MULTICORE, MULTITHREADED, PHASE-SYNCHRONOUS FM SOUND SYNTHESIZER

by

Justin Tomlin

Senior Project

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo

2011

Table of Contents

List of Figures.....3
Acknowledgements.....4
Abstract.....5
Section 1 - Introduction.....6
    Project Overview/Motivation.....6
    Definition of Intended User.....6
    Competitive Solutions.....6
Section 2 - Background.....7
    FM Synthesis.....7
    Common Parameter Modulation Schemes.....7
Section 3 - Requirements.....8
    Functional Requirements.....8
    Performance Specifications.....8
Section 4 - Design Approach Alternatives.....9
    Oscillator.....9
Section 5 - Product Design.....14
    User Interface.....14
Theory of Operation.....15
    Software Subsystems.....19
    Software/Hardware Interfacing.....22
Section 6 – Physical Construction and Integration.....23
    Circuit Board Implementation.....23
    Enclosure Concept.....23
    Power Source.....23
Section 7 – Integrated System Tests and Results.....25
    Project (System-Level) Verification Testing.....25
    Subsystem Verification Testing.....25
    Summarized Test Results.....31
Section 8 – Conclusions.....32
    Proposed Solutions to Design Shortcomings.....32
    Further Improvements.....33
Section 9 – Bibliography.....35
Appendix A: Parts List and Cost.....36
    Budget.....36
    Parts List:.....36
Appendix B: Schedule.....37
Appendix C: Program Listing.....39
Appendix D: Analysis of Senior Project Design.....51

List of Figures

Chapter 4

4.1: DPW Sawtooth Generation 9
4.2: tanh Square Generation..... 10
4.3: tanh() function..... 10
4.4: Basic ModFM flowchart..... 11
4.5: Square to Triangle Conversion..... 11
4.6: Phase-synchronous ModFM..... 12

Chapter 5

5.1: Black Box Diagram..... 14
5.2: Photograph of Physical Interface..... 15
5.3: Layout of Parameter Matrix..... 15
5.4: Overall System Block Diagram..... 16
5.5: Audio DAC Connection Diagram..... 17
5.6: ADC Connection Diagram..... 17
5.7: MIDI Interface Schematic..... 18
5.8: Power Supply Schematic..... 18
5.9: Software Subsystem Block Diagram..... 19
5.10: I/O Port Map..... 22

Chapter 6

6.1: Photograph of Board Implementation..... 23
6.2: Photograph of Panel Wiring..... 24

Chapter 7

7.1: Power Supply/DAC Load Testing..... 25
7.2: AVDD Supply Noise..... 26
7.3: AVSS Supply Noise..... 26
7.4: Frequency Spectrum of 440 Hz Cosine Output..... 27
7.5: Frequency Spectrum (fo = 1760, k = 63, n = 7)..... 28
7.6: Envelope Modulation of Amplitude (short and long attack/decay)..... 28
7.7: Envelope Modulation of k (long attack, A440, n=2)..... 29
7.8: LFO Modulation of Amplitude (short and long period)..... 29
7.9: LFO Modulation of k (long period, A440, n=2)..... 30
7.10: Plot for Determining Audio Latency..... 30
7.11: Performance Specification Test Results..... 31

Acknowledgements

First of all, I would like to thank my senior project advisor Dr. Pilkington for his assistance in planning this project and addressing issues that came up along the way. The digital signal processing techniques and MATLAB exercises taught in his EE 419/459 lecture and lab were also invaluable in my research and design process.

Secondly, I would like to thank Texas Instruments for fulfilling my multiple sample requests while I was trying to find suitable components to interface with my microcontroller.

Finally, I would like to acknowledge Victor Lazzarini and Joseph Timoney for their work in developing the Modified FM synthesis algorithm. Without their work, this project would not have been possible.

Abstract

This project implements a phase-synchronous FM synthesis algorithm in hardware. Envelope and low frequency oscillator modulation of oscillator parameters is implemented. The microcontroller the system is based on, the XMOS XS1-G4, allows for physical parallelism including features such as multiple cores, multiple hardware threads on each core, a hardware event-driven thread scheduler, and channel, channel ends, and link switches for thread and core communication. The event-driven architecture of this device was ideal for implementing this synthesis algorithm. The final product is portable, durable, has a simple, intuitive user interface, and allows for extensive spectral shaping capabilities. The basic functional requirements outlined were successfully implemented. Testing, however, revealed a dynamic range that is out of specification. Some methods of reducing analog and digital noise are proposed in the Conclusions section.

Section 1 - Introduction

Project Overview/Motivation

With the advancement of digital electronics and drop in monetary cost of computational power, digital sound synthesis quickly became a cost-effective alternative to its analog counterpart. Time-to-market is heavily reduced due to the efficiency of embedded development. This opens up resources for research and development into intuitive interface designs and the exploration of novel sound synthesis and processing techniques. As a result, designers have more freedom to innovate in the sound synthesis field and tailor products to very specific needs.

This project is focused on designing a high-quality, affordable microcontroller-based sound synthesizer, which maximizes functionality and flexibility. To maximize functionality and minimize cost, efforts will be put into clever interface design and efficient DSP algorithm implementations.

Definition of Intended User

This project is a digital desktop sound synthesizer intended to be used by a musician either in a recording/production situation, or during live performance. The final product must have an intuitive user interface capable of making real-time sound alterations with no appreciable latency, and must have durable and portable packaging. This project represents the signal generation portion of a broader digital sound synthesis solution. On the input side, a MIDI-capable controller (keyboard, sequencer, or computer interface) is required to control the device. On the output side, either an amplifier and speaker, or an audio recording device, would typically be connected.

Competitive Solutions

- Waldorf Blofeld: Simple and “lean” interface, compact, portable, parameter matrix, externally controllable
- microKorg: Virtual analog capability, compact, portable, parameter matrix, sequencing, vocoding, keyboard, wide variety of oscillator algorithms

With regards to the interface, the Waldorf Blofeld is the most similar to the envisioned final product, implementing efficient yet intuitive parameter control in a lightweight package. However, this project intends to provide a much cheaper alternative to the Blofeld's $600 price tag while retaining professional sound quality and allowing for broad sound sculpting capabilities and complex sound generation.

