Figure 1: Melvin

TerrorMouse - A MIDI Synthesizer

Ron Weiss
rjw98@columbia.edu

Gabriel Glaser
gg389@columbia.edu

Scott Arfin
ska29@columbia.edu

April 1, 2004

1 Overall Architecture

Our project is designed to utilize both hardware and software resources. The first piece of real hardware we need is a cable that will allow us to connect the MIDI output of the keyboard to the RS-232 jack on the XESS-300 board. We will require hardware designed with VHDL to recieve the MIDI input from the keyboard and create a memory-mapped FIFO accessible from the software running on the Microblaze. This software will be used to interpret the MIDI signals and control the hardware synthesizers based on the MIDI input. See figure 2 for a block diagram. We will be using the onboard AK4565 audio codec to output our digitally synthesized sound to the headphone jack. The FPGA clock frequency is 50 MHz, and the audio clock frequency for our application is 24.414 KHz. This means that a new sample of digital audio must be prepared for output every 2048 cycles of the FPGA clock. This is the critical computational limit imposed on our hardware synthesizers.

2 MIDI Interface

The MIDI protocol includes specifications for the generation of many different signals pertaining to musical instrument sounds. The only signals we are interested in processing are 'note on' and 'note off' signals. We intend to ignore everyting else (as allowed by the MIDI specification). Our MIDI controller, the Casio CTK491, does not appear to be velocity sensitive, which will limit the functionality of our synthesizer. To deliver the MIDI signals to the XESS board, we will build a MIDI to RS-232 adapter cable. MIDI uses current loops to transmit information. Therefore we will need to build an electronic circuit to sense the currents in the loop and produce corresponding RS-232 voltages. There will be optical isolation between the current sensing and voltage generating portions of the circuit. We will develop our own serial controller because the MIDI protocol requires different baud rates than RS-232. We only need to recieve MIDI signals from the keyboard, so the controller will be unidirectional.

3 Computer Software

This module acts as the bridge between the MIDI interface and our hardware synthesizers. It will interpert the MIDI signals and translate them into commands for the FM synthesizer or the waveguide. It will allow only one of these devices to be sending signals to the DAC at a time (MUX in block digram). The computer software module will also be repsonsible for keeping track of the number of active notes since there are a limited number of voices available on the FM synth and the digital waveguide. When an additional

