CSEE 4840: MIDI Wavetable Synthesizer

Doga Ozesmi (do2330)

Varun Varahabhotla (vv2282)

Lancelot Wathieu (lbw2148)

Evan Ziebart (erz2109)

1 Overview

1.1 Summary

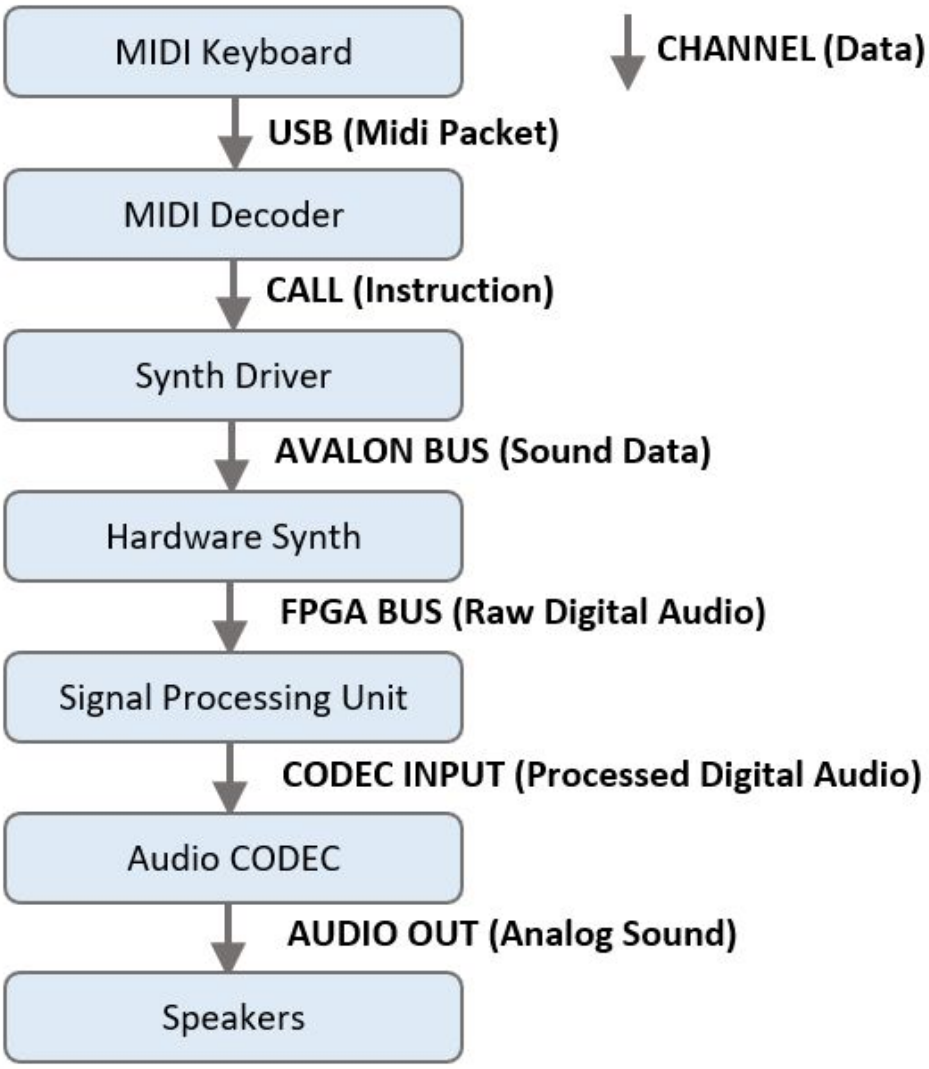
The purpose of this project is to design and implement a digital synthesizer on the Terasic Cyclone DE1 board. When connected to a speaker and keyboard, the board produces a sound of the corresponding pitch when a user presses a key, and the sound halts when a key is released. Additionally, there is an ability to modulate the kind of sound leaving the speakers on a key press and the amplitude of the sound wave, allowing for different instrument sounds and volumes from the synthesizer.

The two main types of synthesizers available are wavetable synthesizers and FM synthesizers. This project implements a wavetable synthesizer, meaning that the available instrument sound wave shapes are stored digitally on the device. This digital sound data can be played back at different rates in order to produce different frequencies on the same instrument. Moreover, multiple wavetables and frequencies can be combined or averaged to produce a richer variety of sounds from the device than just those stored in the wavetables.

