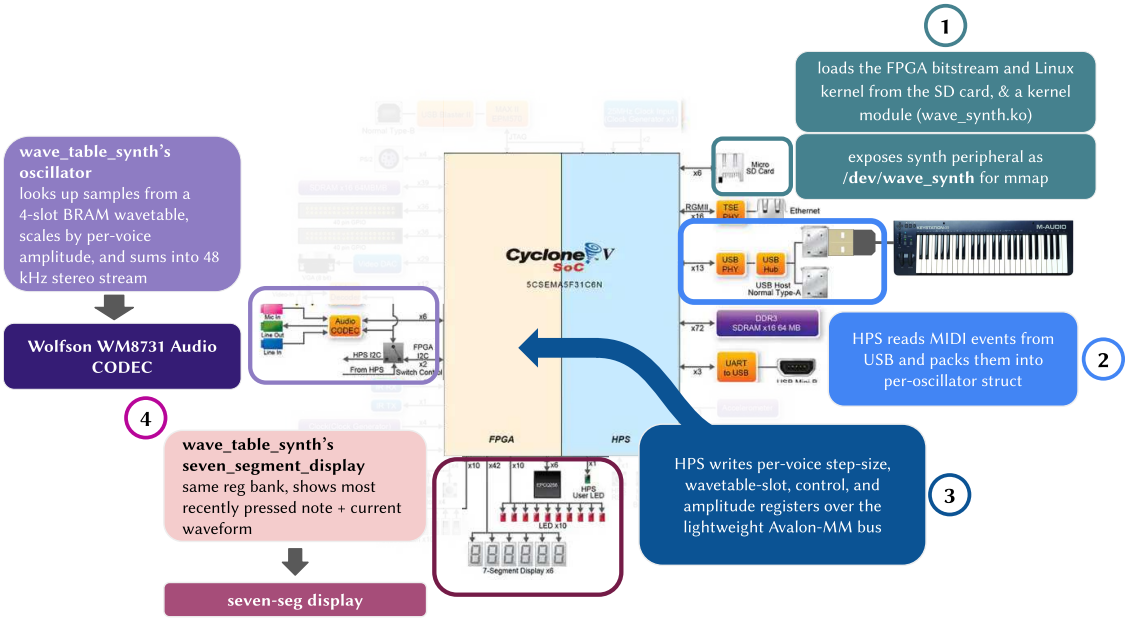

Figure 1: High-level Overview

3 Hardware

3.1 Block Diagram

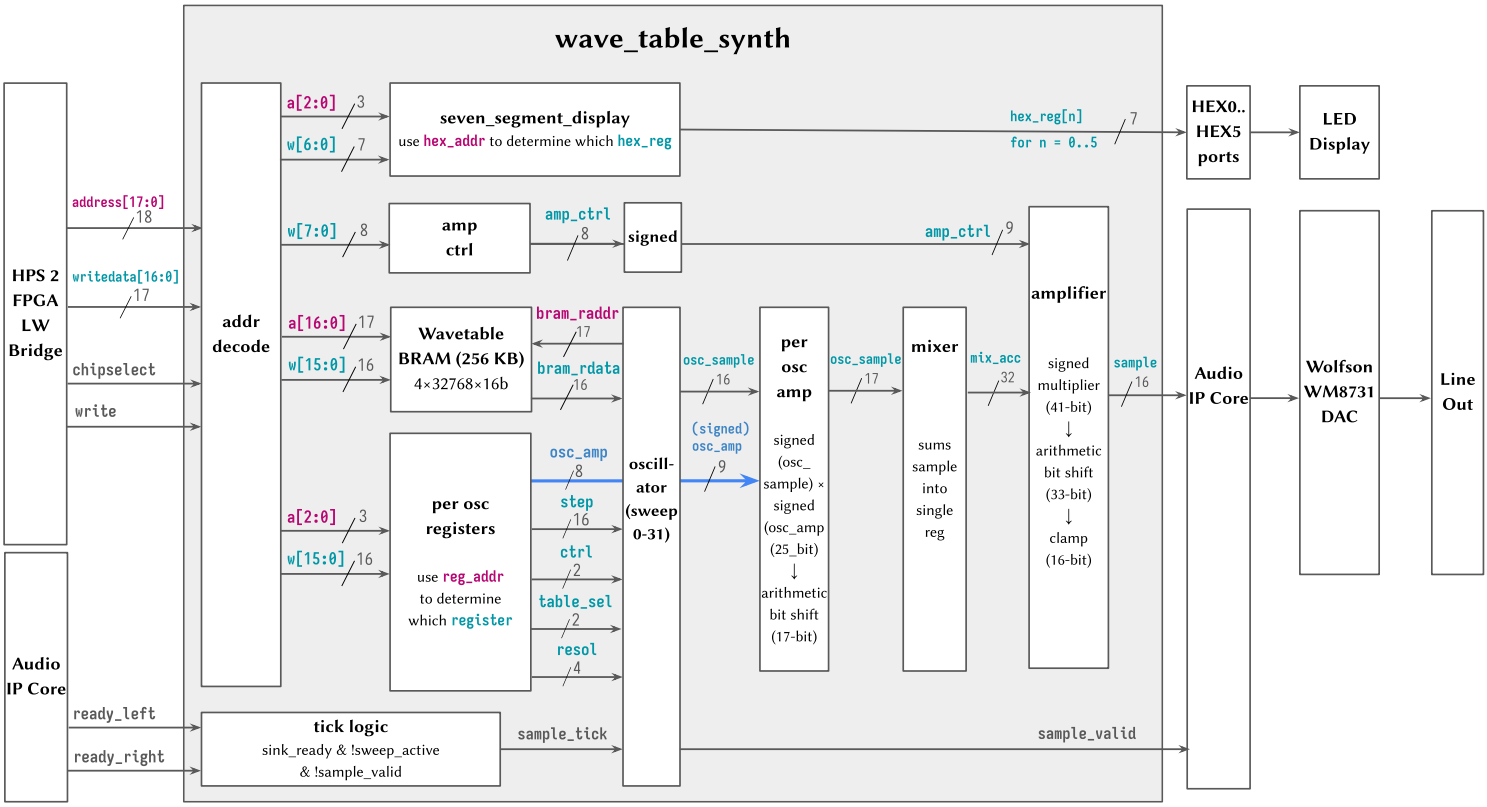

Figure 2: Hardware block diagram, with slash ("/") notation denoting bit size

Top Module `soc_system_top`

Board-level wrapper that instantiates the Platform Designer–generated `soc_system` and drives the physical I/O pins, routing signals between the on-chip IP cores (Audio, Audio PLL, AV Config, HPS) and their external interfaces. Contains no datapath logic. The audio-relevant ports are listed in Table 1. The remaining ports connect standard DE1-SoC board peripherals such as DRAM and DPIO.

Port	Dir	Width	Description
CLOCK_50	in	1	50 MHz system clock
AUD_BCLK	inout	1	Audio bit clock (I^2S)
AUD_DACDAT	out	1	Audio DAC serial data
AUD_DACLRCK	inout	1	DAC left/right clock
AUD_XCK	out	1	Audio master clock (12.288 MHz)
FPGA_I2C_SCLK	out	1	I^2C clock to Wolfson codec
FPGA_I2C_SDAT	inout	1	I^2C data to Wolfson codec

Table 1: soc_system_top audio-relevant port list

Custom Peripheral wave_table_synth

Our main module decodes incoming HPS writes on the lightweight AXI bus and routes them to the appropriate destination (wavetable BRAM, oscillator control registers, or AMP_CTRL, and hex registers). It also drives the Avalon-ST handshake to the Audio IP core, asserting sample_valid when a new mixed sample is available and respecting the ready_left/ready_right backpressure signals from the codec-side FIFOs.

It is a custom synthesizer peripheral instantiated inside soc_system by Platform Designer as wave_table_synth_0. The module implements the Avalon-MM slave interface for HPS register writes and the Avalon-ST source interface for audio output. All oscillator, mixer, and amplifier logic resides within this single module. The mapping region that is used can be found in §3.2. It also instantiates the modules oscillator and seven_segment_display and has inline blocks mixer and amplifier.

On the software side, every FPGA write resolves to a store through the lw_bridge pointer, which lands as an Avalon-MM write on wave_table_synth's address, writedata, write, and chipselect ports, which are among the ports listed in Table 2.

The module (source) also produces one mixed audio sample per Avalon-ST transaction. It asserts sample_valid for exactly one clock cycle when a new 16-bit sample is ready on the sample bus. A transfer to the Audio IP Core (sink) happens on any clock edge where both sample_valid and ready are high simultaneously.

Port	Dir	Width	Description
clk	in	1	50 MHz system clock
reset	in	1	Synchronous active-high reset
address	in	18	Avalon-MM byte address within peripheral
writedata	in	16	Avalon-MM write data
readdata	out	16	Avalon-MM read data
write	in	1	Avalon-MM write strobe
chipselect	in	1	Avalon-MM chip select
ready_left	in	1	Audio IP left FIFO ready (backpressure)
ready_right	in	1	Audio IP right FIFO ready (backpressure)
sample	out	16	Mixed audio sample (Avalon-ST data)
sample_valid	out	1	Avalon-ST valid; pulses one cycle per sample

Table 2: wave_table_synth port list

Wavetable BRAM

stores single-cycle waveforms as a TABLE_SIZE (currently 64KB) × NUM_TABLES (currently 4) × 16-bit array. Configured as a dual-port memory: the write port is used exclusively during initialization for HPS-driven wavetable loading, and the read port is used during playback by the Oscillator Module.

LED Display seven_segment_display

This is a small register file that drives the six on-board HEX displays. It is instantiated inside wave_table_synth and reached by HPS writes whose address decodes to in_hex_registers (per-digit slots in the oscillator region). Each digit holds a 7-bit value that is driven directly onto the corresponding active-low HEX0–HEX5 pins of the DE1-SoC; the software side is responsible for converting the glyph to segment bits. On reset, all six registers are loaded with 7'h7F so every segment is off. Ports are listed in Table 3.

Port	Dir	Width	Description
clk	in	1	System clock
reset	in	1	Synchronous reset; clears all digits to 7'h7F (off)
write	in	1	Write strobe (gated by chipselect && write && in_hex_registers)
addr	in	3	Selects digit 0–5 (writes with addr >= 6 are ignored)
data	in	7	Active-low segment pattern from writedata[6:0]
hex0	out	7	Digit 0 segment outputs
hex1	out	7	Digit 1 segment outputs
hex2	out	7	Digit 2 segment outputs
hex3	out	7	Digit 3 segment outputs
hex4	out	7	Digit 4 segment outputs
hex5	out	7	Digit 5 segment outputs

Table 3: seven_segment_display port list

Phase Accumulator Module oscillator

The oscillator module is a time-multiplexed DDS engine instantiated by wave_table_synth. It generates audio samples for all 32 voices via time-multiplexing (we have an abundance of clock cycles during which to do this between each DAC sample). Receives a step size to iterate through the wave table as calculated via software. A control FSM sequences through a 5-bit voice counter from $0 \rightarrow 31$ on each sample tick; for each voice, it advances the phase accumulator by the configured step size, reads the appropriate sample from BRAM. Maintains 32×24-bit phase accumulators as persistent state across sweeps. The ports are detailed in Table 4.

This is the core functional block in the oscillator by which we generate a range of tones from the wave stored in one of our tables. It maintains a running position (phase_acc) in the wave table and increments it by a fixed step value on each audio sample. By changing the step size, you control the frequency. Larger steps sweep through the table faster (higher pitch), smaller steps sweep slower (lower pitch). The step size is calculated from the desired MIDI note frequency and the table length/sample rate. In the synthesizer, the base_step corresponds to a reference pitch, and higher MIDI notes use larger step values computed via the equal-temperament formula. The Ensoniq chip-inspired frequency control block is shown in Figure 3.

Port	Dir	Width	Description
clk	in	1	System clock
rst	in	1	Synchronous reset
sample_tick	in	1	Begin a new sweep across all 32 voices
step_size	in	32×16	Per-voice phase increments
table_sel	in	32×2	Per-voice wavetable slot select
ctrl	in	32×2	Per-voice control (gate / mode)
osc_resolution	in	32×4	Per-voice phase-increment left-shift
bram_rdata	in	16	Sample read back from wavetable BRAM
bram_raddr	out	17	Address driven to wavetable BRAM
osc_sample	out	16 (signed)	Current voice's sample
osc_idx	out	5	Index of voice on osc_sample (0–31)
osc_valid	out	1	Pulses for each valid per-voice sample
sweep_done	out	1	Pulses when the full 32-voice sweep completes

Table 4: oscillator port list

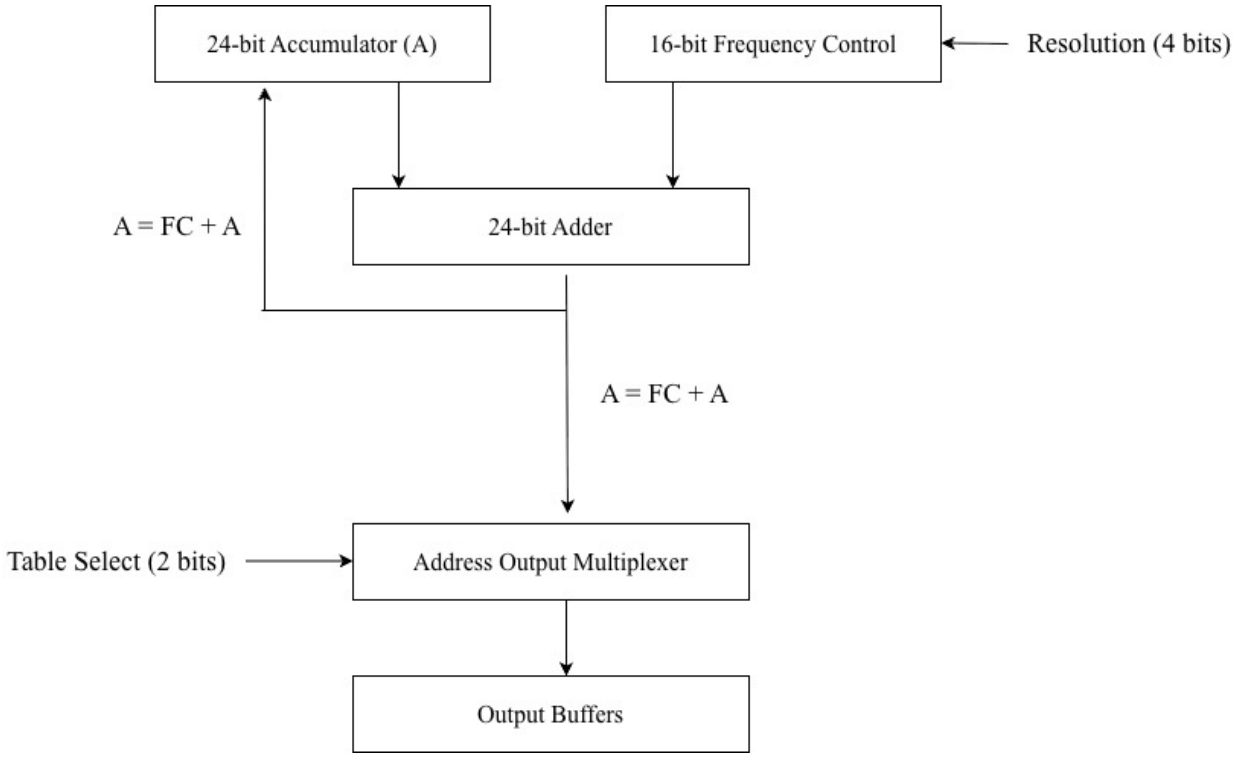

Figure 3: Phase Accumulator Diagram

Per-oscillator amplifier (inline in wave_table_synth)

Before mixing, each voice is scaled by its own gain. The per-oscillator amplifier multiplies the signed 16-bit osc_sample from the oscillator by the unsigned 8-bit gain (zero-extended to 9-bit)

stored in the osc_amp_reg[osc_idx] register (256 levels, written via per-voice slot +3), producing a signed 25-bit product that is arithmetically shifted right by 8 before being handed to the mixer. It uses one DSP multiplier block, time-shared across all 32 voices as the oscillator sweeps. Signals are detailed in Table 5.