The microKorg also employs clever interface design and has the advantage of an on-board keyboard, making it an attractive all-in-one solution. However, a standalone desktop synthesizer has the advantage of easy integration into existing MIDI systems, allowing for maximum flexibility and sparing the cost of the keyboard.

Section 2 - Background

FM Synthesis

Classic analog synthesizers rely on the principle of subtractive sound synthesis to generate sound. This involves taking a spectrally rich waveform, and applying subtractive (low pass, high-pass, and bandpass) filters to customize the output frequency spectrum, and thus the "timbre." This architecture has been carried forward into the digital domain, where it is a fairly efficient synthesis technique. However, this project will implement an algorithm associated with the digital domain, and rarely implemented in the analog domain, frequency modulation synthesis.

Theory of Operation

FM synthesis borrows from the theory of frequency modulation widely used in the communications domain, but utilizes carrier and modulator frequencies that lie within the audio frequency band. FM synthesis in its most basic form has the following form:

$$y = cos[w_c t + k*sin(w_m t)]$$

where w_c is known as the carrier frequency, and w_m is known as the modulator frequency. The index of modulation, k, determines the "strength" of the modulation, and an increase in this parameter results in more spectral brightness (stronger high frequency harmonics). For harmonic spectra, the ratio of carrier frequency and modulator frequency is kept strictly integral. However, to generate inharmonic spectra, useful for emulating bell and percussion sounds, ratios with fractional components can be used. FM synthesis is useful for generating complex sound while minimizing computational complexity. The oscillator itself contains parameters for spectral shaping, relaxing the need for filtering or additive synthesis for forming more complex timbres.

Common Parameter Modulation Schemes

Low Frequency Oscillator (LFO)

This subsystem consists of a low frequency waveform (usually < 20 Hz) used for parameter modulation. This is commonly used to modulate filter cutoff/center frequency and output amplitude, but is sometimes assignable to other parameters. The frequency and amplitude of the LFO are variable, as well as the wave shape.

Envelope Generator

This subsystem modulates the attack time, decay time, sustain level, and release time of various parameters. Again, this is most commonly used to modulate filter frequency and amplitude, but is sometimes assignable to other parameters. This corresponds to 555 timer or comparator/flip-flop retriggerable one-shot based circuits which serve the same purpose in analog synthesizers.

Section 3 - Requirements

Functional Requirements

- At least one oscillator capable of generating spectrally dense waveforms with wave-shape options and minimal undesirable artifacts (aliasing, heterodyning, etc...)
- MIDI control
- Spectral shaping with resonance capabilities
- Amplitude and oscillator parameter modulation envelopes
- Low-frequency oscillator modulation of amplitude and oscillator parameters
- Parameter adjustment via knobs, buttons, and display
- Mono line-out
- No perceptible latency
- Powered by “wall-wart”
- Small/portable
- Durable

Performance Specifications

- audio latency < 10ms
- sample rate > 48000 kHz
- dynamic range > 90 dB
- 16-bit or greater audio resolution

Section 4 - Design Approach Alternatives

Oscillator

I have chosen one common algorithm (differentiated parabolic waveform generation) [3] as well as two novel algorithms proposed by Lazzarini and Timoney (tanh waveshaping and Modified FM synthesis) [1] to research. For each, square, sawtooth, and triangle waves were implemented in MATLAB. Finally, a phase-synchronous version of Modified FM synthesis is explored, also based on work by Lazzarini and Timoney. [2]

Differentiated Parabolic Waveform (DPW) Sawtooth

DPW generation begins with a modulo counter which goes from 0 to 1 in delta = f_o/f_s increments, giving a sawtooth with a frequency of f_o. The result is made bipolar by multiplying it by 2 and subtracting 1. The bipolar sawtooth is squared to obtain a parabolic waveform, then differentiated using $H_D(z) = 1 - z^{-1}$ to obtain an alias-suppressed sawtooth. The output is scaled by $c = f_s/[4f_o(1-f_o/f_s)]$ for normalization.

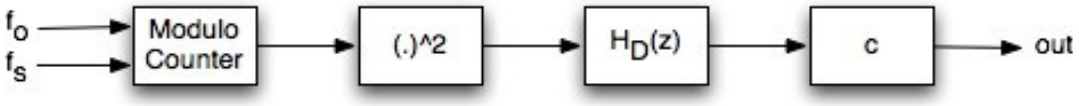

Figure 4.1: DPW Sawtooth Generation

Square from Sawtooth

To obtain a square wave from a sawtooth wave, it is necessary to subtract a 50% phase-shifted sawtooth from the original sawtooth. If an adjustable duty cycle is desirable, the phase shift can be varied.

Tanh Waveshaping Square

Tanh waveshaping uses the concept of distortion synthesis to generate a square wave. The tanh() function, a sigmoid, is overdriven by a sine wave, and a square wave is approached as the sinusoid encounters soft, nonlinear clipping. The input signal is a pi/2 scaled sine wave ($x = pi/2 * \sin(2*pi*f_o*t)$). The general form of the output signal is $y = \tanh(k*x)$, where it is determined that $k = 12000/(f_o \log_{10} f_0)$. The result is a square wave that is, in practice, band-limited. The brilliance in this technique is that k can be made a user-adjustable parameter, and the wave shape can be smoothly varied between sinusoidal and square-shaped (ie the user can adjust the rate of spectral rolloff).