The Musical Instrument Digital Interface (MIDI) protocol is used to send information about key presses from the keyboard in order to communicate the requested sounds. On a key press or release, a MIDI packet is sent from the MIDI keyboard over a USB connection to the device. A software program receives and interprets the MIDI packet, and then it sends instructions from user space to a kernel space device driver for the hardware synth. The device driver can then channel the appropriate data through the Avalon bus to the hardware synth implemented on the FPGA.

The hardware synth will hold 10 different channels which support 10 different sounds (keyboard presses) from the synth at one time. These are sampled digitally from the wavetables. The various signals are sent through a processing unit, which modifies and combines them into one snippet of sound data. This data can then be sent to the audio codec on the chip to produce a sound.

1.2 Block Diagram



1.3 Design Overview

Hardware Systems

- Wavetable synth
- Signal processing and sending to audio CODEC

Software Systems

- MIDI message converter
- Synth device driver

1.4 Technologies

Hardware Systems

- DE1-SOC (Audio Codec, FPGA, Processor, Avalon Bus, SRAM)
- Speaker
- MIDI keyboard

Software Libraries

- Libusb

2 Design Subsystems

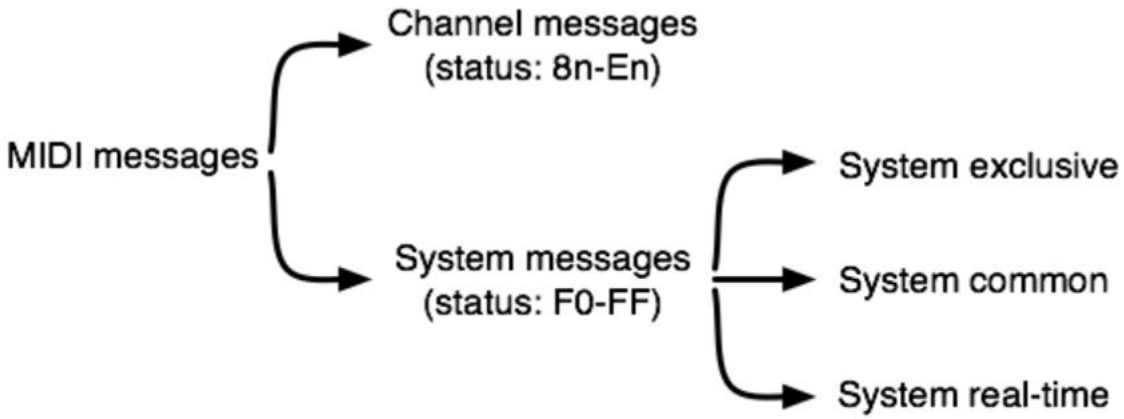
2.1 MIDI Decoder Program

At a high level the MIDI decoder program is responsible for taking the MIDI Protocol messages (described in section 3.1) utilizing the Libusb software library and translating the instructions into logic output. The logic output would be applied to synth commands. The other information which we will use within the software pipeline itself is the synchronization information providing context of where changes should be made.

MIDI Messages

There are two types of MIDI messages, channel and system. Channel messages are sent to individual channels (of which there are 16) and system messages are sent to all channels. Amongst the two messages the channel messages are primarily responsible for the generation

of keys, while system messages amongst other things are utilized for synchronization.



Data To Extract
We will primarily be utilizing the key press and key release events data(described in 3.1).