Signal	Dir	Width	Description
osc_sample	in	16 (signed)	Current voice sample from oscillator
osc_idx	in	5	Index selecting per-osc gain register
osc_amp_reg[osc_idx]	in	8 (unsigned)	Per-voice gain (slot +3)
osc_sample_shifted	out	25 (signed)	Scaled sample handed to mixer

Table 5: Per-oscillator amplifier signals

Mixer (inline in wave_table_synth)

The mixer is an accumulator inside wave_table_synth that sums the 32 per-voice samples produced by the per-oscillator amplifier into a single signed 32-bit register (mix_acc). It uses an accumulator register that sums each "voice" (oscillator sample output) over a given voice cycle (complete iteration through all oscillators); cleared at the start of each sweep and latched as mixed_sample when the oscillator voice counter wraps. Uses adder blocks. The result is consumed by the global amplifier when the oscillator asserts sweep_done. Signals are detailed in Table 6.

Signal	Dir	Width	Description
clk	in	1	System clock
reset	in	1	Synchronous reset
sample_tick	in	1	Start-of-sweep pulse; clears mix_acc
osc_valid	in	1	Per-voice strobe from oscillator
osc_sample_shifted	in	25 (signed)	Per-osc amplified sample to accumulate
sweep_done	in	1	End-of-sweep pulse from oscillator
mix_acc	out	32 (signed)	Running sum forwarded to amplifier

Table 6: Mixer signals

Amplifier (inline in wave_table_synth)

The global amplifier applies output gain by multiplying mix_acc by the unsigned 8-bit AMP_CTRL register value (256 levels) forwarded to the Audio IP Core. The 41-bit signed product is arithmetically shifted right by 8 and clamped to a signed 16-bit range, producing the final sample forwarded to the Audio IP Core (the same value is used for both L and R channels). It uses one DSP multiplier block. Signals are detailed in Table 7.

Signal	Dir	Width	Description
clk	in	1	System clock
reset	in	1	Synchronous reset
mix_acc	in	32 (signed)	Accumulated sum from mixer
sweep_done	in	1	Latches the clamped result into sample
amp_ctrl_reg	in	8 (unsigned)	Global gain (AMP_CTRL register)
sample	out	16 (signed)	Clamped output to Audio IP Core
sample_valid	out	1	Pulses high when sample is ready

Table 7: Amplifier signals

3.2 Hardware/Software Interface

The WaveSURFER peripheral is mapped into the HPS Lightweight AXI bridge starting at base address 0xFF28_0000, occupying a 512 KB window up through 0xFF2F_FFFF. The peripheral's internal address decoding divides this window into a wavetable BRAM region, a per-oscillator control region, a hex-display control region, and a master amplitude control register. The software writes to the LW bridge via a custom driver detailed in 4.1.2.

Region	Base Address	Address start to end	Size	Purpose
audio_and_video_config_0	0xFF20_0000	0xFF20_0000 – 0xFF20_000F	16 B	Auto-inits WM8731 codec
wave_table_synth	0xFF28_0000	0xFF28_0000 – 0xFF2F_FFFF	512 KB	Entire FPGA peripheral

Table 8: LW AXI bus register map (one level above our peripheral)

Region	Base Address	Address start to end	Size	Purpose
Wavetable BRAM	0xFF28_0000	0xFF28_0000 – 0xFF2B_FFFF	256 KB	Storing single audio wave periods
Per-Oscillator Ctrl	0xFF2C_0000	0xFF2C_0000 – 0xFF2C_01FF	512 B	Controlling oscillators
Hex Ctrl	0xFF2C_0200	0xFF2C_0200 – 0xFF2C_020B	12 B	Setting hex display
Master Amp Ctrl	0xFF2C_0300	0xFF2C_0300 – 0xFF2C_0301	2 B	Setting master amplitude

Table 9: HPS to FPGA register map (wave_table_synth peripheral, mmap-ed over LW AXI bus)

Offset	Name	Width	Purpose
+0x0	STEP	16 b	Phase-accumulator increment (pre-shift)
+0x2	CTRL	16 b	Voice command: IDLE/STOP/START/RESET
+0x4	TABLE	16 b	Wavetable slot select (0–3)
+0x6	AMP	16 b	Per-voice amplitude (envelope output)
+0x8	RESOLUTION	4 b	Left-shift applied to STEP per tick
+0xA	—	3×16 b	Reserved

Table 10: Per-oscillator register slot. Voice v slot base = 0xFF2C_0000 + 0x10·v (32 voices, 16 B each). All registers are write-only.

Value	Name	Effect
0x0000	IDLE	No-op (hold current state)
0x0001	STOP	Freeze phase, mute output
0x0002	START	Begin/resume stepping through table
0x0003	RESET	Zero phase accumulator

Table 11: CTRL register encoding

3.3 Resource Utilization

Our estimated resource allocation is shown in Table 12 and Figure 4.

Structure	Size
Wavetable storage (4 × 32kb × 16)	256 KB
Phase accumulators (32 × 24 b)	768 b
Oscillator control registers (32 × 4 × 16 b)	2,048 b
Mixer accumulator	40 b
AMP_CTRL register	40 b

Table 12: Estimated memory usage. Note 1 of the 4 allocated registers for each oscillator is currently unused (hence the three taken into account)

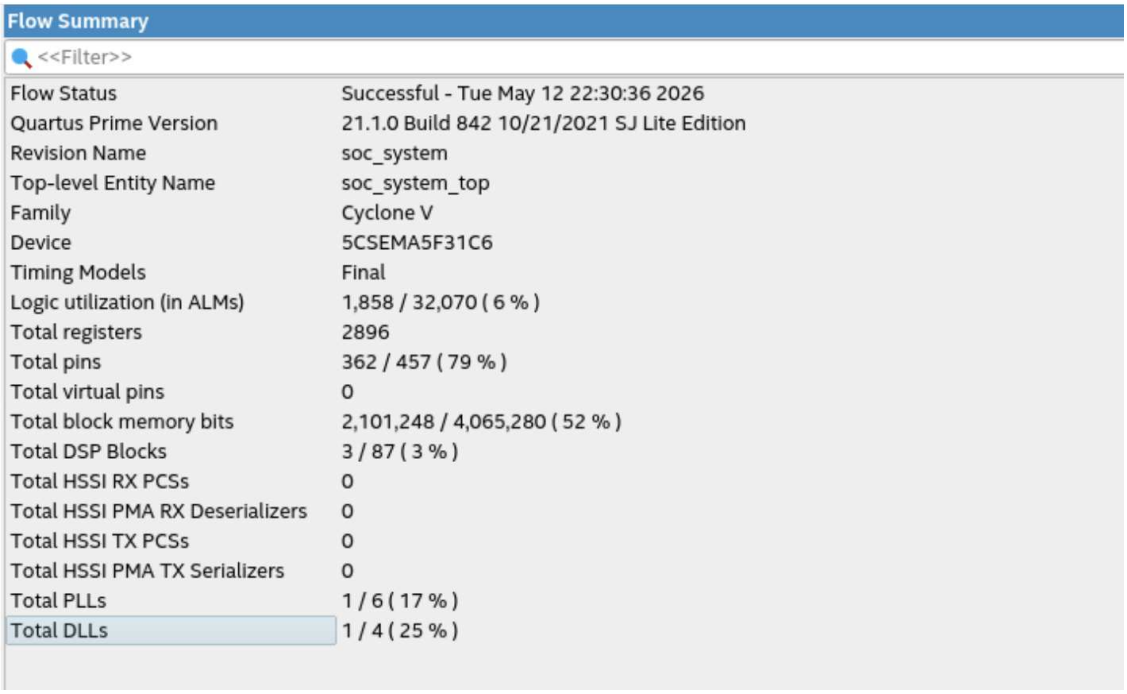

Figure 4: Our by-hand resource estimates via the actual Quartus build report

4 Software

4.1 FPGA Peripheral Device Driver

4.1.1 Original plan

Originally, we had a userspace program that directly reads `"/dev/mem"` and `mmap`s directly to the base of the lightweight bridge. We then write any FPGA updates to `lw_bus->regs`.

```
int fd = open("/dev/mem", O_RDWR | O_SYNC);
void *base = mmap(NULL, WAVE_SYNTH_PERIPHERAL_BYTES,
    PROT_READ | PROT_WRITE,
    MAP_SHARED, fd, 0xFF200000);
lw_bus->regs = (volatile uint16_t *)base;
```

Although this approach worked, it had two main issues:

1. **Safety**: This setup assumes that the user has root (sudo) access, and `"/dev/mem"` exposes all physical memory beyond just the FPGA peripheral, which means a bug in the userspace program can potentially corrupt anything in physical memory
2. **Hardcoded Address**: `0xFF200000` is the physical base of the lightweight bridge on DE1-SoC, which means the program will fail if we either port it to a different board or, say, decide to use the full-weight bridge.

4.1.2 Revised plan

We registered a custom kernel driver to interact with the lightweight HPS-to-FPGA bridge that our custom FPGA peripheral WAVE_TABLE_SYNTH sits on. This allows us to interact with the lightweight bridge as a file called **`"/dev/wave_snyth"`**. The steps taken by the driver are outlined below:

In **`drivers/wave_synth.c`**:

1. ***Declare what the driver handles*** via an `of_device_id` table and a `wave_synth_driver`:

```
static const struct of_device_id wave_synth_of_match[] = {
    { .compatible = "csee4840,wave_table_synth-1.0" },
    {},
};
```

```
static struct platform_driver wave_synth_driver = {
    .driver = {
        .name = WAVE_SYNTH_NAME,
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(wave_synth_of_match),
    },
    .remove = __exit_p(wave_synth_remove),
};
```

- In `soc_system.dts`, the device tree node advertises the same (highlighted) compatible string so the kernel can pair it with this driver:

```
wave_table_synth_0: synth@0x100040000 {
    compatible = "csee4840,wave_table_synth-1.0";
    reg = <0x00000001 0x00040000 0x00040000>;
    clocks = <&clk_0>;
}; //end synth@0x100040000 (wave_table_synth_0)
```

2. ***Match driver to device tree node.*** At boot, the kernel walks the device tree and finds the node whose `compatible` string matches one in our driver's `of_match_table`, binding the two together.

3. ***Probe and claim the peripheral's address range.*** We wire up probing via `platform_driver_probe`, which calls `wave_synth_probe` that runs once the match succeeds and prepares the peripheral for later `mmap`:

   - `misc_register`: exposes WAVE_TABLE_SYNTH as a miscellaneous character device, which gives userspace a `"/dev/wave_synth"` entry to open.
   - `of_address_to_resource`: reads the `reg` property from the matched DT node and fills in a `struct resource` with the peripheral's physical base address and size.
   - `request_mem_region`: reserves that physical range so no other driver can claim it.

4. **Map the peripheral into a process's virtual address space.** `wave_synth_mmap` hooks into the misc device's file ops; when userspace calls `mmap` on `/dev/wave_synth`, the kernel invokes this handler to install page-table entries that point a VMA at the peripheral's physical address range.

- `pgprot_noncached`: marks the mapping as uncached so the CPU never holds register writes in cache — every store reaches the hardware register immediately and in program order.
- `io_remap_pfn_range`: given a page frame number (PFN = physical address ≫ `PAGE_SHIFT`) that the driver has already computed, installs the page-table entries mapping the user VMA to that physical range with the chosen protection.

In **`fpga_bridge.c`**:

1. (In `fpga_bridge.h`) Define a `peripheral` struct holding a file descriptor (`fd`) and a register pointer (`regs`), along with macros such as `WAVE_SYNTH_PERIPHERAL_BYTES`.
2. `fpga_init`: initialize the lightweight-bus (`lw_bus`) peripheral:
   - Obtain `fd` via `open("/dev/wave_synth")`, which selects the device `mmap` will operate on
   - Create a pointer to regs by calling `mmap` with length `WAVE_SYNTH_PERIPHERAL_BYTES` (entire length of the WAVE_TABLE_SYNTH peripheral and offset `0`, so the entire peripheral's register window is mapped starting at the kernel-chosen virtual address returned in `regs`. The remaining flags:
     * `PROT_READ|PROT_WRITE`: allow both reads and writes through the mapping
     * `MAP_SHARED`: writes propagate to both the underlying device (WAVE_TABLE_SYNTH_0) and to any other process that opens and mmaps to `/dev/wave_synth`

4.2 MIDI Device Access via ALSA

4.2.1 Original plan

Pursuant to Lab 2 [4], we originally wrote a libusb-based MIDI receiver in C, where the pipeline similarly read the config descriptor via `libusb_get_config_descriptor`, iterated through the interface classes and subclasses until we found `LIBUSB_CLASS_AUDIO` and `MIDI_STREAMING`, respectively, and obtained a pointer to the MIDI device's endpoint address. Similar to waiting for keyboard inputs in Lab 2, we created an infinite for-loop that reads and parses raw MIDI packets via `libusb_bulk_transfer`.