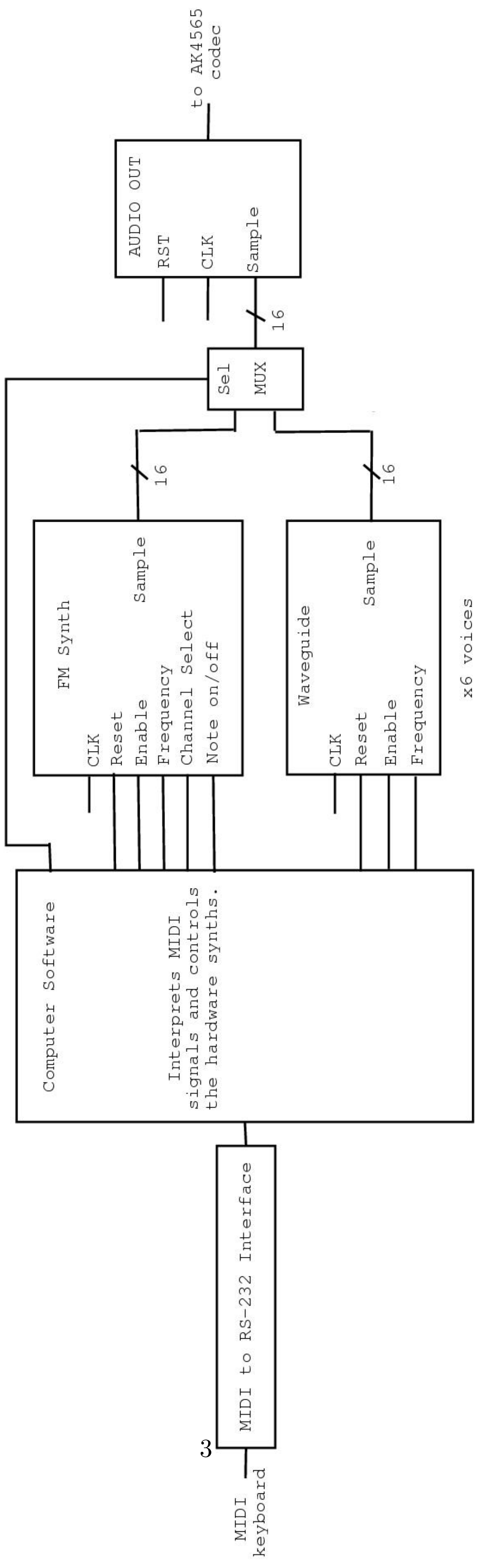

Figure 2: Overall Architecture of TerrorMouse

simulatenous note is pressed on the keyboard, the controlling software will assign the note to the next available channel. If none are available, the note event will be ignored.

4 Digital Waveguide Sound Synthesis

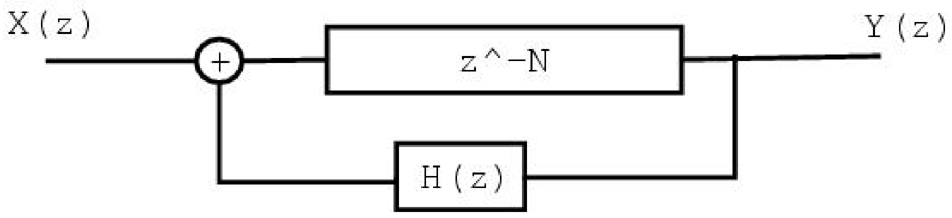

Figure 3: Digital Waveguide

4.1 Overview

This algorithm simulates the motion of a standing wave on a rigidly terminated string. This is implemented as a digital waveguide which consists of a delay line whose output is filtered and fed back to the input. The size of the delay line, N, determines the length of the 'string', and thus the output frequency. With the correct loop filter, $H(z)$, and appropriate input, the basic string sound can be augmented to sound like a specific stringed instrument.

4.2 Definitions

N = sampling frequency/desired frequency. This needs to be varied depending on the desired note. N must be an integer since it is impossible to implement a fractional delay. However, the sampling frequency is not divisible by all possible frequencies, so the synthesized sounds will sometimes be slightly out of tune.

$H(z)$ - Loop filter that shapes the waveform over time, providing frequency dependent damping and shaping the note's harmonics to mimic those of an acoustic guitar.

$X(z)$ - Waveguide input - corresponds to the string excitation (eg. the 'attack' on the string). This has the biggest effect on the sythnesized sound. The input must have a very short duration (about the same as the length of the delay line) or it will dominate the output. Early digital waveguides used white noise at the input which provides a very generic string sound. Instead, we will use an excitation signal corresponding to the pluck of a guitar string.

The above parameters have already been tuned to sound like an acoustic guitar in a Matlab implementation of this algorithm:[1]

$$H(z) = \frac{0.8995 + 0.1087z^{-1}}{1 + 0.0136z^{-1}}$$

4.3 Implementation

The waveguide is programmed in VHDL using a large RAM for the delay line. As with the rest of the project, each sample is 16 bits wide. The delay line is 256 x 16 bits. This gives us a minimum frequency of $\frac{f_s}{256} \approx 95$ Hz (F2), comparable to the frequency of the lowest note on a guitar (E2 is 82 Hz). This uses up one of the block RAMs available on the Xilinx FPGA. The loop filter $H(z)$ is a simple 1st order IIR filter. Finally, the excitation signal will be stored in a ROM to initialize the delay line whenever a new note is played. The only input needed for this module will be the desired frequency. Our synthesizer will make use of six of these waveguides to provide six string voices (as in a guitar). To do this the waveguide module will be duplicated six times. The output of each will be summed to mix all six signals.

5 FM Synthesis

FM Synthesis is a technique for generating sounds with rich harmonic content with relatively low computational expense. With the right choice of carrier and modulating signals, different musical sounds and tones can be created to great effect. It is our goal to make use of an FM synthesizer as one of the methods for generating musical sounds for our project.

5.1 Mathematical Basis

The basic FM equation is:

$$x(t) = A(t)\cos\Big(\omega_c t + I(t)\cos(\omega_m t + \phi_m) + \phi_c\Big)$$

$A(t)$ is the envelope, and in the simplest case may be set to a cconstant. $I(t)$ controls the depth of the modulation. It may also be constant, and its exact value is not critical. ω_c and ω_m are the carrier and modulating frequencies, respectively.