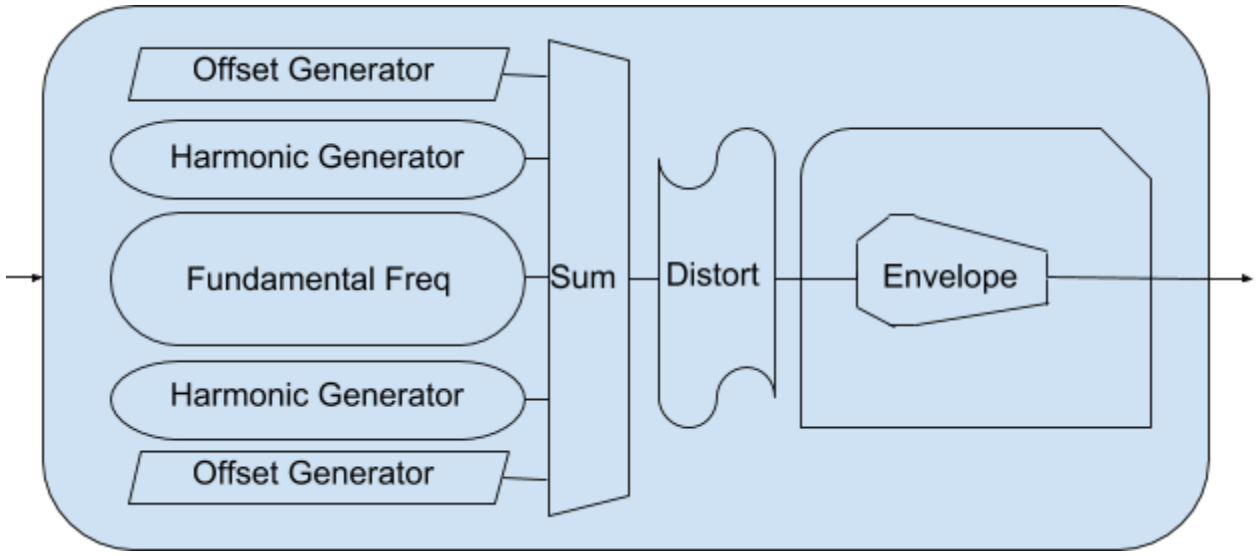
MIDI Synchronization
MIDI handles timing and synchronization of data through system real-time messages. There are two types of synchronization that MIDI handles which are music timing data(bars and beats) along with timecode information(which is needed to sync in “real time”). For our project we will primarily be utilizing the timing data to understand where to insert new music and to implement changes.

Source: https://www.nyu.edu/classes/bello/FMT_files/9_MIDI_code.pdf

2.2 Synth Software Driver

The Synth Software Driver will take the relevant actions from the MIDI Decoder and send the data manipulated in the raw form to the bus using the ioctl system call. This will function as the device driver for the hardware synthesizer implemented on the FPGA, and will set up in kernel space on the chip.

2.3 Wavetable Signal Generator



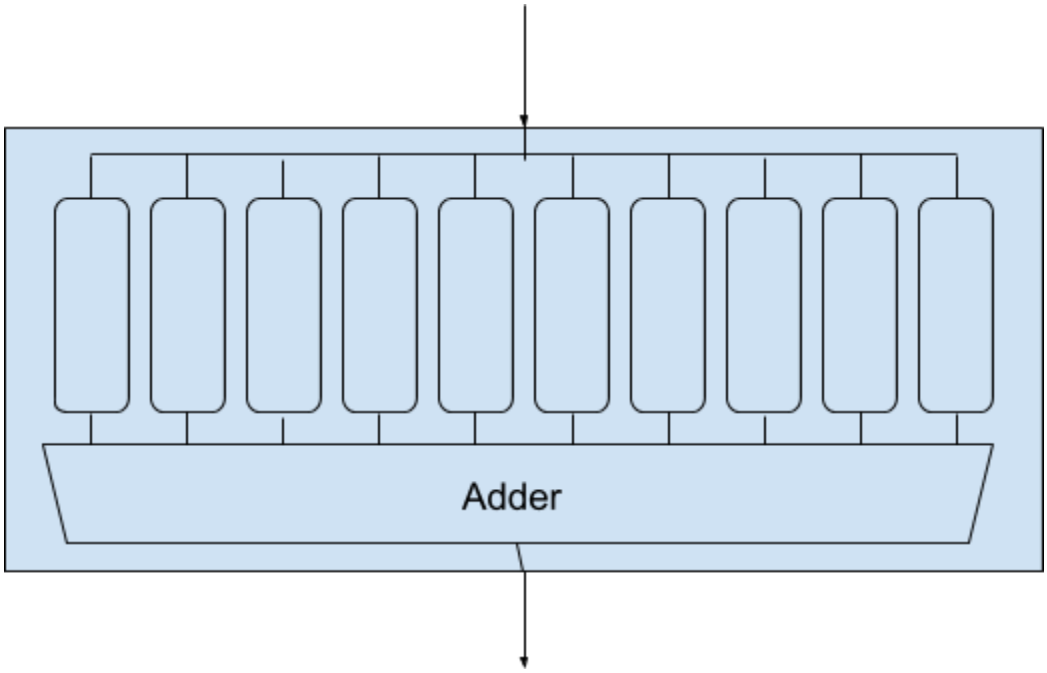
The image above shows the process of generating the sound wave for an individual instrument. The instrument receives what type of instrument it is and what note it should play. Each instrument profile is described by some combination of 5 oscillators. Based on this information the instrument sums the fundamental frequency of the note as well as some combination of harmonic frequencies. These three central oscillators are sinusoidal generators which read from a wave table to determine what value to pump out next. In contrast the offset generators produce a triangle, sawtooth, or square wave dynamically from looking at the value of a counter. The output of these 5 oscillators are then summed together and then their combined output is fed through a filter which introduces distortion by adding a soft limit to create clipping or adding noise.