The original MIDI device we used was the Akai MPK Mini MK3 Keyboard Controller[1]. However, we eventually had to forego the Akai MK3 Controller due to what was likely a power supply

[1]Product page: https://www.akaipro.com/mpk-mini-mk3/

issue of the DE1-SoC Board (Akai MK3 controller is powered over USB). Specifically, we were getting key press dupes when we tried to do more than one key at a time. We debugged by verifying our understanding of the MIDI protocol, the MIDI over USB protocol, the `bulk_transfer` API call for libusb, and trying the controller on different computers where it ended up working fine. We determined that it is probably an issue with the DE1's USB hub, and think it was a power issue (that it could not provide enough to do high speed bulk transfer simultaneously), but we did try a single experiment with a powered usb hub where it also didn't work. We ended up trying a bunch of different controllers and found one that did work with our board, switching to the M-Audio Keystation 49 MK3 49 Key USB/MIDI Controller (hereafter "K49 Controller") which is also powered over usb. [2].

4.2.2 Revised plan

As part of our debugging with the AKAI, we rebuilt the de1 kernel based on the Makefile that had been provided to us in lab3 in hopes of bringing in ALSA drivers (to see if that would provide some additional help in parsing). We had to update the source branch for the socfpga linux fork from altera because the one provided was dead. Beyond that, we had to do some additional flags for cross compilation purposes (we rebuilt the kernel on an x86 machine running ubuntu 24 and kernel v 6.9). Furthermore, we compiled it with USBMON module enabled, which allowed us to do a usb-packet-level trace, where we saw that even at the USB level, the packets we were getting from the AKAI were being duped/lost (looking at the usb packet payloads). Though we did not end up resolving the AKAI issue, one upside of all this work was that we now had ALSA drivers, which simplified our USB/MIDI parsing a good amount.

SOUND[3] and SND_USB_AUDIO kernel modules. The latter module is within the Advanced Linux Sound Architecture (ALSA) framework that provides native support for USB audio devices.

The ALSA rawmidi API lets us open an arbitrary substream of a MIDI device, whereas calling `open()` directly on `"/dev/snd/midiC%iD0"` only ever yields the first substream [2]. It abstracts away the bookkeeping of locating the right card/device pair and picking a substream other than the default. Upon plugging in the K49 Controller into our board's USB port, the following steps happen under the hood:

1. ***Hardware:*** When the USB is plugged in, the Host Controller Driver (HCD) of the DesignWare HS OTG USB 2.0 controller (DWC2) detects the USB device. DWC2 reads the MIDI controller's device descriptor and hands it off to the USB core.

2. ***Software: Linux USB Core*** The USB core then matches with the interface class `USB_CLASS_AUDIO`.

[2]Product page: https://www.m-audio.com/legacy/keystation-49.html
[3]parent gate for SND_*; ALSA tree is invisible without this

The `id_table` matches on the class/subclass fields of `usb_audio_ids`. The match tells USB core to route the process to `usb_audio_driver` called "snd-usb-audio."

- Steps 1 and 2 away from a process similar to finding the USB keyboard by matching the first HID device with a keyboard protocol in Lab 2.

3. ***Software: Linux ALSA driver.*** **sound/usb/card.c:** `usb_audio_probe()` runs:
   - `snd_usb_audio_create`: finds a fresh chip and creates a new chip instance struct `snd_usb_audio` (driver's internal representation of one USB audio device) →
     - `snd_card_new()`: allocates a `snd_card` struct (USB audio interface) →
   - `snd_usb_create_streams()` →
     - `snd_usb_create_stream()` →
     - `snd_usb_midi_v2_create()` →
       - `snd_rawmidi_new()` allocates a `snd_rawmidi` struct →
         - `snd_rawmidi_init()` stores the name pattern "midiC%iD%i" with the two integers representing (card->number, device) respectively
   - `try_to_register_card` →
     - `snd_card_register()`: iterates through all devices attached to card `d` and calls `device_add()` using the pattern from previous step
   - `devtmpfs` creates the node `"/dev/snd/midiC%iD0"` as a character device file, allowing userspace programs to interact with it
4. In **`midi.c`**: `midi_open` scans through card numbers 0–7, trying to open whichever one exists. When open succeeds, the kernel routes that call through the VFS layer to ALSA's rawmidi file operations and enables actual USB endpoint communication

Similar to how our custom driver (§4.1) allows us to open `"/dev/wave_synth"` as a file and write to it, ALSA allows us to treat and communicate with `"/dev/snd/midiC%iD0"` character device in `midi.c` (Step 4).

4.3 Data Flow & Formulas

Our main software block diagram is presented in Figure 5. Our main data structure for storage is `oscillators`, a size-32 array that each contains audio-related fields such as `step_size`, `phase`, `env_amp_q8` (8-bit per-oscillator amplitude).

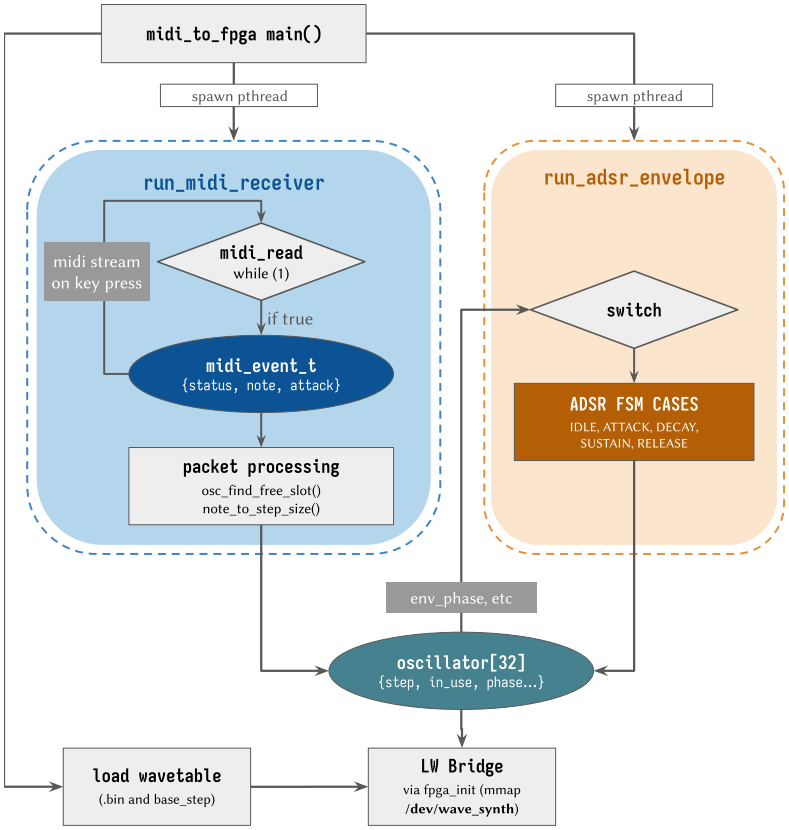

Figure 5: Software Data Flow Diagram

4.3.1 Generate and Load Wave Table

Original plan. We started with writing four generic wave forms (sine, sawtooth, triangle, and square) into a `.bin` file using a Python script. Formulas for each type of wave are mostly from Wolfram [9, 10, 11]. Each sample in the wave table is a signed 16-bit integer. The for-loop first generates a $[-1, 1]$ wave, and the function outputs the wave table by scaling it to the signed 16-bit range by multiplying each sample by 2^{15}, or 32768.

Revised and final plan. We used `build_wavetable.py` to pack up to four mono audio sources into a single .bin. It reads a manifest (samples.txt) listing one entry per line in the form **`<wave_path>,<base_step>,<resolution>`** (e.g., `wavs/banana.wav,1571,0`), where base step is the frequency that, when passed into our equal temperment equation, aligns the tones from the wave with the keys on the keyboard.

For each entry, it opens the WAV, downmixes to mono int16, and resamples it via linear interpolation to exactly `TABLE_SIZE = 32768` samples so every slot occupies a fixed footprint regardless of the original file length. The output begins with a 24-byte header—a "WTSY" magic word, a version number, the per-slot metadata count, and four (base_step, resolution,

reserved) triples—followed by one 65 KB audio block (32768 little-endian int16 samples) per slot listed in the manifest. Padding metadata slots fall back to `DEFAULT_BASE_STEP = 357` and `DEFAULT_RESOLUTION = 8` if not provided. The format is shown in Table 13.

At startup, `midi_to_fpga` calls `load_wavetable_bin()`, which validates the magic, copies the metadata into the CPU-side `wavetable_base_step[]` and `wavetable_slot_resolution[]` arrays, and writes each audio block sample-by-sample into the FPGA's on-chip wavetable SRAM via memory-mapped 16-bit stores over the lightweight HPS-to-FPGA bridge.

4B magic "WTSY"
2B version (=1)
2B num_metadata_slots (=4)
4B per slot × 4 slots:
  2B base_step
  1B resolution
  1B reserved (=0)
	24-byte header

32768 × int16 (LE) | slot 0 audio	wave tables
32768 × int16 (LE) | slot 1 audio	
32768 × int16 (LE) | slot 2 audio	
32768 × int16 (LE) | slot 3 audio	

Table 13: Wave table `.bin`

4.3.2 MIDI Event Thread: `run_midi_receiver`

The following corresponds to the MIDI thread in Fig 5 driven by `midi_read`, a blocking read on the ALSA raw-MIDI file descriptor. The thread sleeps until the kernel hands it a MIDI byte stream from the USB device.

RawMidi to `midi_event_t` The MIDI controller communicates over USB via the RawMidi file descriptor (fd) defined in section 4.2. `midi_read` pulls one complete MIDI Channel-Voice event off the open fd and returns it as a `midi_event_t`. It reads the stream a byte at a time to populate the `midi_event_t` struct. The while-loop skips System Real-Time messages (`0xF8–0xFF`) and clears the running-status cache on any System Common message (`0xF0–0xF7`) to signal a new packet [6].

When the next byte has its high bit set, it is treated as a fresh status byte, latched into both `evt->status` and a `static running_status` so that subsequent "data" packets can implicitly reuse it. Otherwise, the byte is the first data byte of a running-status event and is paired with the cached status. The function then reads the second data byte for all two-data-byte voice messages (Note On/Off) for `evt->note`, with a special case for Program Change (`0xC0`) and Channel Pressure (`0xD0`), which carry only one data byte and so leave

`evt->attack` zeroed. It returns `0` on a successful event and `-1` if any underlying `read()` fails, blocking inside the loop until at least one valid event is assembled.

Byte	Field	Description
0	`status`	MIDI status byte (e.g. `0x90` = note on, `0x80` = note off)
1	`note`	MIDI note number (0–127; middle C = 60)
2	`velocity`	Key velocity (0–127; 0 on note-off)

Table 14: `midi_event_t` struct

On receiving a `NOTE_ON`, the MIDI thread finds a free slot in the `oscillator` array and populates the oscillator's `note` and `step_size`, marks the `phase` as ATTACK, and calculates the 8-bit `amp` for the ADSR envelope. The `step_size` calculation is discussed in detail below.

MIDI Note to frequency multiple. After populating `midi_event_t` struct, the `note` played is extracted from the struct and converted to frequency using the **equal temperament tuning formula** [5], shown in **Equation 1**. In the equation, f_0 is usually set to 440 Hz as the reference frequency (A4 / note 69 in MIDI), and n denotes the *number of semitones* away from the reference. Plugging in these two values, we can see that for note A4 ($n = 0$), $f = 440 \times (2^{1/12})^0 = 440$ Hz. To understand the formula better, consider note A5, which is 12 semitones ($n = 12$) away from A4: $f_{A5} = 440 \times (2^{12/12}) = 880$ Hz.

$$f = f_0 \times (2^{n/12}) \tag{1}$$

Based on the formula, we implemented **Equation 2**, which takes in the MIDI note n to compute scaling factor m, where n is the incoming MIDI note number and r is the per-implementation reference note (we use $r = 60$, corresponding to C4). This is used as a relative transposition used to compute the calibrated step size, detailed in the next section.

$$m(n) = 2^{(n-r)/12} \tag{2}$$

(Original plan) Frequency to step size. After converting to frequency using Equation 1 and A4 as the reference note, we then converted the frequency to step size using **Equation 3**, where f denotes frequency of the note (which we get from Equation 1), t denotes the size of the wave table, and F_s denotes the sampling rate, which is set at 48 kHz for the Audio CODEC.

$$s = \text{round}(f \times \frac{T}{F_s}) \tag{3}$$

Original plan continued. Originally, the number of samples in our wave table is calculated based on the "lowest" frequency perceivable by the human ear, around 20 Hz [8]. Given the default sampling rate of 48kHz, this gives $48000/20 = 2400$ ticks per cycle. The lowest note on the piano is 27Hz, $48000/27 = 1777$ ticks per cycle. Our ideal sample size should be a power of 2 and also between the two aforementioned numbers, which gives us 2048. We used **Equation 4** to calculate the step size.

$$s = \text{round}(f \times \frac{2^t}{F_s} \times 2^x) \tag{4}$$

The variables in Equation 4 are denoted below:

- f is the frequency obtained from Equation 1
- t is the number of bits needed to represent the sample size of the wave table; 2^t is effectively sample size in Eq. 3
  - let T be the sample size of each wave table
  - $t = \log_2(T)$
  - e.g., $t = \log_2(2048) = 11$
- x is the number of bits that can be used to represent the fractional part of a $Qt.x$ fixed-point representation [7].
  - let B be the size of the phase accumulator in HW
  - $x = B - t$

To represent fractional numbers as an integer in the calculated step size, we use fixed-point representation [3]. For example, in our first iteration, the largest step size we need to represent is 2048, which requires at most 11 bits to represent. This leaves us 5 bits to represent the fractional part. The Q format of step size representation can be denoted as in $Q11.x$ format, where $x = \text{len(phase\_acc)} - 11$

(Revised and final plan) Step size via per-slot calibration. In the current implementation, we sidestep the explicit frequency calculation entirely. Each wave table slot ships with a precomputed integer base_step_k stored in the `.bin` header, defined as the phase-increment value that plays slot k at the reference pitch (MIDI note $r = 60$, i.e. C4). At runtime, we obtain the per-voice step size via **Equation 5** by scaling that constant by $m(n)$ from Equation 2:

$$s(n, k) = \text{round}(\text{base\_step}_k \cdot m(n)) \tag{5}$$

The result is clamped to 16 bits to fit the FPGA's phase-increment register. The variables in Equation 5 are:

- n is the incoming MIDI note number
- k is the index of the active wave table slot ($0 \leq k <$ `NUM_TABLE_SLOTS`)
- base_step_k is the per-slot calibration constant, loaded once at startup from the `.bin` header produced offline by `tools/build_wavetable.py`
- $m(n)$ is the equal-temperament scaling factor from Equation 2

The current table size is $T = 2^{15} = 32768$ samples, larger than the originally planned 2048. The fractional-bit scaling 2^x that was previously defined in Equation 4 is now absorbed into base_step_k during offline preparation; the corresponding HW phase-increment shift is configured separately per slot via a `resolution` field also stored in the header. This factoring has three advantages over the original frequency-based pipeline:

- **No runtime division or table-size arithmetic.** The runtime needs only a single `pow(2.0, ·)` call per note-on event to compute $m(n)$; the rest of the pipeline is integer multiplication.
- **Source-pitch agnostic.** Slots may hold single-cycle waveforms (e.g., a sine table) or multi-cycle recorded samples (e.g., an organ patch) interchangeably, because base_step_k absorbs whatever scaling the source audio requires to play at the reference pitch.
- **Hardware decoupling.** The runtime is unaware of the phase accumulator width B or the table size T; both are encoded into the calibration constant and the resolution field by the offline build tool.

