The Design Document for CSEE 4840 Embedded

System Design

Project: AudioSampler

Spandan Das UNI: sd3506

Avik Dhupar UNI: ad3910

Spring 2022

Spandan Das (sd3506)
Avik Dhupar (ad3910)

1. Introduction

We plan to build a sound sampler and playback system which supports the following features:

1. MIDI input using a USB-MIDI Keyboard
2. Sample storage
3. Sample Playback
4. Effects
    a. Delay
    b. Clipping distortion
    c. Reverse Playback
    d. Low-Pass filtering with 64 fixed configurations
5. (Tentative) Sample Recording using Mic / Line-In

We envision the system receiving an input from a USB-MIDI keyboard, which is processed by the HPS, and is forwarded through a memory mapped device to the FPGA fabric. The FPGA side loads a pre-saved sample through the HPS, or will use line-sampled audio, which will then be processed on the FPGA itself, since it will allow us to optimize for signal processing and real-time audio reproduction. Once the FPGA processes the audio, it will also control and drive the WM8731 audio CODEC, which allows for easier playback through the integrated DAC. The DE1-SoC is packed with any necessary audio signal processing circuitry, which will allow us to simply load the audio buffer into the WM8731 which can then be output on the 3.5mm audio output jack.

2. System Block Diagram

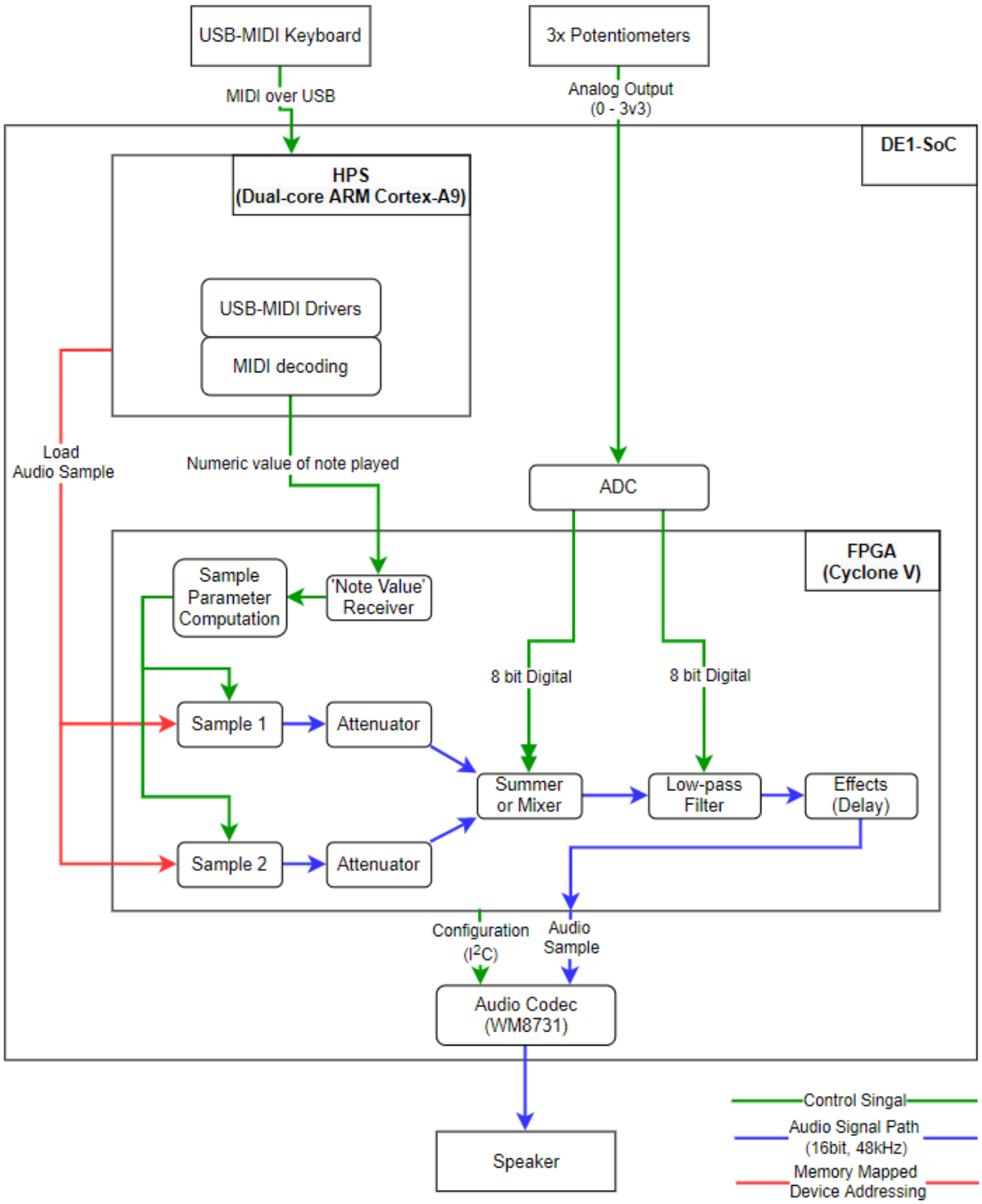


Fig. 1

Figure 1 above shows the system block diagram. Each component is described in the following sections.

USB-MIDI

Musical Instrument Digital Interface is an ancient standard that describes the communication protocols, digital interface, and electrical connectors to connect a variety of electronic musical instruments [1]. MIDI supports some of the key features required for any electronic musical keyboard. For example, key-press, key-release, clock signal (tempo), note's pitch, velocity and aftertouch. Additionally, MIDI also allows for generic and specific 'control-change' messages, which can denote parameters such as portamento, modulation wheel values, and effect amount.

For the lack of better standards in the past, most electronic musical instruments continue to support MIDI, over forty years after its advent. While today's computers do not directly support MIDI without specialized hardware such as sound cards or USB-MIDI converters, recent advancements have helped introduce support for MIDI over USB. In devices that support MIDI over USB, the underlying MIDI standard is maintained, i.e. the devices still speak the same protocol, however the MIDI data is encapsulated in a USB packet, and the electrical interface also follows the USB standard.

We will be using existing drivers for USB-MIDI to capture the data being generated by the MIDI keyboard. This data will then be enumerated in a limited range. For example, when the note 'C2' is pressed, this will be enumerated to 0. We intend to support 4 octaves, therefore, the note 'C6' will be the last playable note, and will be enumerated to 48. This enumeration will be performed on the HPS side and the values will be passed on to the FPGA over a memory mapped device. On the FPGA side, these values will be used to compute parameters for the sample playback.