Audio amplitude levels are encoded in 16 bits and we will have the CODEC sample at 50 kHz in order to sample at well over twice the maximum frequency that humans can hear. This means that 800kb are sent per second. The lowest frequency that humans can hear is 20hz and as a result a single sine wave at this frequency will be sampled 2500 times for a total of 5kB needed to store the points of a sine wave. This table of values that make up a sine wave can be referenced to create a sine wave at any audible frequency.

For the sinusoidal wave table, we will use 5 kB of SRAM memory. This is because, as stated above, we will store a sine wave at 20 hz. This means (50 kHz/20 Hz)*(16 bits)*(1 Byte/8 bits) =

5 kB of the wave. With the 500 kB of SRAM in our FPGA, we have more than enough leeway for other complex waveforms we like or to allow for other systems to use the SRAM if we find out it is needed. These will be produced from MATLAB, as stated in our Milestone 1.

2.4 Digital Signal Processing Unit



The high level processing logic is shown above. Each instrument is told what note to play and what type of instrument it is, ie what combination of waveforms to use as in part 2.3. Then the outputs of each instrument are summed together and forwarded to the audio CODEC. A maximum of 10 instruments can play at the same time, corresponding to 10 fingers playing 10 notes on a piano keyboard.

To do the summing a special formula has to be used. For example if two instruments are played at the same time and if their outputs are linearly added there will likely be clipping. However if they are weighted equally, where each output is cut in half and then they are summed this will not sound the same as two instruments being played together. The total amplitude of the audio sent to the CODEC should be higher than one instrument but also prevent clipping. Therefore to determine the amplitude adjustment on each instrument a formula like the following will be used where n is the number of instruments being currently played.

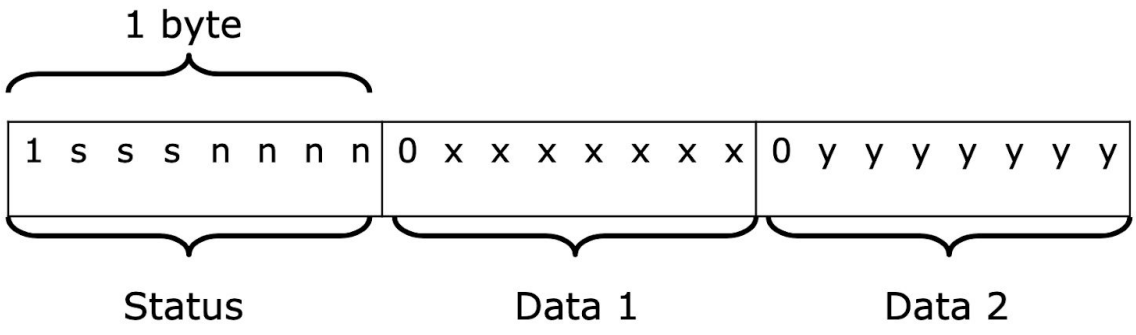
Amplitude coefficient = 3n/(n+2) (as a reasonable example)

3 Interfaces

3.1 MIDI Messages

A MIDI message reports an event that has taken place with regards to the user's interaction with the MIDI instrument, such as a key press, volume adjustment, pitch bend, etc. A message is made up of several MIDI byte packets sent one at a time.

There are two kinds of bytes received as a part of a MIDI message: status bytes and data bytes. Status bytes always appear at the beginning and have a 1 in the most significant bit. Their purpose is to report the type of message being sent. Data bytes are arguments to the status byte and always have a 0 in the most significant bit. If the status byte remains unchanged form the previous message, a MIDI instrument will often omit the status byte to save time. In this case, the status is interpreted as the last status received from the instrument, also called the "running status".



For our purposes, the only two types of message that matter are the key press and key release events. The format for both type of message is shown below.

Event	Status	Data
Key Press	1000nnnn	0kkkkkkk 0vvvvvvv
Key Release	1001nnnn	0kkkkkkk 0vvvvvvv

Here, kkkkkkk says which key is being affected and vvvvvvv reports the velocity at which it is pressed/released. Additionally, nnnn reports the channel on which the note change should take affect.