The choice of $r = 60$ instead of the tuning-standard $r = 69$ ($A4 = 440\,\text{Hz}$) is a chosen convention; any reference note yields identical audio output provided base_step_k is recalibrated consistently. C4 was chosen because (1) MIDI octave boundaries fall on multiples of 12, making $(n - r)/12$ a clean integer at every octave boundary, and (2) it matches the anchor pitch used by most sampled-instrument libraries.

4.3.3 ADSR Envelope Thread: `run_adsr_envelope`

The ADSR thread in Fig 5 wakes up every 1 ms off the Linux `CLOCK_MONOTONIC` of the HPS CPU. It functions as a finite state machine (FSM) that loops through five different states: Idle, Attack, Decay, Sustain, and Release. The phases update the `env_amp_q8` field of each oscillator, which is used by the per-oscillator amplifier (inline of `wave_synth`). Only two phase changes are driven by an MIDI event: Idle → Attack on `NOTE_ON` and Sustain → Release on `NOTE_OFF`.

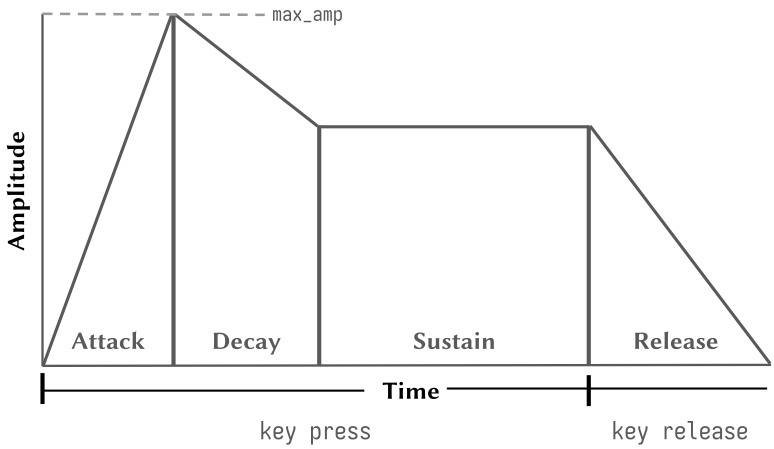

Figure 6: ADSR Diagram

Figure 6 shows the phases of ADSR, which allows a sound to smoothly evolve over time, producing a more fluid, natural sound as opposed to punchy drum hits. Originally, `attack_step`, `peak`, `decay_step`, `sustain`, and `release_step` are global variables, but we decided to scale them using the velocity factor v we calculated based on per-oscillator `attack` a, shown in Equation 6.

$$v = \frac{a}{128} \tag{6}$$

Idle is the default state, and sets `env_amp_q8` to 0. On `note_on`, the `phase` becomes Attack.

Attack is the first stage when the sound materializes. At this state, we add an `attack_step` to the current amplitude until it reaches the `peak`. On a cycle that may exceed peak after an `attack_step`, the amplitude defaults to `peak`. Once reached, the state goes to Decay.

Decay is when the voice slowly settles into the sustain stage. At this state, we subtract `decay_step` from the current amplitude until it reaches the `sustain`. Once reached, the state goes to Sustain.

Sustain holds the note at `sustain` while the key is pressed. It switches to Release on `NOTE_OFF`.

Release is the last stage. At this state, we subtract `release_step` from the current amplitude until it reaches 0. Once reached, the state goes back to Idle. This is the only state that also clears out all the cached per-oscillator fields.

5 Contributions and Reflections

5.1 Who did what

- **Harry:** Worked mostly on hardware. Spent a lot of time trying to understand the core oscillator / phase accumulator logic, and made a lot of mistakes. Spent time debugging and setting up the MIDI controller. Finished device driver and hooked up mmap call for the peripheral. Rebuilt kernel to add ALSA drivers. Played around with lots of samples. Did core software representation of synth (oscillator structs, keeping track of which notes were on and off, and sending corresponding FPGA signals). Did the platform designer set up the IP cores (wiring together the Audio clock and the Wolfson chip, wiring the AV config core to the audio core, setting up the audio core).

- **Sunny:** Built out the initial software that parses MIDI input via the USB interface. Refactored and contributed to (along with Opalina) the original `"/dev/mem"`-based MIDI to FPGA bridge (before switching to the device driver). Implemented the fixed-point representation for step size and phase accumulator. Worked on new register maps. Created the newest iteration of the HW/SW block diagrams, register maps, and ”translated” code into writing (i.e., this final report). Developed the per-oscillator amplifier for ADSR on both SW and HW. Added the seven-segment display to display the current waveform and the last note on (half of it is obsolete since we have funner wave forms now). Also messed around with fun WAVE file samples with Harry.

- **Opalina:**

    * Worked primarily on the software and interface aspects of the design
    * Worked with Sunny on the Midi parsing, and FPGA bridge
    * Wrote out the software to hardware mapping of the wavetable, and iterated on different methodologies of creating the mapping before deciding on the custom driver

5.2 Lessons Learned

Learned about digital representation of sound waves: PCM, sample rates, and representing samples at different precisions.

Learned about fixed-point precision for representing fractions in hardware (phase accumulator logic).

Learned a TON about device drivers and interfacing with peripherals from Linux: How to register a device driver, how the Kernel interfaces with peripherals (at least one way, the AXI protocol), how the Kernel perceives devices (you decide, for better or for worse, based on the tools provided in the kernel). Furthermore, how to set up registers in a device sensibly for control from software (such as our oscillator control region).

5.3 Advice for Future Projects

Make a software simulator. We relied pretty heavily on AI tools to generate working code after going through the designs ourselves, which had a high success rate but made a lot of designs pretty rigid. It would have been worth our weight in gold to make a software simulation of our logic before doing it in hardware

Pick a project you're interested in/want to have in your hands at the end of the semester: it was easier to find time to work on the project because it was something that we found fun to use and play with. It presented a lot of new challenges for us (learning about some basic DSP, fundamentals of music theory, and two of us had no hardware background), but there was the motivation of having a fun device to use at the end.

Use Lab 3 as a starting point for the project. It took us a week or two to wrap our heads around the fact that Lab 3 provides a pretty good template for a project, namely the interface for connecting a hardware peripheral to software.

Find a project that someone else did in the past and use that as a template. We had a pretty good reference from a few semesters back, and arguably would have been even better off in the project had we just implemented their entire project first and then built off it.

6 Code

Code Appendix

Hardware (SystemVerilog)

oscillator.sv

```
module oscillator(
    input logic clk,
    input logic rst,
    input logic sample_tick,
    input logic [15:0] step_size[0:31],
    input logic [1:0] table_sel[0:31],
    input logic [1:0] ctrl [0:31],
    input logic [3:0] osc_resolution [0:31],
    input logic [15:0] bram_rdata,
    output logic [16:0] bram_raddr,
    output logic [15:0] osc_sample,
    output logic [4:0] osc_idx,
    output logic osc_valid,
    output logic sweep_done
);
    typedef enum logic { IDLE, RUNNING } state_t;
    state_t state;
    logic [5:0] step;


    logic [23:0] phase [0:31];

    wire [4:0] osc_to_read = step[4:0] - 5'd2;

    always_ff @(posedge clk) begin
        if (rst) begin
            state <= IDLE;
            step <= 6'd0;
            osc_valid <= 1'b0;
            sweep_done <= 1'b0;
            osc_sample <= 16'h0;
            osc_idx <= 5'h0;
            bram_raddr <= 17'h0;
            for (int i = 0; i < 32; i++) begin
                phase[i] <= 24'h0;
            end
```

```
37          end else begin
38              osc_valid <= 1'b0;
39              sweep_done <= 1'b0;
40
41              unique case (state)
42                  IDLE: begin
43                      if(sample_tick) begin
44                          state <= RUNNING;
45                          step <= 6'd0;
46                      end
47                  end
48
49                  RUNNING: begin
50                      if(step < 6'd32) begin
51                          bram_raddr <= { table_sel[step[4:0]], phase[step[4:0]][23:9]};
52                      end
53
54                      if (step > 6'd1) begin
55                          osc_idx <= osc_to_read;
56                          osc_valid <= 1'b1;
57                          if(ctrl[osc_to_read] == 2'b10) begin
58                              osc_sample <= bram_rdata;
59                          end else begin
60                              osc_sample <= 16'h0;
61                          end
62
63                      if (ctrl[osc_to_read] == 2'b11) begin
64                              phase[osc_to_read] <= 24'h0;
65                          end else if (ctrl[osc_to_read] == 2'b10) begin
66                              phase[osc_to_read] <= phase[osc_to_read]
67                                  + 24'(({8'h0, step_size[osc_to_read]} <<
                                    ↪  osc_resolution[osc_to_read]));
68                          end
69                      end
70
71                      if (step == 6'd33) begin
72                          sweep_done <= 1'b1;
73                          state      <= IDLE;
74                      end else begin
75                          step <= step + 6'd1;
76                      end
77                  end
78              endcase
79          end
```

```
    end

endmodule
```

seven_segment_display.sv

```
module seven_segment_display (
    input  logic        clk,
    input  logic        reset,
    input  logic        write,
    input  logic [2:0]  addr,
    input  logic [6:0]  data,
    output logic [6:0]  hex0,
    output logic [6:0]  hex1,
    output logic [6:0]  hex2,
    output logic [6:0]  hex3,
    output logic [6:0]  hex4,
    output logic [6:0]  hex5
);

    logic [6:0] hex_reg [0:5];

    always_ff @(posedge clk) begin
        if (reset) begin
            for (int i = 0; i < 6; i++) begin
                hex_reg[i] <= 7'h7F;
            end
        end else if (write && addr < 3'd6) begin
            hex_reg[addr] <= data;
        end
    end

    assign hex0 = hex_reg[0];
    assign hex1 = hex_reg[1];
    assign hex2 = hex_reg[2];
    assign hex3 = hex_reg[3];
    assign hex4 = hex_reg[4];
    assign hex5 = hex_reg[5];

endmodule
```