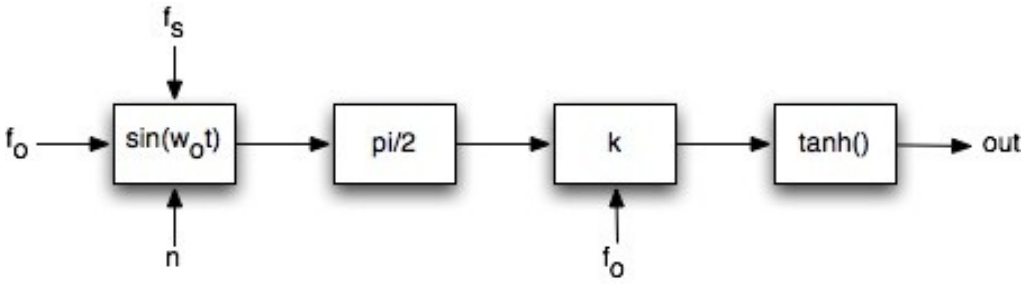

Figure 4.2: tanh Square Generation

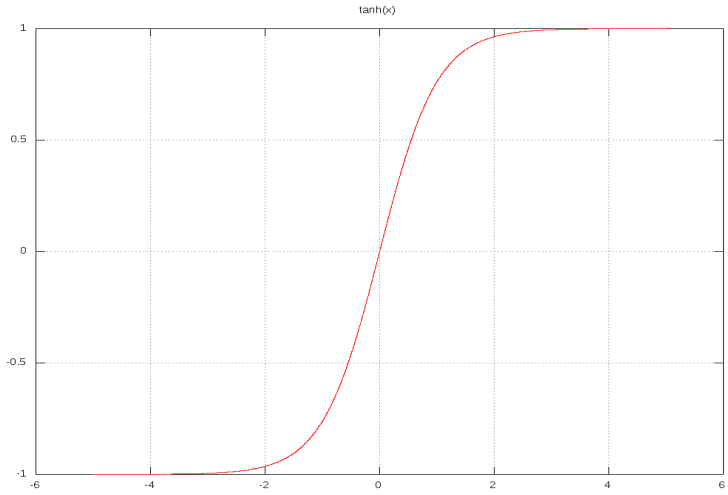

Figure 4.3: tanh() function

Sawtooth from Square

A sawtooth has a combination of odd and even harmonics, while a square wave only contains odd harmonics. We must find a way of generating the even harmonics. We can generate the even harmonics by ring modulating the square wave with a cosine wave at the same frequency: s(t) = square(wt)*cos(wt). Adding the odd harmonics back in: saw(t) = square(wt)*[cos(wt) + 1].

Modified FM Square

In ordinary FM synthesis, frequency is modulated according to the following form: $s(t) = \cos(w_c t + k*\cos(w_m t))$, and is based on ordinary Bessel functions. However, Modified FM synthesis is based on modified Bessel functions, and has the form:

$$s(t) = e^{k*\cos(w_m t)}\cos(w_c t).$$

Scaling the output by e^{-k} and parameterizing the carrier to modulating frequency ratio we have:

$$s(t) = e^{(k*\cos(m*w_o t)-k)}\cos(c*w_o t)\ ,$$

with standard FM parameters *c, m,* and *k*.

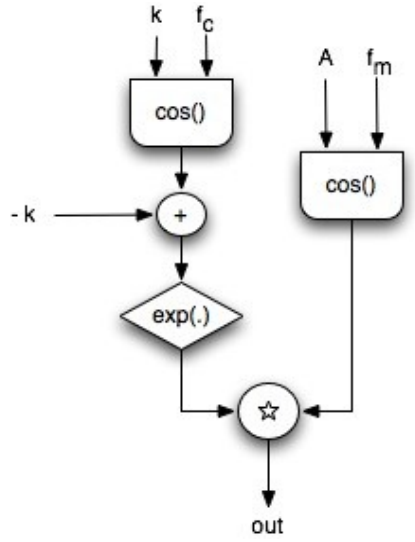

Figure 4.4: Basic ModFM flowchart

To generate a square wave, we must choose a c:m ratio of 1:2 to get a bandlimited bipolar pulse wave. A frequency dependent function for the proper k (modulation index) value must be determined experimentally. Integrating this using a one-pole filter of the form $H(z) = 1/(1-z^{-1})$, we get a nearly bandlimited square wave.

Triangle from Square

To obtain a triangle wave from a square wave, it is necessary to integrate it using a one-pole filter. Because of imperfections in the symmetry of the input square wave, the DC component drifts over time. For this reason a DC block filter is necessary. I have chosen the linear-phase DC blocker proposed by Yates and Lyons (2008). This consists of 4 D-point moving average filters, with the transfer function:

$$H(z) = (1/D)*(1-z^{-D})/(1-z^{-1})$$

whose output is subtracted from the input delayed by the group delay (2D-2). If D is an integer power of two, the division can be carried out using a right shift, and since all coefficients (with 1/D already accounted for) are one, no multiplies are necessary.

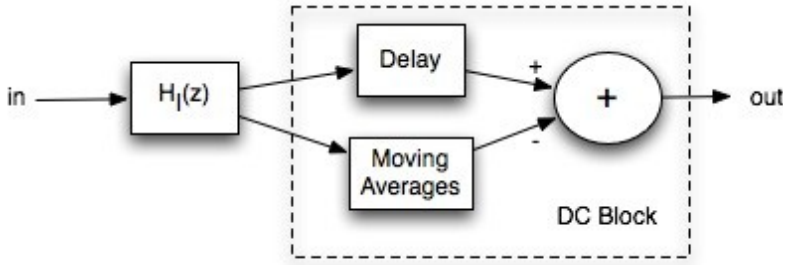

Figure 4.5: Square to Triangle Conversion

Phase-synchronous ModFM

A phase synchronous version of the ModFM algorithm exists. This algorithm allows only strictly integral c:m ratios. The c:m ratio is thus the integer parameter *n*. This allows for the use of a single phase counter. This is very useful for fixed-point implementation of an FM algorithm because small inaccuracies in phase increments can accumulate when using multiple phase counters, resulting in a inconsistent waveform (and tone).

There is still the capability of implementing the fractional component of the c:m ratio. This is accomplished by using two carriers, one with frequency $n*f_o$ and the other tuned to the next adjacent harmonic $(n+1)*f_o$. A cross-fade factor *a* is used to blend some of the second carrier signal with the first, which, viewed from the standpoint of resonant/formant synthesis, will place the center of resonance between the two carrier frequencies.