Sample Parameter Computation

Once the FPGA receives the value of the note being played, with 0 corresponding to C2, and 48 corresponding to C5, we will perform computation to 'pitch shift' the sample in order to allow for 'playability' of the sample.
The specific algorithm details are yet to be determined, however we will stick with the 12-tone equal temperament scale, which means that an octave is divided into 12 parts, all of which are equal on a logarithmic scale, with a ratio equal to the 12th root of 2. [2]

Sample Storage

We will use the SD card for on-board storage of the samples. These samples will be pushed to the FPGA fabric during system initialization. During runtime, these samples will be stored in the embedded memory of the FPGA

Attenuator & Summer

Since we are using more than one sample during playback, a simple summation would lead to an increase in the amplitude. In order to maintain a fixed, maximum output level, we use weighted amplitude of the two signals. This is implemented using attenuators.
For example:

For two input signals, W_1 and W_2 , the amplitude for each is $A_1 = A_2 = 1$
On adding both signals, amplitude will increase to $A_1 + A_2 = 2$. This is undesirable, and would lead to a much louder output from the speakers.

We fix this by using attenuators, who's gain is determined by the mixer inputs. If the mixer is set to 0.75 for W_1 and 0.75 for W_2, the output should be weighted based on these values, such that the total output never exceeds 1. In this case, the output would be $0.75 \times 0.66 + 0.75 \times 0.66 = 1.0$

Low-Pass Filter

We plan to use the Altera FIR compiler IP which comes with the Altera IP toolbench interface. We can use this to implement a variety of filter architectures, including fully parallel, serial, or multibit serial distributed arithmetic, and multicycle fixed/variable filters. The FIR Compiler includes a coefficient generator.

In contrast to IIR filters, FIR filters have a linear phase and inherent stability. This benefit makes FIR filters attractive enough to be designed into a large number of systems. However, for a given frequency response, FIR filters are of a higher order than IIR filters, making FIR filters more computationally expensive.

The general steps to configure the filter are as follows:

a. Specifying the coefficients
b. Analyzing the coefficients
c. Specify the architecture specifications
d. Specify the input-output specifications

Avalon Streaming Interface(with filter):

The Avalon Streaming (Avalon-ST) interface defines a standard, flexible, and modular protocol for data transfers from a source interface to a sink interface and simplifies the process of controlling the flow of data in a datapath.