soc_system_top.sv

```
// ============================================================
```

```
// Copyright (c) 2013 by Terasic Technologies Inc.
// ====================================================================
//
// Modified 2019 by Stephen A. Edwards
//
// Permission:
//
//   Terasic grants permission to use and modify this code for use
//   in synthesis for all Terasic Development Boards and Altera
//   Development Kits made by Terasic.  Other use of this code,
//   including the selling ,duplication, or modification of any
//   portion is strictly prohibited.
//
// Disclaimer:
//
//   This VHDL/Verilog or C/C++ source code is intended as a design
//   reference which illustrates how these types of functions can be
//   implemented.  It is the user's responsibility to verify their
//   design for consistency and functionality through the use of
//   formal verification methods.  Terasic provides no warranty
//   regarding the use or functionality of this code.
//
// ====================================================================
//
//  Terasic Technologies Inc

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

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

 ///////// AUD /////////
 input         AUD_ADCDAT,
 inout         AUD_ADCLRCK,
 inout         AUD_BCLK,
 output        AUD_DACDAT,
```

```
46  inout         AUD_DACLRCK,
47  output        AUD_XCK,
48
49  ///////// CLOCK2 /////////
50  input         CLOCK2_50,
51
52  ///////// CLOCK3 /////////
53  input         CLOCK3_50,
54
55  ///////// CLOCK4 /////////
56  input         CLOCK4_50,
57
58  ///////// CLOCK /////////
59  input         CLOCK_50,
60
61  ///////// DRAM /////////
62  output [12:0] DRAM_ADDR,
63  output [1:0]  DRAM_BA,
64  output        DRAM_CAS_N,
65  output        DRAM_CKE,
66  output        DRAM_CLK,
67  output        DRAM_CS_N,
68  inout [15:0]  DRAM_DQ,
69  output        DRAM_LDQM,
70  output        DRAM_RAS_N,
71  output        DRAM_UDQM,
72  output        DRAM_WE_N,
73
74  ///////// FAN /////////
75  output        FAN_CTRL,
76
77  ///////// FPGA /////////
78  output        FPGA_I2C_SCLK,
79  inout         FPGA_I2C_SDAT,
80
81  ///////// GPIO /////////
82  inout [35:0]  GPIO_0,
83  inout [35:0]  GPIO_1,
84
85  ///////// HEX0 /////////
86  output [6:0]  HEX0,
87
88  ///////// HEX1 /////////
89  output [6:0]  HEX1,
```

```
90
91  ///////// HEX2 /////////
92  output [6:0]  HEX2,
93
94  ///////// HEX3 /////////
95  output [6:0]  HEX3,
96
97  ///////// HEX4 /////////
98  output [6:0]  HEX4,
99
100 ///////// HEX5 /////////
101 output [6:0]  HEX5,
102
103 ///////// HPS /////////
104 inout         HPS_CONV_USB_N,
105 output [14:0] HPS_DDR3_ADDR,
106 output [2:0]  HPS_DDR3_BA,
107 output        HPS_DDR3_CAS_N,
108 output        HPS_DDR3_CKE,
109 output        HPS_DDR3_CK_N,
110 output        HPS_DDR3_CK_P,
111 output        HPS_DDR3_CS_N,
112 output [3:0]  HPS_DDR3_DM,
113 inout [31:0]  HPS_DDR3_DQ,
114 inout [3:0]   HPS_DDR3_DQS_N,
115 inout [3:0]   HPS_DDR3_DQS_P,
116 output        HPS_DDR3_ODT,
117 output        HPS_DDR3_RAS_N,
118 output        HPS_DDR3_RESET_N,
119 input         HPS_DDR3_RZQ,
120 output        HPS_DDR3_WE_N,
121 output        HPS_ENET_GTX_CLK,
122 inout         HPS_ENET_INT_N,
123 output        HPS_ENET_MDC,
124 inout         HPS_ENET_MDIO,
125 input         HPS_ENET_RX_CLK,
126 input [3:0]   HPS_ENET_RX_DATA,
127 input         HPS_ENET_RX_DV,
128 output [3:0]  HPS_ENET_TX_DATA,
129 output        HPS_ENET_TX_EN,
130 inout         HPS_GSENSOR_INT,
131 inout         HPS_I2C1_SCLK,
132 inout         HPS_I2C1_SDAT,
133 inout         HPS_I2C2_SCLK,
```

```
134   inout         HPS_I2C2_SDAT,
135   inout         HPS_I2C_CONTROL,
136   inout         HPS_KEY,
137   inout         HPS_LED,
138   inout         HPS_LTC_GPIO,
139   output        HPS_SD_CLK,
140   inout         HPS_SD_CMD,
141   inout [3:0]   HPS_SD_DATA,
142   output        HPS_SPIM_CLK,
143   input         HPS_SPIM_MISO,
144   output        HPS_SPIM_MOSI,
145   inout         HPS_SPIM_SS,
146   input         HPS_UART_RX,
147   output        HPS_UART_TX,
148   input         HPS_USB_CLKOUT,
149   inout [7:0]   HPS_USB_DATA,
150   input         HPS_USB_DIR,
151   input         HPS_USB_NXT,
152   output        HPS_USB_STP,
153
154   ///////// IRDA /////////
155   input         IRDA_RXD,
156   output        IRDA_TXD,
157
158   ///////// KEY /////////
159   input [3:0]   KEY,
160
161   ///////// LEDR /////////
162   output [9:0]  LEDR,
163
164   ///////// PS2 /////////
165   inout         PS2_CLK,
166   inout         PS2_CLK2,
167   inout         PS2_DAT,
168   inout         PS2_DAT2,
169
170   ///////// SW /////////
171   input [9:0]   SW,
172
173   ///////// TD /////////
174   input         TD_CLK27,
175   input [7:0]   TD_DATA,
176   input         TD_HS,
177   output        TD_RESET_N,
```

```
 input         TD_VS,


 ///////// VGA /////////
 output [7:0]  VGA_B,
 output        VGA_BLANK_N,
 output        VGA_CLK,
 output [7:0]  VGA_G,
 output        VGA_HS,
 output [7:0]  VGA_R,
 output        VGA_SYNC_N,
 output        VGA_VS
);

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

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

     .hps_hps_io_emac1_inst_TX_CLK ( HPS_ENET_GTX_CLK ),
     .hps_hps_io_emac1_inst_TXD0   ( HPS_ENET_TX_DATA[0] ),
     .hps_hps_io_emac1_inst_TXD1   ( HPS_ENET_TX_DATA[1] ),
     .hps_hps_io_emac1_inst_TXD2   ( HPS_ENET_TX_DATA[2] ),
     .hps_hps_io_emac1_inst_TXD3   ( HPS_ENET_TX_DATA[3] ),
     .hps_hps_io_emac1_inst_RXD0   ( HPS_ENET_RX_DATA[0] ),
     .hps_hps_io_emac1_inst_MDIO   ( HPS_ENET_MDIO  ),
     .hps_hps_io_emac1_inst_MDC    ( HPS_ENET_MDC   ),
     .hps_hps_io_emac1_inst_RX_CTL ( HPS_ENET_RX_DV ),
```

```
222         .hps_hps_io_emac1_inst_TX_CTL ( HPS_ENET_TX_EN ),
223         .hps_hps_io_emac1_inst_RX_CLK ( HPS_ENET_RX_CLK ),
224         .hps_hps_io_emac1_inst_RXD1   ( HPS_ENET_RX_DATA[1]  ),
225         .hps_hps_io_emac1_inst_RXD2   ( HPS_ENET_RX_DATA[2]  ),
226         .hps_hps_io_emac1_inst_RXD3   ( HPS_ENET_RX_DATA[3]  ),
227
228         .hps_hps_io_sdio_inst_CMD     ( HPS_SD_CMD          ),
229         .hps_hps_io_sdio_inst_D0      ( HPS_SD_DATA[0]      ),
230         .hps_hps_io_sdio_inst_D1      ( HPS_SD_DATA[1]      ),
231         .hps_hps_io_sdio_inst_CLK     ( HPS_SD_CLK          ),
232         .hps_hps_io_sdio_inst_D2      ( HPS_SD_DATA[2]      ),
233         .hps_hps_io_sdio_inst_D3      ( HPS_SD_DATA[3]      ),
234
235         .hps_hps_io_usb1_inst_D0      ( HPS_USB_DATA[0]     ),
236         .hps_hps_io_usb1_inst_D1      ( HPS_USB_DATA[1]     ),
237         .hps_hps_io_usb1_inst_D2      ( HPS_USB_DATA[2]     ),
238         .hps_hps_io_usb1_inst_D3      ( HPS_USB_DATA[3]     ),
239         .hps_hps_io_usb1_inst_D4      ( HPS_USB_DATA[4]     ),
240         .hps_hps_io_usb1_inst_D5      ( HPS_USB_DATA[5]     ),
241         .hps_hps_io_usb1_inst_D6      ( HPS_USB_DATA[6]     ),
242         .hps_hps_io_usb1_inst_D7      ( HPS_USB_DATA[7]     ),
243         .hps_hps_io_usb1_inst_CLK     ( HPS_USB_CLKOUT      ),
244         .hps_hps_io_usb1_inst_STP     ( HPS_USB_STP         ),
245         .hps_hps_io_usb1_inst_DIR     ( HPS_USB_DIR         ),
246         .hps_hps_io_usb1_inst_NXT     ( HPS_USB_NXT         ),
247
248         .hps_hps_io_spim1_inst_CLK    ( HPS_SPIM_CLK  ),
249         .hps_hps_io_spim1_inst_MOSI   ( HPS_SPIM_MOSI ),
250         .hps_hps_io_spim1_inst_MISO   ( HPS_SPIM_MISO ),
251         .hps_hps_io_spim1_inst_SS0    ( HPS_SPIM_SS   ),
252
253         .hps_hps_io_uart0_inst_RX     ( HPS_UART_RX     ),
254         .hps_hps_io_uart0_inst_TX     ( HPS_UART_TX     ),
255
256         .hps_hps_io_i2c0_inst_SDA     ( HPS_I2C1_SDAT     ),
257         .hps_hps_io_i2c0_inst_SCL     ( HPS_I2C1_SCLK     ),
258
259         .hps_hps_io_i2c1_inst_SDA     ( HPS_I2C2_SDAT     ),
260         .hps_hps_io_i2c1_inst_SCL     ( HPS_I2C2_SCLK     ),
261
262         .hps_hps_io_gpio_inst_GPIO09  ( HPS_CONV_USB_N ),
263         .hps_hps_io_gpio_inst_GPIO35  ( HPS_ENET_INT_N ),
264         .hps_hps_io_gpio_inst_GPIO40  ( HPS_LTC_GPIO ),
265
```

```
266       .hps_hps_io_gpio_inst_GPIO48  ( HPS_I2C_CONTROL ),
267       .hps_hps_io_gpio_inst_GPIO53  ( HPS_LED ),
268       .hps_hps_io_gpio_inst_GPIO54  ( HPS_KEY ),
269       .hps_hps_io_gpio_inst_GPIO61  ( HPS_GSENSOR_INT ),
270
271
272     .audio_0_avalon_left_channel_sink_data(sample),
273     .audio_0_avalon_left_channel_sink_valid(sample_valid),
274     .audio_0_avalon_left_channel_sink_ready(ready_left),
275     .audio_0_avalon_right_channel_sink_data(sample),
276     .audio_0_avalon_right_channel_sink_valid(sample_valid),
277     .audio_0_avalon_right_channel_sink_ready(ready_right),
278
279     .audio_0_external_interface_BCLK(AUD_BCLK),
280     .audio_0_external_interface_DACDAT(AUD_DACDAT),
281     .audio_0_external_interface_DACLRCK(AUD_DACLRCK),
282
283     .audio_and_video_config_0_external_interface_SDAT(FPGA_I2C_SDAT),
284     .audio_and_video_config_0_external_interface_SCLK(FPGA_I2C_SCLK),
285
286     .audio_pll_0_audio_clk_clk(AUD_XCK),
287
288     .wave_table_synth_0_wave_table_synth_sample      ( sample       ),
289     .wave_table_synth_0_wave_table_synth_sample_valid ( sample_valid ),
290     .wave_table_synth_0_wave_table_synth_ready_left   ( ready_left   ),
291     .wave_table_synth_0_wave_table_synth_ready_right  ( ready_right  ),
292
293     .hex_hex0 ( HEX0 ),
294     .hex_hex1 ( HEX1 ),
295     .hex_hex2 ( HEX2 ),
296     .hex_hex3 ( HEX3 ),
297     .hex_hex4 ( HEX4 ),
298     .hex_hex5 ( HEX5 ),
299   );
300
301    // The following quiet the "no driver" warnings for output
302    // pins and should be removed if you use any of these peripherals
303
304   logic ready_left;
305   logic ready_right;
306   logic [15:0] sample;
307   logic sample_valid;
308
309    assign ADC_CS_N = SW[1] ? SW[0] : 1'bZ;
```

```
    assign ADC_DIN = SW[0];
    assign ADC_SCLK = SW[0];


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

    assign FAN_CTRL = SW[0];


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

    // HEX0..HEX5 driven by wave_table_synth_0 hex conduit

    assign IRDA_TXD = SW[0];

    assign LEDR = {sample[15:7], sample_valid};

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

    assign TD_RESET_N = SW[0];



endmodule
```

wave_table_synth.sv

```
module wave_table_synth (
    input  logic        clk,
    input logic        reset,
    input logic [15:0]  writedata,
  output logic [15:0]  readdata,
    input logic        write,
    input logic        chipselect,
    input logic [17:0]  address,
    input  logic        ready_left,
```

```
    input  logic        ready_right,
    output logic [15:0] sample,
    output logic        sample_valid,
    output logic [6:0]  hex0,
    output logic [6:0]  hex1,
    output logic [6:0]  hex2,
    output logic [6:0]  hex3,
    output logic [6:0]  hex4,
    output logic [6:0]  hex5
);


    wire in_wavetable = (address[17] == 1'b0);
    wire in_osc_region = (address[17] == 1'b1);
    /* per-voice block: 32 voices x 8 16-bit slots (5 used, 3 reserved) = 0x00000..0x000FF
       per-voice slots: +0 step, +1 ctrl, +2 table, +3 amp, +4 resolution */
    wire in_osc_registers = in_osc_region && (address[16:8] == 9'h000);
    wire in_hex_registers = in_osc_region && (address[16:7] == 10'h002);
    wire is_amp_ctrl = in_osc_region && (address[16:0] == 17'h00180);

    wire [4:0] osc_addr = address[7:3];
    wire [2:0] reg_addr = address[2:0];
    wire [2:0] hex_addr = address[2:0];

    seven_segment_display u_seven_segment (
        .clk    (clk),
        .reset  (reset),
        .write  (chipselect && write && in_hex_registers),
        .addr   (hex_addr),
        .data   (writedata[6:0]),
        .hex0   (hex0),
        .hex1   (hex1),
        .hex2   (hex2),
        .hex3   (hex3),
        .hex4   (hex4),
        .hex5   (hex5)
    );


    (* ramstyle = "M10K" *) logic [15:0] wavetable_bram [0:131071];


    logic [16:0] bram_raddr;
    logic [15:0] bram_rdata;
```

```
54
55      always_ff @(posedge clk) begin
56          if (chipselect && write && in_wavetable) begin
57              wavetable_bram[address[16:0]] <= writedata;
58          end
59          bram_rdata <= wavetable_bram[bram_raddr];
60      end
61
62      logic [15:0] step_size_reg [0:31];
63      logic [1:0] ctrl_reg [0:31];
64      logic [1:0] table_sel_reg [0:31];
65      logic [7:0] osc_amp_reg [0:31]; /* per oscillator amp control */
66      logic [7:0] amp_ctrl_reg;
67      logic [3:0] osc_resolution_reg [0:31]; /* per-oscillator step_size resolution (left-shift
        ↪  applied to phase increment) */
68
69      always_ff @(posedge clk) begin
70          if (reset) begin
71              for(int i = 0; i < 32; i++) begin
72                  step_size_reg[i] <= 16'h0;
73                  ctrl_reg[i] <= 2'b00;
74                  table_sel_reg[i] <= 2'h0;
75                  osc_amp_reg[i] <= 8'h0;
76                  osc_resolution_reg[i] <= 4'd8;
77              end
78              amp_ctrl_reg <= 8'hFF;
79          end else if (chipselect && write && in_osc_registers) begin
80              case (reg_addr)
81                  3'd0: step_size_reg[osc_addr] <= writedata;
82                  3'd1: ctrl_reg[osc_addr] <= writedata[1:0];
83                  3'd2: table_sel_reg[osc_addr] <= writedata[1:0];
84                  3'd3: osc_amp_reg[osc_addr] <= writedata[7:0];
85                  3'd4: osc_resolution_reg[osc_addr] <= writedata[3:0];
86                  default: ; /* slots +5..+7 reserved */
87              endcase
88          end else if (chipselect && write && is_amp_ctrl) begin
89              amp_ctrl_reg <= writedata[7:0];
90          end
91      end
92
93
94
95      wire sink_ready = ready_left & ready_right;
96
```

```
97      logic sweep_active;
98      logic sweep_done;
99
100     wire sample_tick = sink_ready && !sweep_active && !sample_valid;
101
102     always_ff @(posedge clk) begin
103         if(reset) begin
104             sweep_active <= 1'b0;
105         end else if (sample_tick) begin
106             sweep_active <= 1'b1;
107         end else if (sweep_done) begin
108             sweep_active <= 1'b0;
109         end
110     end
111
112     logic [15:0] osc_sample;
113     logic [4:0]  osc_idx;
114     logic        osc_valid;
115
116     oscillator u_oscillator (
117         .clk          (clk),
118         .rst          (reset),
119         .sample_tick  (sample_tick),
120         .step_size    (step_size_reg),
121         .table_sel    (table_sel_reg),
122         .ctrl         (ctrl_reg),
123         .osc_resolution    (osc_resolution_reg),
124         .bram_rdata   (bram_rdata),
125         .bram_raddr   (bram_raddr),
126         .osc_sample (osc_sample),
127         .osc_idx    (osc_idx),
128         .osc_valid  (osc_valid),
129         .sweep_done   (sweep_done)
130     );
131
132     logic signed [31:0] mix_acc;
133     logic signed [40:0] mix_scaled;
134     logic signed [32:0] mix_shifted;
135     assign mix_scaled  = mix_acc * $signed({1'b0, amp_ctrl_reg}); /* applied after oscillator
        ↪  sum */
136     assign mix_shifted = mix_scaled >>> 8;
137
138     wire signed [24:0] osc_sample_scaled;
139     wire signed [24:0] osc_sample_shifted;
```

```
    assign osc_sample_scaled = $signed({1'b0, osc_amp_reg[osc_idx]}) * $signed(osc_sample);
    assign osc_sample_shifted = osc_sample_scaled >>> 8;

    always_ff @(posedge clk) begin
        if(reset) begin
            mix_acc <= '0;
            sample <= 16'h0;
        end else begin
            if (sample_tick) begin
                mix_acc <= '0;
            end else if (osc_valid) begin
                //sign extension nonsense that i don't understand
                mix_acc <= mix_acc + osc_sample_shifted;
            end

            if (sweep_done) begin
                if (mix_shifted > 33'sd32767) begin
                    sample <= 16'h7FFF;
                end else if (mix_shifted < -33'sd32768) begin
                    sample <= 16'h8000;
                end else begin
                    sample <= mix_shifted[15:0];
                end
            end
        end
    end

    always_ff @(posedge clk) begin
        if(reset) begin
            sample_valid <= 1'b0;
        end else begin
            sample_valid <= sweep_done;
        end
    end


endmodule
```