Inharmonicity, accomplished through the use of an irreducible c:m ratio, can be implemented by means of frequency shift. This requires a second phase counter, which is not synchronous with the main phase counter. A flowchart describing the implementation of phase-synchronous ModFM is listed below.

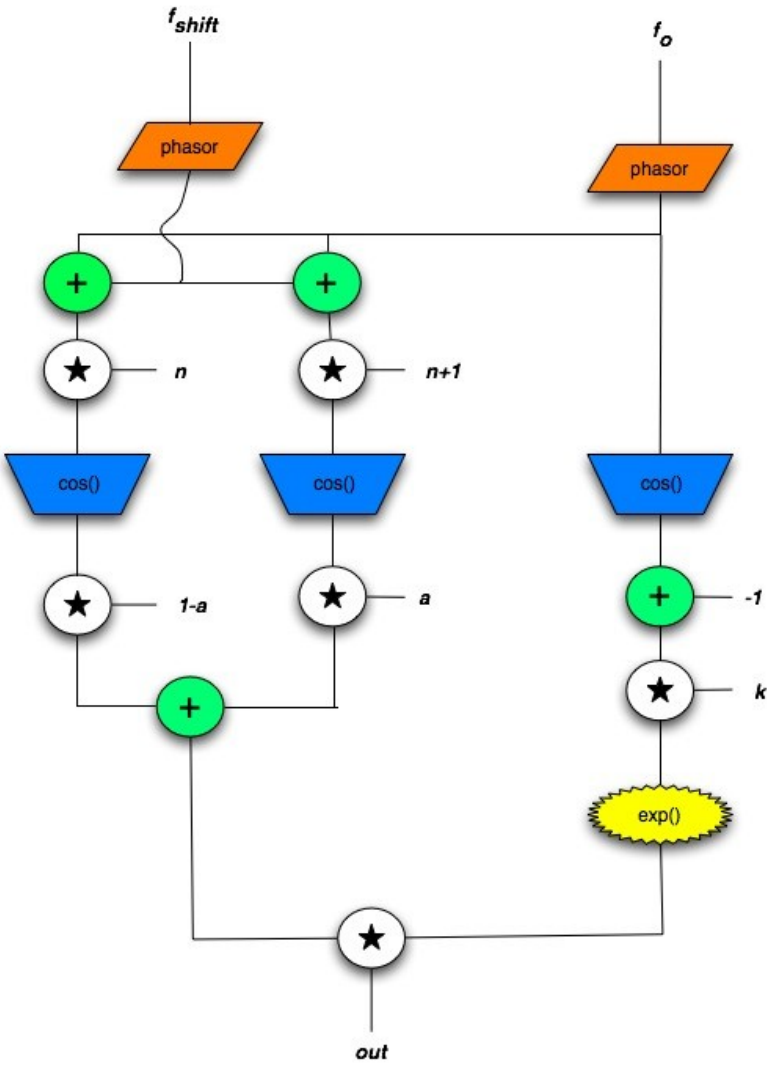

Figure 4.6: Phase-synchronous ModFM

Assessment of Oscillator Alternatives

The DPW sawtooth algorithm is the most computationally efficient. It requires no oscillators, just a counter. It also only requires one state variable. Conversion to square wave is straightforward as well, only requiring a delay unit and subtraction. Finally, a triangle wave can be generated using an up/down modulo counter. However, this algorithm has the greatest aliasing penalty, as shown in Figure 8.

The tanh waveshaping algorithm is fairly efficient as well. It requires an oscillator and a lookup table, as well as a function for calculating the modulation index, k. It is probably the best choice for generating square, sawtooth, and triangle waveforms, as aliasing seems to be practically non-existent. An added bonus of this technique is the presence of waveshaping parameters. The input sinusoid can be scaled to allow for smooth transitions between a sinusoid and square wave. The square-to-sawtooth conversion also allows for a parameter for smooth transitioning between a square and sawtooth wave. This, however, requires an amplitude normalization function.

The ModFM algorithm is the best choice if it is desirable to have the option of complex waveform generation. The presence of non-desirable harmonics in square, sawtooth, and triangle waveforms is fairly minimal. Also, the modulation index and c:m ratio can be modified to create sounds such as bells, brass, and electric piano.

The DC blocking filter used to generate triangle waveforms adds a fair amount of complexity (4x 64 point MA filters), as well as ~4 ms latency. This latency is acceptable as long as hardware latency from the other algorithms is minimal.

Normalization of triangle waves was also necessary, as amplitude drops while frequency increases. These functions were found empirically to be decaying frequency-dependant power functions.
The phase-synchronous implementation of ModFM requires only a single phase counter for the modulator and carrier signals. This is useful in achieving consistent waveforms, especially in a fixed-point implementation. Also, the additional parameters allow for extensive spectral shaping without a filter, using carrier-to-modulator ratio *n* and cross-fade factor *a* to precisely locate the center of resonance, while allowing for bandwidth adjustment using index of modulation *k*.

Oscillator Selection

The phase-synchronous ModFM algorithm will be implemented for the main oscillator. This algorithm allows for the greatest flexibility in timbre with a single instance of the algorithm, and spectral shaping can be performed on the fly without a filter. The resulting common waveforms are nearly band-limited as well (as long as index of modulation *k* is constrained properly.

For the LFO waveforms, arbitrary waveforms will be used. Aliasing/undesirable harmonics aren't much of an issue since these waveforms are outside of the audio frequency range.