Avalon-ST interface signals can describe traditional streaming interfaces supporting a single stream of data without knowledge of channels or packet boundaries. Such interfaces typically contain data, ready, and valid signals. The Avalon-ST interface can also support more complex protocols for burst and packet transfers with packets interleaved across multiple channels.

The Avalon-ST interface inherently synchronizes multi-channel designs, which allows to achieve efficient, time-multiplexed implementations without having to implement complex control logic.

The Avalon-ST interface supports backpressure, which is a flow control mechanism where a sink can signal to a source to stop sending data. The sink typically uses backpressure to stop the flow of data when its FIFO buffers are full or when there is congestion on its output.

When designing a datapath which includes the FIR Compiler MegaCore function, we may not need backpressure if you know the downstream components can always receive data. We may achieve a higher clock rate by driving the ast_source_ready signal of the FIR Compiler high, and not connecting the ast_sink_ready signal.

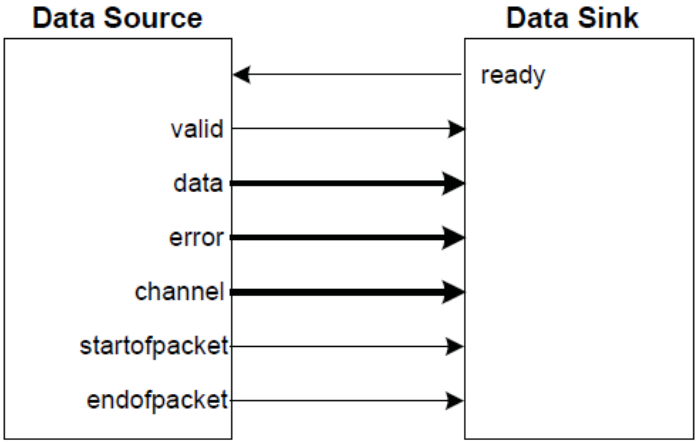
Table 4–1. Avalon-ST Interface Parameters

Parameter Name	Value
READY_LATENCY	0
BITS_PER_SYMBOL	data width
SYMBOLS_PER_BEAT	1
SYMBOL_TYPE	signed/unsigned
ERROR_DESCRIPTION	00: No error 01: Missing `startofpacket` (SOP) 10: Missing `endofpacket` (EOP) 11: Unexpected EOP or any other error

Table 4–2. Avalon-ST Interface Signal Types

Signal Type	Width	Direction
ready	1	Sink to Source
valid	1	Source to Sink
data	data width	Source to Sink
channel	$\log_2$(number of channels)	Source to Sink
error	2	Source to Sink
startofpacket	1	Source to Sink
endofpacket	1	Source to Sink

Figure 4–11. Avalon-ST Interface



Effects

We plan to implement simple effects such as delay, filtering and clipping distortion. The algorithms are described in Section 3: Algorithms.

ADC & Potentiometers

We sample potentiometers using the ADC to set parameters for the mixer and low-pass filter. Two potentiometers will be used to scale the amplitude of each sample, whereas a third potentiometer will be used to alter the cutoff frequency of the low-pass filter. Since we have 'banks' of filter, i.e. the cutoff frequency will be discretized into 64 steps, the potentiometer will be used to select one of the 64 predetermined cutoff frequencies.

Audio CODEC

DE1-SoC ships with an onboard audio CODEC, the Wolfson Audio WM8731. These devices are used to encode or decode audio signals using DACs and ADCs respectively. The datasheet provides specific details of the CODEC, which are as follows. [4]
WM8731 is a low power stereo CODEC with an integrated headphone driver. It also supports line and mono microphone level audio inputs, programmable line level volume control, and a bias voltage output suitable for electret microphones. The WM8731 uses stereo 24-bit sigma delta ADCs and DACs. It supports audio input word lengths from 16-32 bits and sampling rates from 8kHz to 96kHz. The CODEC can be configured using 2 or 3 wire bus. We will use the 2 wire I2C bus in our project.

3. Algorithms (Effects)

1. Audio Delay

Delay is an audio signal processing technique that records an input signal to a storage medium and then plays it back after a period of time. When the delayed playback is mixed with the live audio, it creates an echo-like effect, whereby the original audio is heard followed by the delayed audio. The delayed signal may be played back multiple times, or fed back into the recording, to create the sound of a repeating, decaying echo.