Software (C)

fpga_bridge.h

```
#ifndef SYNTH_TO_FPGA_H
#define SYNTH_TO_FPGA_H
```

```
3
4  #include <stdint.h>
5  #include <stdbool.h>
6
7  #define WAVE_SYNTH_PERIPHERAL_BYTES 0x80000
8
9  #define OSC_STEP(v)       (0x20000u + (v) * 8u + 0u)
10 #define OSC_CTRL(v)       (0x20000u + (v) * 8u + 1u)
11 #define OSC_TABLE(v)      (0x20000u + (v) * 8u + 2u)
12 #define OSC_AMP(v)        (0x20000u + (v) * 8u + 3u)
13 #define OSC_RESOLUTION(v) (0x20000u + (v) * 8u + 4u)
14
15 #define AMP_CTRL 0x20180u
16
17 #define HEX_REG(i)   (0x20100u + (i))
18
19 /* DE1-SoC HEX segments are active-low. bit order: gfedcba */
20 #define SEG_BLANK 0x7Fu
21 #define SEG_0 0x40u
22 #define SEG_1 0x79u
23 #define SEG_2 0x24u
24 #define SEG_3 0x30u
25 #define SEG_4 0x19u
26 #define SEG_5 0x12u
27 #define SEG_6 0x02u
28 #define SEG_7 0x78u
29 #define SEG_8 0x00u
30 #define SEG_9 0x10u
31 #define SEG_A 0x08u
32 #define SEG_B 0x03u
33 #define SEG_C 0x46u
34 #define SEG_D 0x21u
35 #define SEG_E 0x06u
36 #define SEG_F 0x0Eu
37 #define SEG_G 0x10u
38 #define SEG_S 0x12u
39 #define SEG_I 0x79u
40 #define SEG_N 0x2Bu
41 #define SEG_R 0x2Fu
42 #define SEG_T 0x07u
43 #define SEG_U 0x41u
44 #define SEG_Q 0x18u
45
46 #define TABLE_SIZE 32768
```

```c
#define NUM_TABLE_SLOTS 4

#define WAVETABLE_WORD(slot,sample) (((slot)* TABLE_SIZE) + (sample))

#define CTRL_IDLE   0x0000u
#define CTRL_STOP   0x0001u
#define CTRL_START  0x0002u
#define CTRL_RESET  0x0003u

#define SAMPLE_RATE 48000
#define NUM_OSCILLATORS 32

/* adsr macros */
#define ENV_TICK_NS             1000000L
#define ENV_PEAK                255
#define ENV_SUSTAIN_LEVEL       192
#define ENV_ATTACK_PER_TICK     64
#define ENV_DECAY_PER_TICK      4
#define ENV_RELEASE_PER_TICK    2

typedef struct {
    volatile uint16_t *regs;
    int fd;
} peripheral;

int fpga_init(peripheral *lw_bus);
void fpga_kill_voices(peripheral *lw_bus);
void fpga_cleanup(peripheral *lw_bus);

/* per voice setters */
void fpga_set_step(peripheral *lw_bus, int voice, uint16_t step_size);
void fpga_set_ctrl(peripheral *lw_bus, int voice, uint16_t ctrl);
void fpga_set_table(peripheral *lw_bus, int voice, uint16_t slot);
void fpga_set_amp(peripheral *lw_bus, int voice, uint16_t amp);
void fpga_set_hex(peripheral *lw_bus, int idx, uint8_t pattern);
void fpga_set_osc_resolution(peripheral *lw_bus, int voice, uint8_t resolution);
void fpga_voice_start(peripheral *lw_bus, int voice, uint16_t step_size,
                      uint16_t slot);
void fpga_kill_voice(peripheral *lw_bus, int voice);
int fpga_load_slot(peripheral *lw_bus, int slot, const int16_t *samples,
                   int n);

/* master amp*/
void fpga_set_master_amp(peripheral *lw_bus, uint16_t amp);
```

```
91
92 #endif
```

fpga_bridge.c

```
1  #include "fpga_bridge.h"
2  #include <fcntl.h>
3  #include <stdint.h>
4  #include <stdio.h>
5  #include <sys/mman.h>
6  #include <time.h>
7  #include <unistd.h>
8
9  int fpga_init(peripheral *lw_bus) {
10   lw_bus->fd = open("/dev/wave_synth", O_RDWR | O_SYNC);
11
12   if (lw_bus->fd < 0) {
13     perror("open /dev/wave_synth");
14     return -1;
15   }
16
17   lw_bus->regs = mmap(NULL, WAVE_SYNTH_PERIPHERAL_BYTES, PROT_READ | PROT_WRITE,
18                       MAP_SHARED, lw_bus->fd, 0);
19
20   if (lw_bus->regs == MAP_FAILED) {
21     perror("mmap /dev/wave_synth");
22     close(lw_bus->fd);
23     lw_bus->fd = -1;
24     lw_bus->regs = NULL;
25     return -1;
26   }
27
28   fpga_kill_voices(lw_bus);
29   lw_bus->regs[AMP_CTRL] = 127;
30   return 0;
31 }
32
33 void fpga_kill_voices(peripheral *lw_bus) {
34   for (int i = 0; i < NUM_OSCILLATORS; i++) {
35     lw_bus->regs[OSC_CTRL(i)] = CTRL_STOP;
36   }
37 }
38
39 void fpga_cleanup(peripheral *lw_bus) {
```

```
  if (lw_bus->regs) {
    fpga_kill_voices(lw_bus);
    munmap((void *)lw_bus->regs, WAVE_SYNTH_PERIPHERAL_BYTES);
    lw_bus->regs = NULL;
  }
  if (lw_bus->fd >= 0) {
    close(lw_bus->fd);
    lw_bus->fd = -1;
  }
}

void fpga_set_step(peripheral *lw_bus, int voice, uint16_t step_size) {
  if (voice < 0 || voice >= NUM_OSCILLATORS) {
    return;
  }
  lw_bus->regs[OSC_STEP(voice)] = step_size;
}

void fpga_set_ctrl(peripheral *lw_bus, int voice, uint16_t ctrl) {
  if (voice < 0 || voice >= NUM_OSCILLATORS) {
    return;
  }
  lw_bus->regs[OSC_CTRL(voice)] = ctrl;
}

void fpga_set_table(peripheral *lw_bus, int voice, uint16_t slot) {
  if (voice < 0 || voice >= NUM_OSCILLATORS) {
    return;
  }
  if (slot >= NUM_TABLE_SLOTS) {
    return;
  }
  lw_bus->regs[OSC_TABLE(voice)] = slot;
}

void fpga_set_amp(peripheral *lw_bus, int voice, uint16_t amp) {
  lw_bus->regs[OSC_AMP(voice)] = amp;
}

void fpga_set_master_amp(peripheral *lw_bus, uint16_t amp) {
  lw_bus->regs[AMP_CTRL] = amp;
}

void fpga_set_hex(peripheral *lw_bus, int idx, uint8_t pattern) {
```

```
  if (idx < 0 || idx >= 6) {
    return;
  }
  lw_bus->regs[HEX_REG(idx)] = pattern & 0x7Fu;
}

void fpga_set_osc_resolution(peripheral *lw_bus, int voice, uint8_t resolution) {
  if (voice < 0 || voice >= NUM_OSCILLATORS) {
    return;
  }
  lw_bus->regs[OSC_RESOLUTION(voice)] = (uint16_t)(resolution & 0x0Fu);
}

void fpga_voice_start(peripheral *lw_bus, int voice, uint16_t step_size,
                      uint16_t slot) {
  if (voice < 0 || voice >= NUM_OSCILLATORS) {
    return;
  }
  if (slot >= NUM_TABLE_SLOTS) {
    return;
  }
  lw_bus->regs[OSC_STEP(voice)] = step_size;
  lw_bus->regs[OSC_TABLE(voice)] = slot;
  lw_bus->regs[OSC_CTRL(voice)] = CTRL_RESET;

  struct timespec delay = {0, 200 * 1000L};
  nanosleep(&delay, NULL);

  lw_bus->regs[OSC_CTRL(voice)] = CTRL_START;
}

void fpga_kill_voice(peripheral *lw_bus, int voice) {
  if (voice < 0 || voice >= NUM_OSCILLATORS) {
    return;
  }
  lw_bus->regs[OSC_CTRL(voice)] = CTRL_STOP;
}

int fpga_load_slot(peripheral *lw_bus, int slot, const int16_t *samples,
                   int n) {
  if (slot < 0 || slot >= NUM_TABLE_SLOTS) {
    return -1;
  }
  if (!samples || n <= 0) {
```

```
    return -1;
  }
  if(n > TABLE_SIZE){
    n = TABLE_SIZE;
  }

  for(int i = 0; i < n; i++){
    lw_bus->regs[WAVETABLE_WORD(slot, i)] = (uint16_t)samples[i];
  }


  for (int i = n; i < TABLE_SIZE; i++){
    lw_bus->regs[WAVETABLE_WORD(slot, i)] = 0;
  }

  return 0;
}
```

midi.h

```
#ifndef _MIDI_H
#define _MIDI_H

#include <stdint.h>

typedef struct {
    uint8_t status;
    uint8_t note;
    uint8_t attack;
} midi_event_t;


#define MIDI_NOTE_ON 0x90
#define MIDI_NOTE_OFF 0x80
#define MIDI_CONTROL_CHANGE 0xB0
#define MIDI_PROGRAM_CHANGE 0xC0
#define MIDI_PITCH_BEND 0xE0

#define MIDI_STATUS_MASK 0xF0
#define MIDI_CHANNEL_MASK 0x0F


int midi_open(void);
int midi_read(int fd, midi_event_t *evt);
```

```
25 void midi_close(int fd);
26
27 #endif
```

midi.c

```
1  /*
2   *
3   * CSEE 4840 Final Project
4   *
5   * Name/UNI: Henry Minsky (hm3121), Opalina Khanna (ok2373), Sunny Fang (yf2610)
6   */
7  #include "midi.h"
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <stdint.h>
11 #include <unistd.h>
12 #include <fcntl.h>
13
14 int midi_open(void){
15     char path[256];
16     for(int card = 0; card < 8; card++){
17         snprintf(path, sizeof(path), "/dev/snd/midiC%dD0",card);
18         int fd = open(path, O_RDONLY);
19         if(fd >= 0){
20             return fd;
21         }
22     }
23     fprintf(stderr, "midi open: no midi sound device found in /dev/snd\n");
24     return -1;
25 }
26
27 int midi_read(int fd, midi_event_t *evt){
28     static uint8_t running_status;
29
30     uint8_t byte;
31
32     while(1){
33
34         if(read(fd,&byte,1) != 1){
35             return -1;
36         }
37
38         if(byte >= 0xF8){
```

```
            continue;
        }
        if(byte >= 0xF0){
            running_status = 0;
            continue;
        }

        if (byte & 0x80){
            evt->status = byte;
            running_status = byte;
            if(read(fd,&evt->note,1)!= 1){
                return -1;
            }
        }
        else{
            if(!(running_status & 0x80)){
                continue;
            }
            evt->status = running_status;
            evt->note = byte;
        }

        uint8_t kind = evt->status & 0xF0;
        if(kind == 0xC0 || kind == 0xD0){
            evt->attack = 0;
        }
        else{
            if(read(fd, &evt->attack,1) != 1){
                return -1;
            }
        }
        return 0;
    }

}

void midi_close(int fd){
    if(fd >= 0){
        close(fd);
    }
}
```