Section 5 - Product Design

User Interface

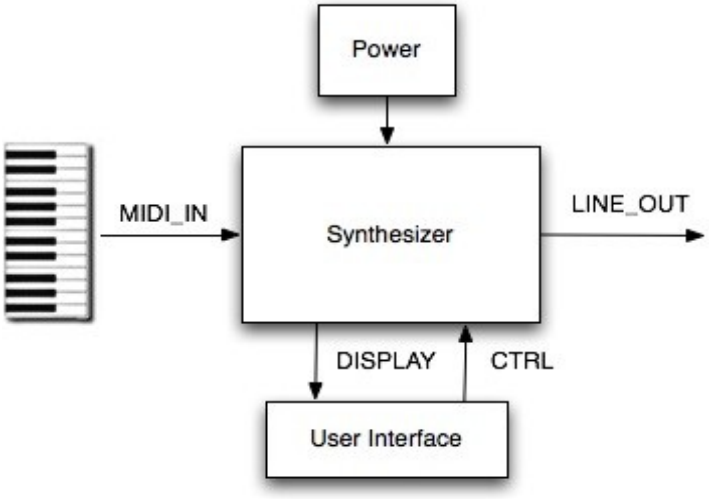

Figure 5.1: Black Box Diagram

User-controllable parameters:

Oscillator

- Pitch: MIDI note 21 to 107
- ModFM parameters:
  - cross-fade factor a: 0 to 1
  - carrier to modulator ratio n: 0 to 7, strictly integral
  - index of modulation k: 0 to 63
  - shift frequency f_{shift}: 0% to 100% of fundamental frequency

Mixer/Amp

- Volume: 0% to 100%

LFO Modulation (amplitude, *k*, *a*, *f shift*)

- Amount: 0% to 100%
- Speed: 0 Hz – 10 Hz

Envelope (amplitude, *k)*

- Attack: 0.015 s – 4.25 s
- Release: 0.015 s – 4.25 s

Feedback/Display:

- Parameter names and values are displayed back to the user when parameters are adjusted

Physical Interface:

- "Parameter matrix" consisting of 4 knobs aligned horizontally, push button to cycle "vertically" through parameter menus, and parameters labeled inside the matrix
- LCD alphanumeric display to display parameter name and value

Figure 5.2: Photograph of Physical Interface

Knob 0	**Knob 1**	**Knob 2**	**Knob 3**	
k	*n*	a	*f shift*	**Oscillator**
LFO amount	LFO frequency	Attack	Decay	**Amp Modulation**
LFO amount	LFO frequency	Attack	Decay	**k Modulation**
LFO amount [*a*]	LFO frequency [*a*]	LFO amount [f *shift*]	LFO frequency [*f shift*]	**a/shift Modulation**

Figure 5.3: Layout of Parameter Matrix

Theory of Operation

Hardware Subsystems

The hardware side of this system is built around the XMOS XC-1A development board, which includes an XS1-G4 microcontroller with four cores, hardware multithreading and thread scheduling, ample timer and clock resources, and link channel ends and switches to allow for intra- and inter-core communication between threads. The master clock is fixed to 400 MHz and I/O is clocked at 100 MHz. [4]

A single channel, 16x oversampling DAC is used to output line level sound. Parameter values are read into the system using four potentiometers and a four-channel ADC. An optocoupler-based circuit is used to convert line current from the MIDI cable to serial digital values to be input using a UART-like

format. An UART-enabled LCD display coupled with a pushbutton is used to display parameter values and navigate display/parameter selection menus. Finally, the power supply provides a reference voltage and +/- 5 V rails to supply the microcontroller, digital circuitry, and DAC.

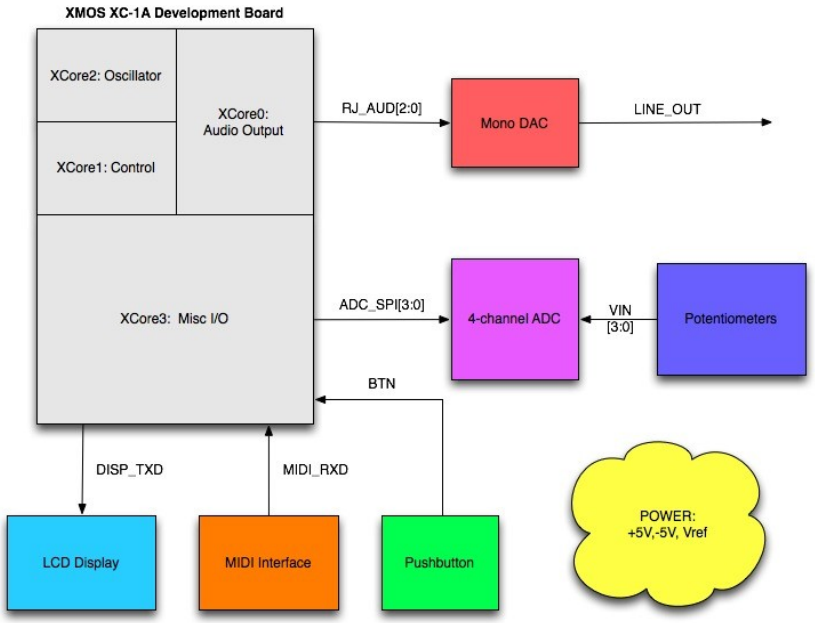

Figure 5.4: Overall System Block Diagram

Audio DAC

The DAC chosen for this system is the 16-bit DAC8580 manufactured by TI. This DAC was chosen because it provides up to 16x oversampling, doesn't use noise shaping, and doesn't require an external buffer to drive line loads. These features reduce the need for the microcontroller to run at high sample rates in order to push image frequencies farther up the frequency band. Also, external circuitry requirements are minimized (no buffer or LPF required). The reference voltage input keeps the output voltage swing between +/- VREF.

All supplies are bypassed to ground for clean operation. AGND and DGND pins are connected to analog ground. The device is configured for maximum oversampling (16x) with digital filter on and connected to DAC latch. Right-justified serial audio data is supplied to SCLK, FSYNC (frame sync), and DIN pins.