In a digital delay effect, audio is passed through a memory buffer and recalled from the buffer a short time after. By feeding some of the delayed audio back into the buffer, multiple repeats of the audio playback (feedback). High levels of feedback can cause the level of the output to rapidly increase (self-oscillation), becoming louder and louder; this may be managed with limiters.The delayed signal may be treated separately from the input audio - for example, with an equalizer.

Digital delay systems function by sampling the input signal through an analog-to-digital converter, after which the signal is passed through a digital signal processor that records it into a storage buffer, and then plays back the stored audio based on parameters set by the user. The delayed ("wet") output may be mixed with the unmodified ("dry") signal after, or before, it is sent to a digital-to-analog converter for output. [5]

2. Audio Clipping

In digital signal processing, clipping occurs when the signal is restricted by the range of a chosen representation. For example, in a system using 16-bit signed integers, 32767 is the largest positive value that can be represented. If, during processing, the amplitude of the signal is doubled, sample values of, for instance, 32000 should become 64000, but instead cause an integer overflow and saturate to the maximum, 32767. Clipping is preferable to the alternative in digital systems—wrapping—which occurs if the digital processor is allowed to overflow, ignoring the most significant bits of the magnitude, and sometimes even the sign of the sample value, resulting in gross distortion of the signal.

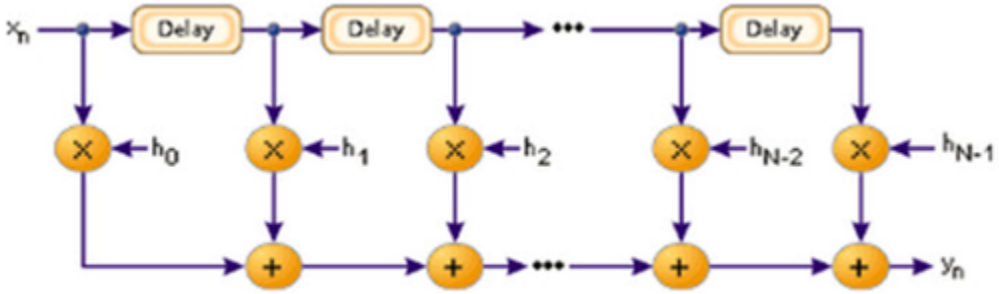
3. Reverse Playback

Reverse playback of an audio sample can be achieved by storing the audio sample in the buffer(memory). During playback the audio samples can be played back from the backend of the buffer storage.

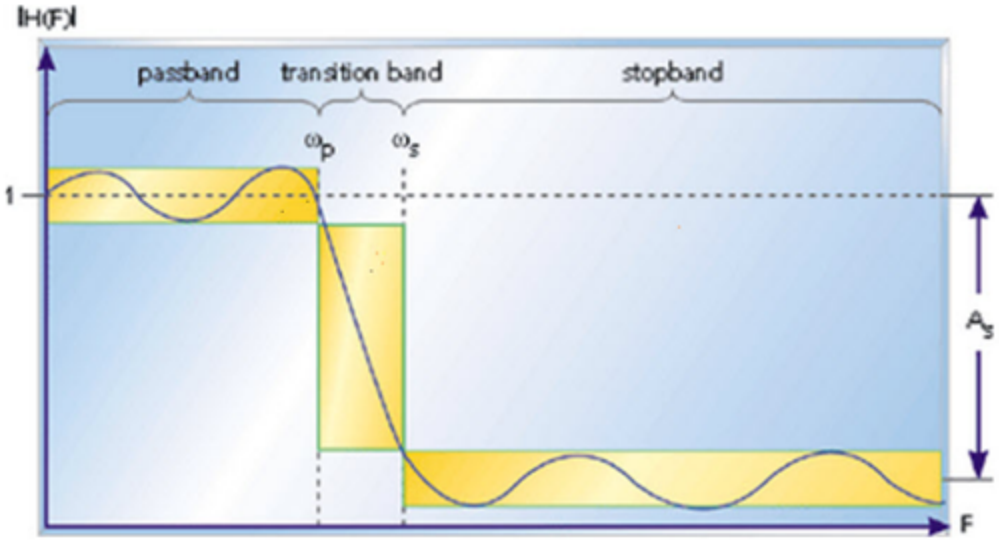
4. Low-Pass Filtering with 64 filter configurations

We plan to use an FIR filter IP already present in the IP designer to design a low-pass filter. We will use the many bank configurations in the IP to store the different filter coefficients.