[1]Weiss, Ron and Steven Sanders“Synthesizing a Guitar Using Physical Modeling Techniques” http://www1.cs.columbia.edu/~ronw/dsp/

5.2 Implementation

In our project, we will be using a VHDL hardware module to perform the aforementioned compuations. Sine waves of different frequencies will be generated from a high resolution ROM containing the value of $\cos(x)$ for many values of x. The exact number of values in the ROM is an implementation detail to be determined experiementally, and will be a compromise between size minimization of the ROM and distortion minimization of the synthesized sound. The same ROM can be used to generate sinusoids of arbitrary frequency by indexing the ROM at position $int(f * x)$, where f is the normalized frequency, represented as a fixed point binary coded decimal, and $int(\alpha)$ truncates any decimal part of α. The decimal truncation is required due to the finite size of the cosine ROM and will result in distortion and the production of experimentally observed high frequency artifacts. A digital lowpass filter may be implemented if the artficats are severe.

5.3 Computational Complexity and Polyphony

Any good sythesizer has the ability to produce multiple tones simultaneously. To see if this is possible with our system, we must discuss timing. As discussed earlier, a new sample of audio output must be prepared for output once every 2048 cycles. In our preliminary tests, we found that generating the next sample of a simple sine wave could be done in as few as two clock cycles, and that adding additional sinusoids to the mix required an additional 3 clock cycles per sinusoid. For FM synthesis, more steps are involved. Simple multiplication may be required to achieve different modulation depths anenvelopes.

To generate a sample, the cosine rom must be accessed twice per note in every 2048 clock cycles. Some multplication is going to be necessary as well. It's hard to guess exactly how many clock cycles will be needed to synthesize each note, but it seems apparent that there is enough time to synthesize a reasonable number of notes, say 16.

6 Audio Output Module

6.1 General Description

This module reads a 16 bit sample from its input, and outputs it one bit at a time to the onboard DAC. In actuality, the onboard DAC is expecting stereo audio in two channels. To produce a mono signal, we must output the 16 bit sample two consecutive times - once for the left and once for the right. The DAC is also expecting sound samples to arrive at 48.8kHz. We have chosen to support audio at 24.4kHz to give reasonable signal processsing time in between samples as demanded by the digital waveguide alogorithm.

We must therefore repeat each sample an additional two times, or four times total, in order to halve the sampling frequency and produce mono sound.

6.2 Implementation

The DAC requires 3 clock signals.

MCLK is a 12.5 MHz 'master clock.'

BCLK is a 1.5625 MHz clock (this is the serial clock, which determines how fast the sample bits are output to the DAC).

LRCK is derived by dividing down the BCLK, to obtain a 48.828 khz clock. ($\frac{1.5625}{2^5}$MHz = 48.828 kHz)

This is used by the DAC to time between individual left and right audio signals. However we will generating monoaural sound, so we will be producing 24 samples per second by holding each sample for twice as long in the shift register. Every 16 cycles of BCLK, LRCK shifts to indicate that a new

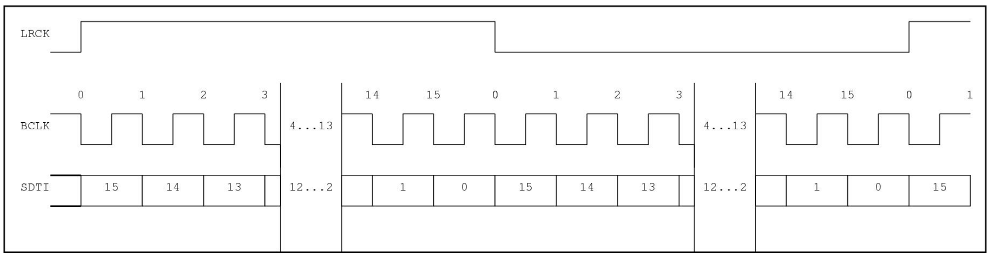

Figure 4: Timing for the Audio Codec

sample is being output. The STDI line represents how the highest order bit of each sample is output first, followed by the next 15, one at each BLCK cycle.