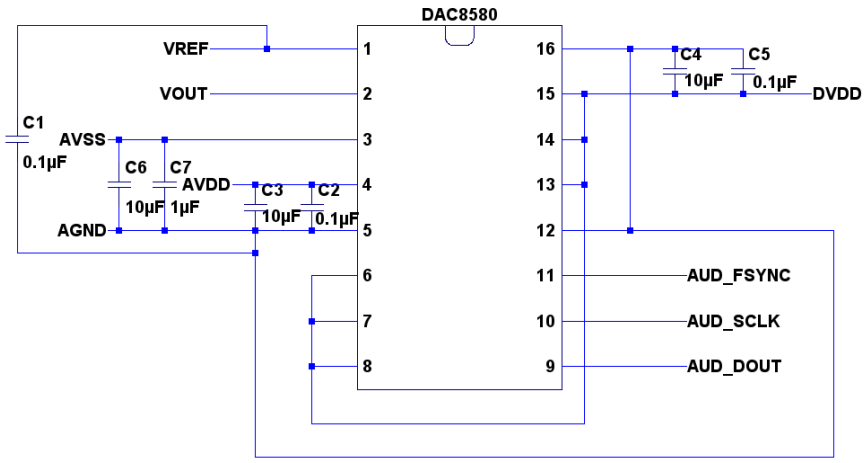

Figure 5.5: Audio DAC Connection Diagram

Four-channel ADC

This system uses the TI ADS7841, a four-channel, 12-bit ADC with SPI interface. The four voltage inputs are read using software-controlled multiplexing. The SPI MOSI byte from the microcontroller tells the ADC which channel to read, and instructs it to take a single-ended reading. The device responds with a two byte value containing the 12-bit ADC reading. The device is connected to four potentiometers in voltage divider configuration. The supply and reference voltages are bypassed to ground for clean operation.

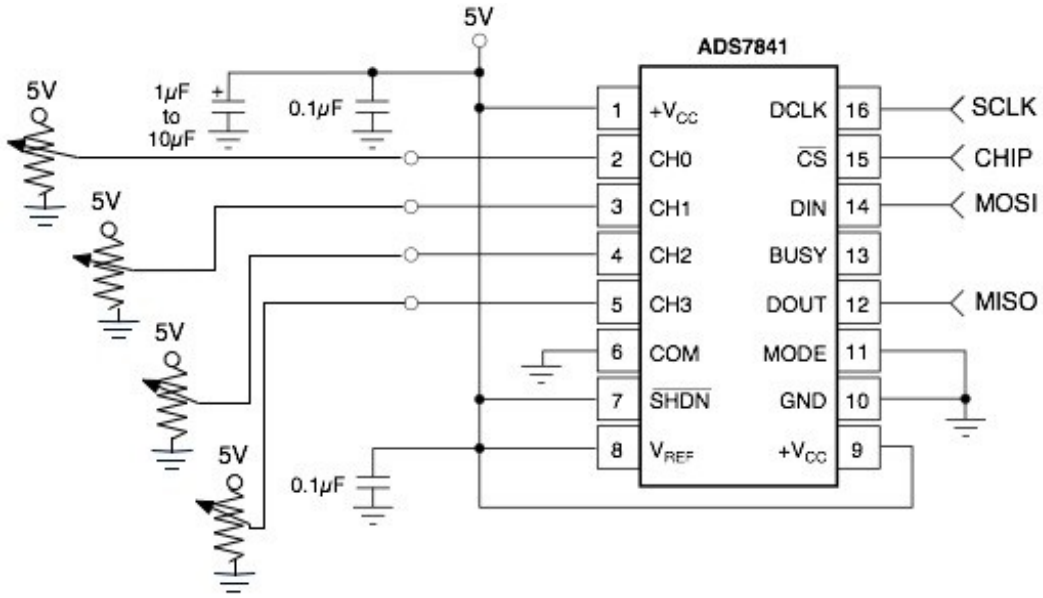

Figure 5.6: ADC Connection Diagram

MIDI

MIDI data is transmitted between controllers and devices using a two wire current loop sent through a DIN-5 cable. The hardware interface on the device side consists of a female DIN jack, diode, and pull

up resistors to convert the current loop to digital voltage. Logic 1 is signified by no current flow (optoisolator LED off, high voltage at output due to optoisolator logic conversion). Logic 0, conversely, is signified by current loop flow. The MIDI input hardware is built according to MIDI specification standards.

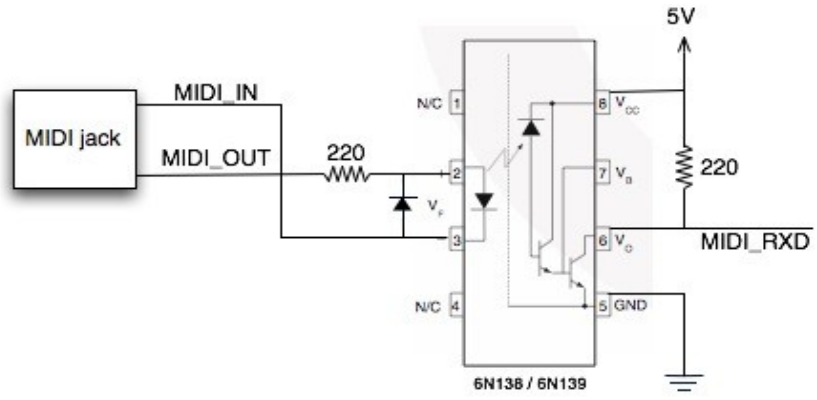

Figure 5.7: MIDI Interface Schematic

Power Supply

The power supply is designed to take an unregulated DC power source and provide regulated +/- 5V rails and a reference voltage for the DAC. The input voltage is first regulated down to 9V to provide proper input voltage levels for the REF02 (5V reference) and MAX635 (inverting switching regulator). The +5V supply is generated using a simple 5V linear regulator (LM7805). Because a single DC supply is used, the -5V supply is generated using a -5V inverting switching regulator (wired to generate factory preset -5V voltage). Finally, the TI REF02 is wired to trim down the reference voltage as much as possible (5V is too high for many audio inputs, trimming reduces the reference voltage to about 4.2V).