In digital signal processing, an FIR is a filter whose impulse response is of finite period, as a result of it settles to zero in finite time. This is often in distinction to IIR filters, which can have internal feedback and will still respond indefinitely. The impulse response of an Nth order discrete time FIR filter takes precisely N+1 samples before it then settles to zero.



Logical Structure of FIR Filter



Frequency Response of an FIR Filter

The frequency response plot of the filter is shown below, where ωp is the passband ending frequency, ωs is the stopband beginning frequency, As is the amount of attenuation in the stopband. Frequencies b/n ωp and ωs drop in the transition band and are reduced to some lesser degree. That confirms that the filter meets the preferred specifications including transition bandwidth, ripple, filter's length and coefficients. The longer the filter, the more finely the response can be tuned. With the N length and coefficients, float h[N] = {……}, decided upon, the FIR filter implementation is fairly straightforward.

4. Resource Budgets

Planned Memory Usage on FPGA

64 Filter Banks

$$\underbrace{8}_{Bit\ Width} \times \underbrace{36}_{Coefficients} \times \underbrace{64}_{Banks} = 2.25\ KiB$$

2 Sample Banks

$$\underbrace{2}_{Sample\ Banks} \times \underbrace{16\ bit}_{Bit\ Depth} \times \underbrace{48000}_{Sample\ Rate} = 187.5\ KiB$$

Summer Output

$$\underbrace{1}_{Sample} \times \underbrace{16}_{Bit\ Depth} \times \underbrace{48{,}000}_{Sample\ Rate} = 96\ KiB$$

Filter Output

$$\underbrace{1}_{Sample} \times \underbrace{16}_{Bit\ Depth} \times \underbrace{48{,}000}_{Sample\ Rate} = 96\ KiB$$

Total = 2.25 + 187.5 + 96 + 96 = **381.75 KiB**

Total available embedded memory = 4450 Kbits = 552 KiB

While we initially planned to use 3 samples, we quickly realized that we will occupy more than 80% of the available embedded memory. We now plan to use only 2 samples in order to permit overheads that we haven't foreseen.

5. Hardware-Software Interface

The primary hardware-software interfacing will be done over the Avalon Bus.
We will also introduce a memory-mapped device for the sample, so the HPS will be able to directly write to this area and change the samples.

Avalon Bus:

Avalon interfaces simplify system design by allowing you to easily connect components in Intel® FPGA.

Avalon Memory Mapped Interface (Avalon-MM)—an address-based read/write interface typical of Host-Agent connections.

We can use Avalon Memory-Mapped (Avalon-MM) interfaces to implement read and write interfaces for Host and Agent components. The following are examples of components that typically include memory-mapped interfaces:

• Microprocessors
• Memories
• UARTs
• DMAs
• Timers

Signal Role	Width	Direction	Required	Description
address	1 to 64	Host to Agent	No	By default, the address signal represents a byte address. The value of the address must align to the data width. To write to specific bytes within a data word, the host must use the byteenable signal.
writedata	8,16,32...1024	Host to Agent	No	Data for write transfers. The width must be the same as the width of readdata if both are present. Required for interfaces that support writes.
waitrequest waitrequest_n	1	Agent to Host	No	An agent asserts waitrequest when unable to respond to a read or write request. Forces the host to wait until the interconnect is ready to proceed with the transfer. At the start of all transfers, a host initiates the transfer and waits until waitrequest is deasserted.

The USB-MIDI keyboard connects to the HPS over a USB bus. MIDI devices typically show up as HID devices, and we plan to use existing HID device drivers to access the bytes being sent by the MIDI device. Once we capture these bytes, they will be converted to a numeric value

between 0 to 48, since we support only 5 octaves ranging from C2 to C6. These numeric values will then be written to the FPGA through a memory-mapped device. We plan to reuse some of the existing code from Lab3 for this. While an existing driver may support various other MIDI controls, such as control change and modulation wheel, we will only send the note's numeric value, and note on/off signal.