midi_to_fpga.c

```c
#include "fpga_bridge.h"
#include "midi.h"
#include "synth_functions.h"
#include "wavetable.h"
#include <fcntl.h>
#include <math.h>
#include <pthread.h>
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <time.h>
#include <unistd.h>

// our global array to track our oscillator states!
static struct oscillator oscillators[NUM_OSCILLATORS] = {0};

static int global_wavetable = 0;

pthread_mutex_t osc_lock;

static uint8_t last_note = 0;
static bool have_last_note = false;

/* note class 0..11 = C, C#, D, D#, E, F, F#, G, G#, A, A#, B */
static const uint8_t NOTE_LETTER[12] = {
    SEG_C, SEG_C, SEG_D, SEG_D, SEG_E, SEG_F,
    SEG_F, SEG_G, SEG_G, SEG_A, SEG_A, SEG_B,
};
static const uint8_t NOTE_IS_SHARP[12] = {0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0};
static const uint8_t SEG_DIGIT[10] = {SEG_0, SEG_1, SEG_2, SEG_3, SEG_4,
                                      SEG_5, SEG_6, SEG_7, SEG_8, SEG_9};

/* HEX2, HEX1, HEX0 patterns per wavetable slot */
static const uint8_t TABLE_NAME[NUM_TABLE_SLOTS][3] = {
    {SEG_S, SEG_I, SEG_N}, /* slot 0: sin (sine) */
    {SEG_S, SEG_A, SEG_U}, /* slot 1: sau (sawtooth) */
    {SEG_S, SEG_Q, SEG_R}, /* slot 2: sqr (square) */
    {SEG_T, SEG_R, SEG_I}, /* slot 3: tri (triangle) */
};

static void update_display(peripheral *lw_bus) {
```

```
44    /* HEX2..HEX0: current wavetable abbreviation (always shown) */
45    int slot = global_wavetable;
46    if (slot < 0 || slot >= NUM_TABLE_SLOTS)
47      slot = 0;
48    fpga_set_hex(lw_bus, 2, TABLE_NAME[slot][0]);
49    fpga_set_hex(lw_bus, 1, TABLE_NAME[slot][1]);
50    fpga_set_hex(lw_bus, 0, TABLE_NAME[slot][2]);
51
52    /* HEX5..HEX3: most recently pressed note (blank until first note) */
53    if (!have_last_note) {
54      fpga_set_hex(lw_bus, 5, SEG_BLANK);
55      fpga_set_hex(lw_bus, 4, SEG_BLANK);
56      fpga_set_hex(lw_bus, 3, SEG_BLANK);
57      return;
58    }
59
60    uint8_t cls = last_note % 12;
61    int octave = (int)(last_note / 12) - 1; /* MIDI: note 0 = C-1 */
62    if (octave < 0)
63      octave = 0;
64    if (octave > 9)
65      octave = 9;
66
67    fpga_set_hex(lw_bus, 5, NOTE_LETTER[cls]);
68    fpga_set_hex(lw_bus, 4, NOTE_IS_SHARP[cls] ? SEG_S : SEG_BLANK);
69    fpga_set_hex(lw_bus, 3, SEG_DIGIT[octave]);
70  }
71
72  double AMP_STEP = 1.0 / 128;
73
74  void *run_midi_reciever(void *arg) {
75    peripheral *lw_bus = (peripheral *)arg;
76
77    int midi_fd = midi_open();
78    if (midi_fd < 0) {
79      return NULL;
80    }
81
82    midi_event_t midi_packet;
83
84    while (1) {
85      if (midi_read(midi_fd, &midi_packet) < 0) {
86        continue;
87      }
```

```
88
89      if ((midi_packet.status & MIDI_STATUS_MASK) == MIDI_NOTE_ON) {
90        uint16_t step = 0;
91        pthread_mutex_lock(&osc_lock);
92        int i = osc_find_free_slot(oscillators);
93        if (i >= 0) {
94          step = voice_step(midi_packet.note, (uint8_t)global_wavetable);
95          fpga_set_osc_resolution(lw_bus, i, wavetable_slot_resolution[global_wavetable]);
96          oscillators[i].note = midi_packet.note;
97          oscillators[i].step_size = step;
98          oscillators[i].wavetable_slot = global_wavetable;
99          oscillators[i].in_use = true;
100         oscillators[i].phase = ENV_ATTACK;
101         double vel_factor = AMP_STEP * midi_packet.attack;
102         oscillators[i].sustain      = (uint16_t)(vel_factor * ENV_SUSTAIN_LEVEL);
103         oscillators[i].attack_step  = (uint16_t)(vel_factor * ENV_ATTACK_PER_TICK);
104         oscillators[i].decay_step   = (uint16_t)(vel_factor * ENV_DECAY_PER_TICK);
105         oscillators[i].release_step = (uint16_t)(vel_factor * ENV_RELEASE_PER_TICK);
106         oscillators[i].peak = (uint16_t)(vel_factor * ENV_PEAK);
107         if (oscillators[i].attack_step  == 0) oscillators[i].attack_step  = 1;
108         if (oscillators[i].decay_step   == 0) oscillators[i].decay_step   = 1;
109         if (oscillators[i].release_step == 0) oscillators[i].release_step = 1;
110       }
111       pthread_mutex_unlock(&osc_lock);
112
113       if (i >= 0) {
114         fpga_voice_start(lw_bus, i, step, global_wavetable);
115         last_note = midi_packet.note;
116         have_last_note = true;
117         update_display(lw_bus);
118       }
119
120     } else if ((midi_packet.status & MIDI_STATUS_MASK) == MIDI_NOTE_OFF) {
121       pthread_mutex_lock(&osc_lock);
122       int i = osc_find_note_slot(oscillators, midi_packet.note);
123
124       pthread_mutex_unlock(&osc_lock);
125
126       if (i >= 0) {
127         update_display(lw_bus);
128       }
129     } else if ((midi_packet.status & MIDI_STATUS_MASK) == MIDI_PROGRAM_CHANGE) {
130       global_wavetable = midi_packet.note % NUM_TABLE_SLOTS;
131       for (int j = 0; j < NUM_OSCILLATORS; j++) {
```

```
132         if (oscillators[j].in_use) {
133           pthread_mutex_lock(&osc_lock);
134           oscillators[j].wavetable_slot = global_wavetable;
135           pthread_mutex_unlock(&osc_lock);
136           fpga_set_table(lw_bus, j, global_wavetable);
137         }
138       }
139       update_display(lw_bus);
140     } else if (midi_packet.status == 0xB1 && midi_packet.note == 0x07) {
141       uint16_t amp = (uint16_t)(midi_packet.attack & 0x7F) << 1;
142       fpga_set_master_amp(lw_bus, amp);
143     }
144   }
145   return NULL;
146 }
147
148 void *run_adsr_envelope(void *arg) {
149   peripheral *lw_bus = (peripheral *)arg;
150   struct timespec tick = {0, ENV_TICK_NS};
151   static uint16_t last_written[NUM_OSCILLATORS] = {0};
152
153   while (1) {
154     clock_nanosleep(CLOCK_MONOTONIC, 0, &tick, NULL);
155     /* iterate over each oscillator */
156
157     for (int i = 0; i < NUM_OSCILLATORS; i++) {
158       struct oscillator *curr = &oscillators[i];
159       uint16_t curr_amp;
160       bool kill_voice = false;
161       pthread_mutex_lock(&osc_lock);
162       switch (curr->phase) {
163       case (ENV_IDLE):
164         curr->env_amp_q8 = 0;
165         break;
166       case (ENV_ATTACK):
167         if (curr->env_amp_q8 + curr->attack_step >= curr->peak) {
168           curr->phase = ENV_DECAY;
169           curr->env_amp_q8 = curr->peak;
170         } else {
171           curr->env_amp_q8 += curr->attack_step;
172           /* no phase change */
173         }
174         break;
175       case (ENV_DECAY):
```

```
176         if (curr->env_amp_q8 - curr->decay_step <= curr->sustain) {
177           curr->phase = ENV_SUSTAIN;
178           curr->env_amp_q8 = curr->sustain;
179         } else {
180           /* no phase change */
181           curr->env_amp_q8 -= curr->decay_step;
182         }
183         break;
184       case (ENV_SUSTAIN):
185         curr->env_amp_q8 = curr->sustain;
186         break;
187       case (ENV_RELEASE):
188         if (curr->env_amp_q8 <= curr->release_step) {
189           curr->phase = ENV_IDLE;
190           curr->env_amp_q8 = 0;
191           curr->in_use = false;
192           curr->note = 0;
193           curr->step_size = 0;
194           curr->wavetable_slot = 0;
195           kill_voice = true;
196         } else {
197           /* no phase change */
198           curr->env_amp_q8 -= curr->release_step;
199         }
200       }
201       curr_amp = curr->env_amp_q8;
202       pthread_mutex_unlock(&osc_lock);
203       if (kill_voice)
204         fpga_kill_voice(lw_bus, i);
205       if (curr_amp != last_written[i]) {
206         fpga_set_amp(lw_bus, i, curr_amp);
207         last_written[i] = curr_amp;
208       }
209     }
210   }
211   return NULL;
212 }
213
214 int main(int argc, char **argv) {
215
216   if (argc < 2) {
217     fprintf(stderr, "need to pass a *.bin file for loading wavetable in");
218   }
219
```

```
  peripheral lw_bus;
  if (fpga_init(&lw_bus) != 0) {
    return -1;
  }

  int loaded = load_wavetable_bin(argv[1], &lw_bus);

  if (loaded <= 0) {
    fprintf(stderr, "No wavetables loaded, check validity of %s \n", argv[1]);
  }

  update_display(&lw_bus);

  pthread_t midi_thread;
  pthread_t adsr_thread;
  pthread_mutex_init(&osc_lock, NULL);

  pthread_create(&midi_thread, NULL, run_midi_reciever, &lw_bus);
  pthread_create(&adsr_thread, NULL, run_adsr_envelope, &lw_bus);

  pthread_join(midi_thread, NULL);
  return 0;
}
```

synth_functions.h

```
#ifndef _SYNTH_FUNCTIONS_H
#define _SYNTH_FUNCTIONS_H

#include <stdint.h>
#include <stdbool.h>

/*
* core struct, respresents a single oscillator and its corresponding registers in the fpga
*/
typedef enum { ENV_IDLE, ENV_ATTACK, ENV_DECAY, ENV_SUSTAIN, ENV_RELEASE } env_phase_t;
struct oscillator {
    uint16_t step_size;
    uint16_t control;
    uint16_t velocity;
    uint16_t wavetable_slot;
    uint16_t resolution; /* unused right now*/
    uint8_t note;
    bool in_use;
```

```
    env_phase_t phase;
    uint16_t    env_amp_q8;
    uint16_t    sustain;
    uint16_t    attack_step;
    uint16_t    decay_step;
    uint16_t    release_step;
    uint16_t    peak;
};

uint16_t voice_step(uint8_t note, uint8_t slot);

int osc_find_free_slot( struct oscillator *oscillators);
int osc_find_note_slot( struct oscillator *oscillators, uint8_t note);

#endif
```

synth_functions.c

```
#include <stdint.h>
#include <math.h>
#include "fpga_bridge.h"
#include "synth_functions.h"
#include "wavetable.h"



int osc_find_note_slot( struct oscillator *oscillators, uint8_t note){
    int found = 0;
    for(int i = 0; i < NUM_OSCILLATORS; i++){
        if(oscillators[i].in_use && oscillators[i].note == note){
           oscillators[i].phase = ENV_RELEASE;
           found++;
        }
    }
    if(found >= 0){
        return found;
    }
    else {
        return -1;
    }
}


```

```c
int osc_find_free_slot( struct oscillator *oscillators){
    for(int i = 0; i < NUM_OSCILLATORS; i++){
        if(!oscillators[i].in_use){
            return i;
        }
    }
    return -1;
}

uint16_t voice_step(uint8_t note, uint8_t slot){
    if(slot >= NUM_TABLE_SLOTS) slot = 0;
    double mult = pow(2.0, ((double)note - 60.0) / 12.0);
    double step = (double)wavetable_base_step[slot] * mult;
    uint32_t rounded = (uint32_t)(step + 0.5);
    return (uint16_t)(rounded > 0xFFFFu ? 0xFFFFu : rounded);
}
```

wavetable.h

```c
#ifndef _WAVETABLE_H
#define _WAVETABLE_H

#include <stdint.h>
#include <stddef.h>
#include "fpga_bridge.h"


#define WAVETABLE_DEFAULT_BASE_STEP 357u
#define WAVETABLE_DEFAULT_SLOT_RESOLUTION 8u

/* per-slot pitch anchor and HW phase-increment shift, populated from
   the .meta sidecar at startup. Defaults preserve legacy single-cycle
   behavior if the manifest is missing. */
extern uint16_t wavetable_base_step[NUM_TABLE_SLOTS];
extern uint8_t  wavetable_slot_resolution[NUM_TABLE_SLOTS];

/* Loads wavetable bin generated from tools/build_wavetable.py.
   File layout: 24B header ("WTSY" magic, u16 version, u16 num_meta_slots,
   then per-slot u16 base_step + u8 resolution + u8 reserved), followed by
   one TABLE_SIZE-sample audio block per populated slot. Also populates
   wavetable_base_step[] and wavetable_slot_resolution[] from the header. */
int load_wavetable_bin(const char *path, peripheral *lw_bus);

#endif
```

wavetable.c

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include "fpga_bridge.h"
#include "wavetable.h"

uint16_t wavetable_base_step[NUM_TABLE_SLOTS] = {
    WAVETABLE_DEFAULT_BASE_STEP,
    WAVETABLE_DEFAULT_BASE_STEP,
    WAVETABLE_DEFAULT_BASE_STEP,
    WAVETABLE_DEFAULT_BASE_STEP,
};

uint8_t wavetable_slot_resolution[NUM_TABLE_SLOTS] = {
    WAVETABLE_DEFAULT_SLOT_RESOLUTION,
    WAVETABLE_DEFAULT_SLOT_RESOLUTION,
    WAVETABLE_DEFAULT_SLOT_RESOLUTION,
    WAVETABLE_DEFAULT_SLOT_RESOLUTION,
};

int load_wavetable_bin(const char *path, peripheral *lw_bus){
    FILE *f = fopen(path,"rb");

    if(!f){
        perror("load_wavetable_bin: fopen");
        return -1;
    }

    /* Header: 4B magic "WTSY", u16 version, u16 num_meta_slots,
       then per-slot {u16 base_step, u8 resolution, u8 reserved}. */
    char magic[4];
    if (fread(magic, 1, 4, f) != 4 || memcmp(magic, "WTSY", 4) != 0) {
        fprintf(stderr, "wavetable: bad magic in %s\n", path);
        fclose(f);
        return -1;
    }
    uint16_t version, num_meta;
    if (fread(&version, 2, 1, f) != 1 || fread(&num_meta, 2, 1, f) != 1) {
        fprintf(stderr, "wavetable: short read on header\n");
        fclose(f);
        return -1;
    }
    if (version != 1) {
```

```
        fprintf(stderr, "wavetable: unsupported version %u\n", version);
        fclose(f);
        return -1;
    }
    for (uint16_t i = 0; i < num_meta; i++) {
        uint16_t bs; uint8_t res, reserved;
        if (fread(&bs, 2, 1, f) != 1
            || fread(&res, 1, 1, f) != 1
            || fread(&reserved, 1, 1, f) != 1) {
            fprintf(stderr, "wavetable: short read on slot %u metadata\n", i);
            fclose(f);
            return -1;
        }
        if (i < NUM_TABLE_SLOTS) {
            wavetable_base_step[i] = bs;
            wavetable_slot_resolution[i] = res;
        }
    }

    static int16_t slot_buf[TABLE_SIZE];
    int slot = 0;

    while(slot < NUM_TABLE_SLOTS){
        size_t tableslot = fread(slot_buf, sizeof(int16_t), TABLE_SIZE, f);

        if(tableslot == 0){
            break;
        }

        if (tableslot != TABLE_SIZE){
            fprintf(stderr, "wavetable: malformed audio block at slot %d (got %zu samples)\n",
                    slot, tableslot);
            fclose(f);
            return -1;
        }
        fpga_load_slot(lw_bus, slot, slot_buf, TABLE_SIZE);
        printf("slot %d: loaded (base_step=%u, resolution=%u)\n",
               slot, wavetable_base_step[slot], wavetable_slot_resolution[slot]);
        slot++;
    }
    fclose(f);

    return slot;
}
```

Kernel Driver

sw/driver/wave_synth.h

```h
#ifndef _WAVE_SYNTH_H
#define _WAVE_SYNTH_H

#define WAVE_SYNTH_NAME "wave_synth"