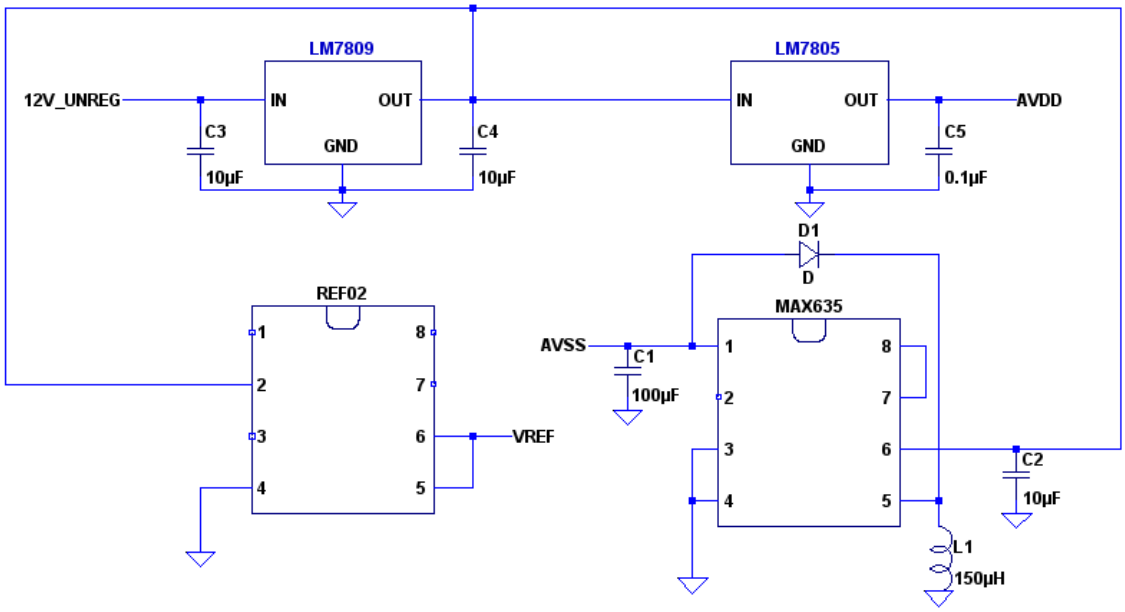

Figure 5.8: Power Supply Schematic

Software Subsystems

The general software flow of this system begins with events generated by hardware abstractions of input peripherals. These input events are handled in the parameter control subsystems which are also passed applicable data values. Control data is streamed to the oscillator, which streams audio data to the right-justified audio output hardware abstraction. Parameter and menu state data from the control subsystems is sent to the LCD display hardware abstraction, which handles state change and parameter update events. The software flow will be elaborated on with the description of each individual subsystem. A high-level software flow diagram is presented below.

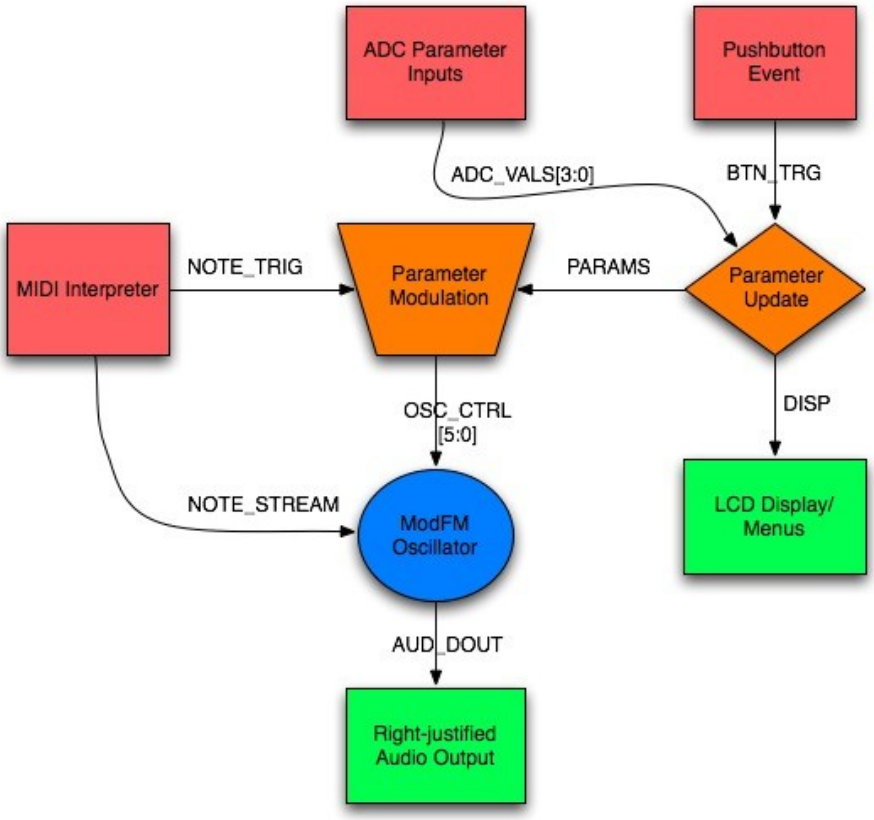

Figure 5.9: Software Subsystem Block Diagram

Input Hardware Abstractions

1. **ADC Parameter Inputs:**

   This subsystem implements SPI in software to control the ADC and retrieve ADC values. Hardware value buffering, serialization, deserialization, and I/O clocking features are interfaced with through code. The SPI is clocked at 500 kHz, giving a data rate of 500 kHz/(8 bits/byte) = 62.5 kB/s. A one-byte command is output to the MOSI pin giving the channel to be read, and configuring the device to perform a single-ended voltage conversion. This is followed by reading in two bytes of conversion data from the MISO pin. This is repeated for all four channels, and once values are retrieved, they are sent over a data channel to the parameter update thread. ADC read rates are controlled by a timer and set to 10 values per second per channel.

SPI functionality is implemented using an unaltered SPI module available from XMOS.

2. **MIDI Interpreter:**

   This subsystem begins with a UART like interface that is triggered by a start bit on the UART RX pin. Each byte is read in at 31250 baud and sent over a data channel to a FIFO buffer operating on a separate thread. Bytes from the buffer are sent out to a MIDI parsing function on another thread. MIDI data is sent in the format NOTE ON/OFF → KEY → VELOCITY. The MIDI parsing function handles NOTE ON/OFF events by reading the next two values in the buffer (KEY and VELOCITY). For NOTE ON, a variable holding the note value is given the KEY byte, and for NOTE OFF, the note variable gets -KEY (to distinguish between note on and note off for a given note). The note variable is passed to the oscillator and a note on or note off trigger is passed to the control subsystem on each MIDI event.