6. Simulations

Simulink

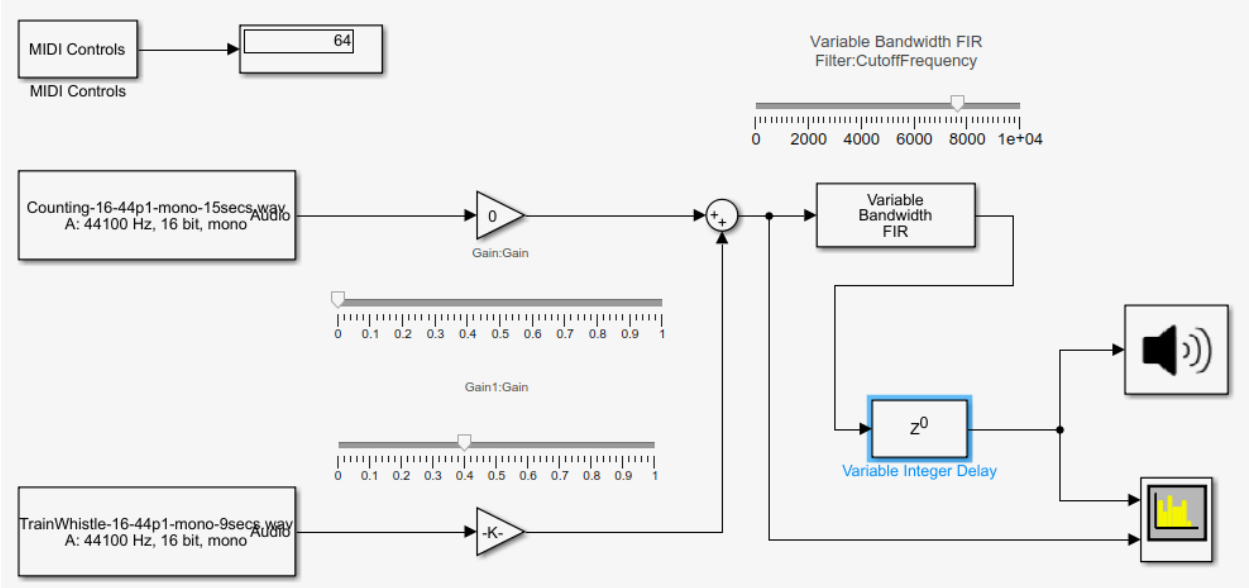


Fig 2: Simulink simulation diagram

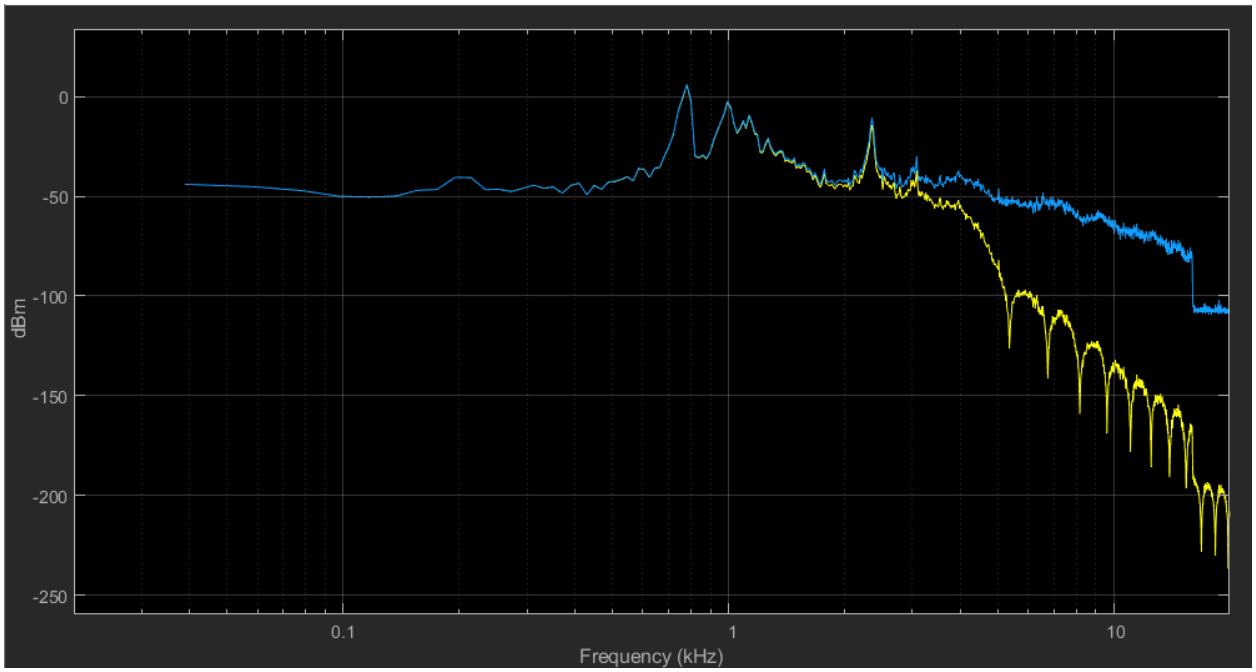


Fig 3: Spectrogram when Fcutoff = 2907 Hz

Legend:
Blue: Original Input
Yellow: Filtered and delayed output

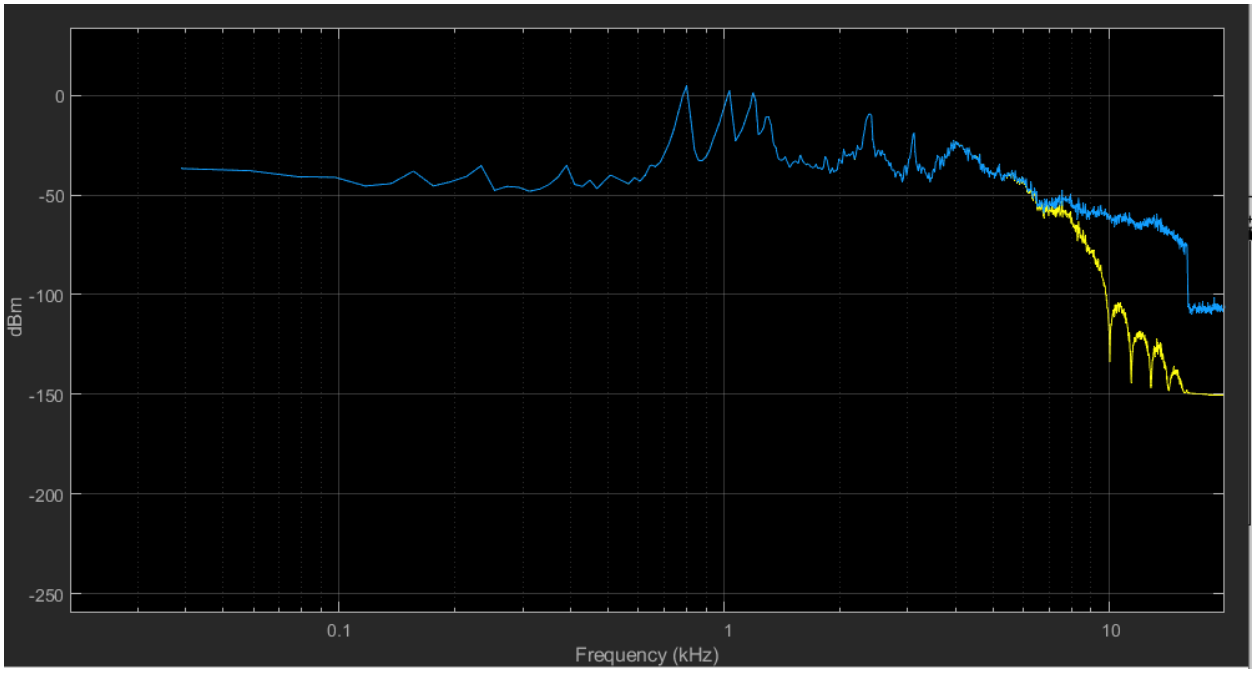


Fig 4: Spectrogram when Fcutoff = 7601 Hz

Legend:
Blue: Original Input
Yellow: Filtered and delayed output

VCV Rack

While the simulation in Simulink is sufficient to verify our design, our low-end Intel i5 systems clocked at 3.15GHz with 16GB RAM were not good enough to run the simulation without jitter. Therefore, to get a better understanding of how the system would sound, we created a VCVRack simulation.

VCVRack runs jitter free, and allows us to listen to the output. However, since it is focussed on music production, as opposed to scientific research, most components are (virtual) voltage-controlled, as is typical in the world of analog synthesizers. In our system, however, we will not have voltage control since the entire control and signal path is in the digital domain, except the CODEC output.