The two data bytes: key and velocity, are used to determine the note frequency and amplitude respectively. The appropriate frequency can be determined from the equation $freq = 440 \cdot (\frac{key-69}{12})$. There is no prescribed way to determine amplitude from velocity, but one of our sources provides a table which relates certain velocity values to musical notation indicating an appropriate volume.

pppp = 8

ppp = 20

pp = 31

p = 42

mp = 53

mf = 64

f = 80

ff = 96

fff = 112

ffff = 127

Sources:

- https://www.midi.org/specifications-old/item/table-1-summary-of-midi-message
- http://www.inspiredacoustics.com/en/MIDI_note_numbers_and_center_frequencies
- https://www.cs.cmu.edu/~music/cmsip/readings/MIDI%20tutorial%20for%20programmers.html

3.2 Synth Driver Instruction Set

We will implement 4 instructions which a user-space program can call to communicate with the synth device driver in kernel-space. There are two commands to set and read sound data from one of the 10 channels. Sending sound data to a channel is the main mechanism by which synth will receive instructions about what sounds to play from the software. The command to read this data is not used in the final system but is useful for testing purposes.

The two other commands instruct the synth to play or stop playing a sound on a given channel. These essentially serve as an indicator of whether or not the sound data stored in a channel is valid for playing through the speaker. When a key is released, for example, it is easiest to disable the playing of that channel rather than overwriting the sound data stored there.

Summary

- set_sound(channel, sound)
- read_sound(channel)
- play_sound(channel)
- stop_sound(channel)

3.3 Synth Hardware Data Format

To specify a sound on the synthesizer, it is necessary to specify several parameters. Two of them are amplitude and frequency, which can be directly translated from the MIDI data received in software. Under this scheme, those two parameters take 7 bits each, totalling 14.

In addition, it is necessary to specify the instrumental sound which is being played with the note. Our wavetable synth is going to sample from 4 types of fundamental waves: sine, triangle, square, and sawtooth. Different instrument sounds are produced by linear combinations of the sounds played from these four wavetables. The instrument is specified, therefore, from

coefficients which give the relative strengths of these fundamental instrument sounds. The higher the coefficient, the more prominent the wavetable will be in the final sound. Each coefficient is specified up to 4 bits of precision.

Summary

- Amplitude (7 bits)
- Frequency (7 bits)
- c_sin (4 bits)
- c_tri (4 bits)
- c_sqr (4 bits)
- c_saw (4 bits)

TOTAL = 30 bits

3.4 Audio CODEC Configuration and Protocol

To configure the Audio Codec Chip on the DE1 board, we can use both the spec sheet for the WM8731 chip and some example files from the DE1 manual. The folder “DE1_SOC_i2sound” has many examples of verilog to load onto the FPGA. For example, the karaoke machine example (section 5.3 of the DE1-SoC Manual) sets the WM8731 chip inputs such that it blends the audio inputs from the Mic-In and the Line-in, and outputs to the Line-out. It even uses the pushbutton KEY0 to cycle through volume settings. We can add to and change verilog from this project to configure the codec such that it can read in digital audio signal from our hardware and output it to speaker. This will be tested as part of Milestone 1.

4 Milestones

4.1 Milestone 1

1) MATLAB proof-of-concept:
    a) Make the sine, sawtooth, pulse, and other waveform tables in MATLAB
    b) Synthesize different example frequencies and waveforms added together
    c) Use MATLAB to play the audio signals
2) Synthesize a square wave: Write the verilog to setup the codec chip, and send a square wave (can implement with timer) to a speaker
3) Be able to read(not use) MIDI Data in software

4.2 Milestone 2

1) Transfer MATLAB tables to the wavetables in verilog
2) Get frequency (and instrument) information from software, and write software/hardware driver to sample the wavetables and output that audio with verilog

4.3 Milestone 3

1) Finish up, add different wavetables, test instruments
2) Aim to have a working MVP and use the remaining two weeks to debug any issues and prepare for a demo.

5 Challenges and Roadblocks

1. One challenge which still needs to be addressed is which configuration is best to use for the audio codec. There are a few different available methods available for sending digital audio to the codec. At the moment, a configuration from the hardware seems more reasonable since the audio synthesis is happening in hardware.
2. Another challenge is the storage of wavetable data for the hardware synthesizer. Our current design seems to only require storage of sine data points (as the rest of the fundamental sounds can be generated dynamically). The current plan is to have this data on SRAM. In this case, the challenge is to find a way to retrieve data from the SRAM without significant slowdown of the circuit. Moreover, if for some reason more wavetables become required, it can increase the memory usage, which would require some reworking of the current model.