#endif
```

sw/driver/wave_synth.c

```c
#include "wave_synth.h"
#include <linux/io.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <linux/module.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/platform_device.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/version.h>



struct wave_synth_dev {
    struct resource res;
} dev;


// see https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html
static int wave_synth_mmap(struct file *filp, struct vm_area_struct *vma){
    unsigned long len = vma->vm_end - vma->vm_start;
    unsigned long dev_size = resource_size(&dev.res);

    if(vma->vm_pgoff != 0){
        return -EINVAL;
    }

    if(len > dev_size){
        return -EINVAL;
    }
```

```
33
34      // necessary so that read/writes to this portion of memory are never using
35      // cache
36      vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
37
38      if(io_remap_pfn_range(vma, vma->vm_start, dev.res.start >> PAGE_SHIFT,len,
        ↪  vma->vm_page_prot)){
39          return -EAGAIN;
40      }
41      return 0;
42
43  }
44
45  static const struct file_operations wave_synth_fops = {
46      .owner = THIS_MODULE,
47      .mmap = wave_synth_mmap,
48  };
49
50  static struct miscdevice wave_synth_misc_device = {
51      .minor = MISC_DYNAMIC_MINOR,
52      .name = WAVE_SYNTH_NAME,
53      .fops = &wave_synth_fops,
54  };
55
56  static int __init wave_synth_probe(struct platform_device *pdev){
57      int ret;
58
59      ret = misc_register(&wave_synth_misc_device);
60      if(ret){
61          return ret;
62      }
63
64      ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
65      if(ret){
66          ret = -ENOENT;
67          goto out_deregister;
68      }
69
70      if (request_mem_region(dev.res.start,resource_size(&dev.res),WAVE_SYNTH_NAME) == NULL){
71          ret = -EBUSY;
72          goto out_deregister;
73      }
74
75      return 0;
```

```
76
77  out_deregister:
78      misc_deregister(&wave_synth_misc_device);
79      return ret;
80  }
81
82
83  static int __exit wave_synth_remove(struct platform_device *pdev){
84      release_mem_region(dev.res.start,resource_size(&dev.res));
85      misc_deregister(&wave_synth_misc_device);
86      return 0;
87  }
88
89  static const struct of_device_id wave_synth_of_match[] = {
90      { .compatible = "csee4840,wave_table_synth-1.0" },
91      {},
92  };
93  MODULE_DEVICE_TABLE(of, wave_synth_of_match);
94
95  static struct platform_driver wave_synth_driver = {
96      .driver = {
97          .name = WAVE_SYNTH_NAME,
98          .owner = THIS_MODULE,
99          .of_match_table = of_match_ptr(wave_synth_of_match),
100     },
101     .remove = __exit_p(wave_synth_remove),
102 };
103
104 static int __init wave_synth_init(void){
105     pr_info(WAVE_SYNTH_NAME ": init\n");
106     return platform_driver_probe(&wave_synth_driver, wave_synth_probe);
107 }
108
109 static void __exit wave_synth_exit(void){
110     platform_driver_unregister(&wave_synth_driver);
111     pr_info(WAVE_SYNTH_NAME ": exit \n");
112 }
113
114 module_init(wave_synth_init);
115 module_exit(wave_synth_exit);
116
117 MODULE_LICENSE("GPL");
118 MODULE_AUTHOR("Harry Minsky, CSEE4840");
119 MODULE_DESCRIPTION("Mmap connector driver for comms over lw avalon bus to de1soc");
```

sw/driver/Makefile

```
ifneq (${KERNELRELEASE},)
  obj-m := wave_synth.o
else

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

default: module
module:
  ${MAKE} -C ${KERNEL_SOURCE} M=${PWD} modules
clean:
  ${MAKE} -C ${KERNEL_SOURCE} M=${PWD} clean

.PHONY: default module clean

endif
```

Tooling (Python)

sw/tools/build_wavetable.py

```
import argparse
import struct
import sys
import wave
from pathlib import Path


def load_wav_mono_i16(path):
    """Return (mono int16 samples, source sample rate) from a PCM .wav file"""
    with wave.open(str(path), "rb") as w:
        channels = w.getnchannels()
        samp_width = w.getsampwidth()
        rate = w.getframerate()
        n_frames = w.getnframes()
        raw = w.readframes(n_frames)

    if channels not in (1, 2):
        raise ValueError("unsupported channel count ")
    if samp_width not in (1, 2):
        raise ValueError("Unsupposrted sample width (must be 8 or 16 bit)")

    if samp_width == 2:
```

```
        count = len(raw) // 2
        frames = list(struct.unpack("<{}h".format(count), raw))
    else:
        frames = [(b - 128) << 8 for b in raw]

    if channels == 2:
        frames = [(frames[i] + frames[i + 1]) // 2 for i in range(0, len(frames), 2)]

    return frames, rate


def resample_linear(samples, target_n):
    """Resample to exactly target_n samples via linear interpolation."""
    n_in = len(samples)
    if n_in == target_n:
        return list(samples)
    out = [0] * target_n
    scale = (n_in - 1) / (target_n - 1) if target_n > 1 else 0
    for i in range(target_n):
        src = i * scale
        lo = int(src)
        hi = min(lo + 1, n_in - 1)
        frac = src - lo
        v = samples[lo] * (1.0 - frac) + samples[hi] * frac
        if v > 32767:
            v = 32767
        elif v < -32768:
            v = -32768
        out[i] = int(v)
    return out


TABLE_SIZE = 32768
MAX_SLOTS = 4
TARGET_RATE = 48000
DEFAULT_BASE_STEP = 357
DEFAULT_RESOLUTION = 8


ap = argparse.ArgumentParser(
    description="takes in newline seperated text file of wavs and converts into a binary"
)
ap.add_argument("manifest")
ap.add_argument("-o", "--output", default="wavetable.bin")
```

```
67
68  args = ap.parse_args()
69
70  # Header format (24 bytes for NUM_TABLE_SLOTS=4):
71  #   4B  magic "WTSY"
72  #   2B  version (=1)
73  #   2B  num_metadata_slots
74  #   per metadata slot (4B each):
75  #       2B  base_step
76  #       1B  resolution
77  #       1B  reserved (=0)
78  HEADER_MAGIC = b"WTSY"
79  HEADER_VERSION = 1
80
81
82  manifest_path = args.manifest
83
84  base = Path(manifest_path).resolve().parent
85
86  entries = []  # list of (path, base_step, resolution, path_str)
87
88  with open(manifest_path, "r") as f:
89      for line_no, raw_line in enumerate(f, start=1):
90          line = raw_line.strip()
91          if not line or line.startswith("#"):
92              continue
93          parts = [p.strip() for p in line.split(",")]
94          path_str = parts[0]
95          if len(parts) >= 2 and parts[1] != "":
96              try:
97                  bs = int(parts[1])
98              except ValueError:
99                  print(
100                     "samples.txt line {}: invalid base_step {!r}".format(
101                         line_no, parts[1]
102                     )
103                 )
104                 sys.exit(1)
105         else:
106             bs = DEFAULT_BASE_STEP
107         if len(parts) >= 3 and parts[2] != "":
108             try:
109                 res = int(parts[2])
110             except ValueError:
```

```
                print(
                    "samples.txt line {}: invalid resolution {!r}".format(
                        line_no, parts[2]
                    )
                )
                sys.exit(1)
        else:
            res = DEFAULT_RESOLUTION
        p = Path(path_str)
        if not p.is_absolute():
            p = base / p
        entries.append((p, bs, res, path_str))

if not entries:
    print("No wav files found you dope")
    sys.exit(1)

if len(entries) > MAX_SLOTS:
    print("Too many wavs put in you dope")
    sys.exit(1)


with open(args.output, "wb") as out:
    # Header (24 bytes for MAX_SLOTS=4)
    out.write(HEADER_MAGIC)
    out.write(struct.pack("<HH", HEADER_VERSION, MAX_SLOTS))
    for i in range(MAX_SLOTS):
        if i < len(entries):
            _, bs, res, _ = entries[i]
        else:
            bs = DEFAULT_BASE_STEP
            res = DEFAULT_RESOLUTION
        out.write(struct.pack("<HBB", bs, res, 0))

    # Audio: one TABLE_SIZE int16 block per slot listed in samples.txt
    for slot, (p, base_step, resolution, path_str) in enumerate(entries):
        raw, rate = load_wav_mono_i16(p)
        n_in = len(raw)
        if n_in != TABLE_SIZE:
            ratio = n_in / TABLE_SIZE
            effective_rate = int(round(rate * TABLE_SIZE / n_in))
            direction = "down" if n_in > TABLE_SIZE else "up"
            print(
                "NOTE: slot {} ({}): resampled {} {} -> {} samples "
```

```
                "(ratio {:.3f}, effective rate {} Hz)".format(
                    slot, path_str, n_in, direction, TABLE_SIZE, ratio, effective_rate
                )
            )
            stored = resample_linear(raw, TABLE_SIZE)
        else:
            stored = list(raw)
        out.write(struct.pack("<{}h".format(TABLE_SIZE), *stored))
        print(
            "slot {}: {} ({} Hz, {} samples in, base_step={}, resolution={})".format(
                slot, p, rate, n_in, base_step, resolution
            )
        )

print("Wrote {} slot(s) to {} (header + {} audio block(s))".format(
    len(entries), args.output, len(entries)
))
```

Build System

kernel/Makefile

```
# linux kernel make file for altera de1 soc
# to be run on ubuntu x86 machine with kernel
# version 6.17
# shamelessly ripped and modified from CSEE4840 lab3
# https://www.cs.columbia.edu/~sedwards/classes/2026/4840-spring/lab3-hw.tar.gz

KERNEL_REPO = https://github.com/altera-opensource/linux-socfpga.git
KERNEL_BRANCH = rel_socfpga-4.19_19.04.01_pr
DEFAULT_CONFIG = socfpga_defconfig
CROSS = env CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm

KERNEL_DIR = linux-socfpga
KERNEL_CONFIG = $(KERNEL_DIR)/.config
ZIMAGE = $(KERNEL_DIR)/arch/arm/boot/zImage

# kernel-download
#
#   Clone the Linux kernel repository
#
# kernel-config
#
#   Set up the default kernel configuration
```

```
#
# kernel-menuconfig
#
#   (Optional) Access the kernel configuration menu to make further
#   adjustments about which modules are included
#
# zimage
#
#   Compile the kernel

.PHONY : kernel-download kernel-config kernel-menuconfig zimage
kernel-download : $(KERNEL_DIR)
kernel-config : $(KERNEL_CONFIG)
kernel-menuconfig :
  $(CROSS) $(MAKE) -C $(KERNEL_DIR) menuconfig
zimage : $(ZIMAGE)

$(KERNEL_DIR) :
  git clone --branch $(KERNEL_BRANCH) $(KERNEL_REPO) $(KERNEL_DIR)

# Configure the kernel.  Start from a provided default,
#
# Turn off version checking (makes it easier to compile kernel
# modules and not have them complain about version)
#
# Turn on large file (+2TB) support, which the ext4 filesystem
# requires by default (it will not be able to mount the root
# filesystem read/write otherwise)
# Order-only prereq so an existing .config is left alone | only regenerated
# when the file is missing or when `make kernel-config` is invoked directly.
$(KERNEL_CONFIG) : | $(KERNEL_DIR)
  $(CROSS) $(MAKE) -C $(KERNEL_DIR) $(DEFAULT_CONFIG)
  $(KERNEL_DIR)/scripts/config --file $(KERNEL_CONFIG) \
    --disable CONFIG_LOCALVERSION_AUTO \
    --enable CONFIG_LBDAF \
    --disable CONFIG_XFS_FS \
    --disable CONFIG_GFS2_FS \
    --disable CONFIG_TEST_KMOD

# Compile the kernel for my ubuntu 24.04 kernel 6.17.0-20-generic.
#   HOSTCFLAGS=-fcommon  | host-side dtc fix (GCC 10+ -fno-common default
#                          clashes with duplicate yylloc in dtc-lexer/parser).
#   KCFLAGS=-march=armv7-a | Cortex-A9 target. Forces the kernel's
#                          arch-v7 cc-option fallback (which lands on
```

```
67 #                          -march=armv5t on gcc 13 hard-float toolchains)
68 #                          to be overridden.  Last -march wins.
69 $(ZIMAGE) : $(KERNEL_CONFIG)
70   $(CROSS) $(MAKE) -C $(KERNEL_DIR) LOCALVERSION= HOSTCFLAGS=-fcommon KCFLAGS=-march=armv7-a
     ↪  zImage
71
72 # clean
73
74 .PHONY : kernel-clean config-clean
75 kernel-clean :
76   rm -rf $(KERNEL_DIR)
77
78 config-clean :
79   rm -rf $(KERNEL_CONFIG)
```

References

[1] Ensoniq 5503 digital oscillator chip. URL https://6502.org/documents/datasheets/ensoniq/ensoniq_5503_digital_oscillator_chip.pdf.

[2] Midi 2.0 on linux, 2026. URL https://docs.kernel.org/sound/designs/midi-2.0.html#midi-2-0-on-linux.

[3] S. Arar. Fixed-point representation: The q format and addition examples. *All About Circuits*, 30, 2017.

[4] S. Edwards. Csee 4840 embedded system design lab 2: Using c, linux, sockets, and usb. URL https://www.cs.columbia.edu/~sedwards/classes/2026/4840-spring/lab2.pdf.

[5] K. Forinash and W. Christian. 14.1.2: Equal temperament, July 2020. URL https://phys.libretexts.org/Bookshelves/Waves_and_Acoustics/Sound_-_An_Interactive_eBook_(Forinash_and_Christian)/14%3A_Musical_Scales/14.01%3A_Musical_Scales/14.1.02%3A_Equal_Temperament.

[6] S. Hutchinson. The midi protocol: System messages — simon hutchinson, 2024. URL https://www.youtube.com/watch?v=y5Sah0bJYkg&t=14s.

[7] P. Johnston. Simple fixed-point conversion in c - embedded artistry, July 2018. URL https://embeddedartistry.com/blog/2018/07/12/simple-fixed-point-conversion-in-c/.

[8] H. Moller and C. S. Pedersen. Hearing at low and infrasonic frequencies. *Noise and health*, 6(23):37–57, 2004.

[9] E. W. Weisstein. Sawtooth wave. *MathWorld–a Wolfram Web Resource*, 2010. URL https://mathworld.wolfram.com/SawtoothWave.html.

[10] E. W. Weisstein. Square wave. *MathWorld–a Wolfram Web Resource*, 2010. URL https://mathworld.wolfram.com/SquareWave.html.

[11] E. W. Weisstein. Triangle wave. *MathWorld–a Wolfram Web Resource*, 2010. URL https://mathworld.wolfram.com/TriangleWave.html.