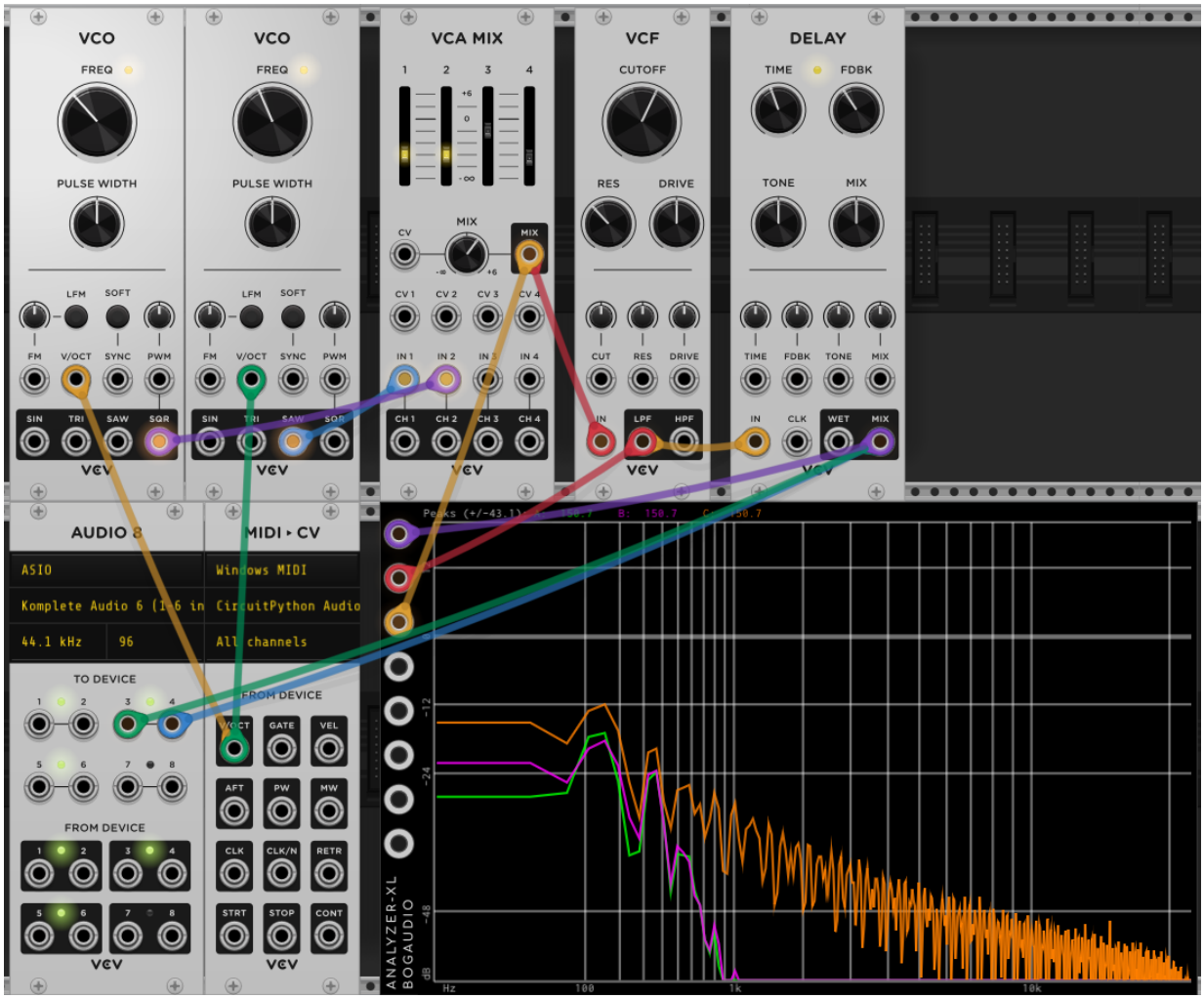


Fig 5: VCVRack simulation

Legend:
Orange: Unprocessed output of samples (Saw and square wave in this case)
Purple: Filtered output
Green: Filtered and Delayed Output

References

[1] https://en.wikipedia.org/wiki/MIDI
[2] https://en.wikipedia.org/wiki/Equal_temperament
[3] https://cdrdv2.intel.com/v1/dl/getContent/654277?explicitVersion=true
[4] https://www.digikey.com/en/datasheets/cirruslogicinc/cirrus-logic-inc-wm8731_v49
[5] https://en.wikipedia.org/wiki/Delay_(audio_effect)
[6] https://en.wikipedia.org/wiki/Clipping_(audio)
[7] https://www.elprocus.com/fir-filter-for-digital-signal-processing/