3. **Pushbutton Event:**

   This subsystem waits for a 0 to 1 transition on the pushbutton and debounces the button with a hardware timer (ignores events for 1 ms after the transition). If this transition is detected, the menu state is incremented. Whenever incremented, this state is sent to the parameter update subsystem.

Control Subsystems

1. **Parameter Update**

   This subsystem handles pushbutton and ADC value read events. Also, a two dimensional user parameter matrix is initialized and updated (of the form o_params[state#][value#]). When initially entering a new menu state, the values previously stored in that state are left unaltered. A parameter is only updated with the current ADC value once the knob is turned to the position of its previously stored value. This functionality is implemented using a matrix of previous value flags. All flags in the matrix are set high on a state change to signal that parameters in the new state should keep their old values. Conditional statements detect whether incoming ADC values fall within a short range of previously stored parameter values. When this condition is detected, the flag corresponding to the index of the parameter the condition was detected on is set low to signal that the parameter should be updated with current ADC values. Parameter values are then sent to the parameter control and display subsystems. For more information on the layout of the parameter matrix, refer to Figure 5.3 (menu names listed in order of increasing state).

2. **Parameter Control**

   This system handles pushbutton, parameter update, and note trigger events. Parameter modulation is performed here. All oscillator parameters can be modulated by a LFO with a cosine wave-shape and adjustable frequency and amplitude. Amplitude and *k* values can be modulated using an attack-release envelope. An attack is triggered by reception of a high note trigger. A decay is triggered by reception of a low note trigger. A matrix of the same form described in Parameter Update is implemented in this subsystem and updated by values sent by the parameter update subsystem (stored into the current state index). These values are used for setting and modulating each of the oscillator parameters. Oscillator parameter control signals (amplitude, *k, n, a, f shift)* are updated at audio rate, allowing for transmission over a *streaming channel* to the oscillator subsystem (for fastest possible thread communication). All calculations use 8.24 fixed-point math.

ModFM Oscillator

The oscillator subsystem handles note events and streams in parameter control signals. Upon a note event, a new MIDI key number is read in and its corresponding phase increment value is read from a lookup table. New parameter values are read in at the beginning of each iteration of the oscillator loop. These values are input into the phase-synchronous ModFM algorithm, as depicted by the flowchart in Figure 4.6. The equation used to implement this algorithm is shown below:

$$y = A * e^{(k*\cos(phase)-k)} * [(1-a)\cos(n(phase + phase_{shift})) + a*\cos((n+1)(phase + phase_{shift}))]$$

This oscillator is always running, and notes are turned on and off by the amplitude attack-release envelopes. Once each sample value is calculated, a bit reversing operation is performed to prepare the sample for right-justified audio output. Phases are wrapped around (0 to 2*PI) before each sample calculation, and incremented after each sample calculation. All calculations are performed using 8.24 fixed phase math, and samples are downshifted by 10 bits before output to transform the value into a 16-bit signed integer.

Output Hardware Abstractions

1. **Right-justified Audio Output**

   The DAC8580 device requires right-justified audio output. This is similar to SPI, except a frame synchronization signal is sent to signal the end of a data word and update the DAC latch. Data is clocked into the DAC shift register on each rising edge of SCLK. The frame sync signal is raised high sometime in the middle of a serial sample output, and dropped low on the first falling edge after the final data bit is clocked into the DAC. The DAC8580 includes an oversampling digital filter, which requires a continuous SCLK. These constraints require fairly strict timing requirements. SCLK must operate at a frequency exactly 16 times the sample rate. This subsystem is the only thread in the streaming path starting at the parameter control subsystem, moving through the oscillator subsystem, and terminating at right-justified audio output with a data rate controlled by a timer or clock. Thus, in effect, the clock source configured for this subsystem sets the sampling rate for the entire system. The 100 MHz clock is divided by a factor of 100, giving an SCLK rate of 1 MHz. This sets the sample rate at 1 MHz/(16 bits/sample) = 62500 samples/second.

   The SPI module available from XMOS was modified extensively to satisfy these timing requirements.

2. **LCD Display**

   This subsystem takes in state and parameter values from the parameter update subsystem. Based on the state value, a menu title string is selected. A command is sent to set the cursor to home position, and print commands are used to send the menu title and parameter values over UART. XLOG, a module available from XMOS that redirects all print statement outputs to a UART server, was installed for its string formatting capabilities and to avoid synchronization issues.

Software/Hardware Interfacing

Port assignments are defined using an ".xn" file, allowing for easy cross-platform porting. An ".xn" file is provided for the XMOS XC-1A development board. Below is a listing of the port and pin assignments of each input and output signal.

Signal	**Direction**	**Processor**	**Pin**
BTN	Input	X1PortA	X1D0
MIDI_RXD	Input	X3PortB	X3D12
DISP_TXD	Output	X0PortA	X0D39
RJ_AUD[2:0]			
AUD_SCLK	Output	X0PortA	X0D36
AUD_FSYNC	Output	X0PortA	X0D37
AUD_DOUT	Output	X0PortA	X0D38
ADC_SPI[3:0]			
MOSI	Input	X3PortA	X3D0
MISO	Output	X3PortA	X3D1
CHIP	Output	X3PortA	X3D10
SCLK	Output	X3PortA	X3D11

Figure 5.10: I/O Port Map

Section 6 – Physical Construction and Integration

Circuit Board Implementation

- Breadboard style perf-board with point-to-point soldered connections (saves time and money for prototype vs. custom PCB, component count is low)
- Components distributed among three boards:
  - Power: provides +/- 5 V supplies as well as a reference source for the DAC
  - Microcontroller: the XMOS XC-1A development board breaks out all available I/O and includes a prototyping area which will be used to house the DAC
  - Peripherals: a separate board will be used to house the MIDI hardware and ADC chip

Enclosure Concept

- Packaged in ABS plastic enclosure with parameter controls on top and connections on back panel, no cooling required

Power Source

- 9V unregulated wall-mount transformer coupled with internal voltage regulation

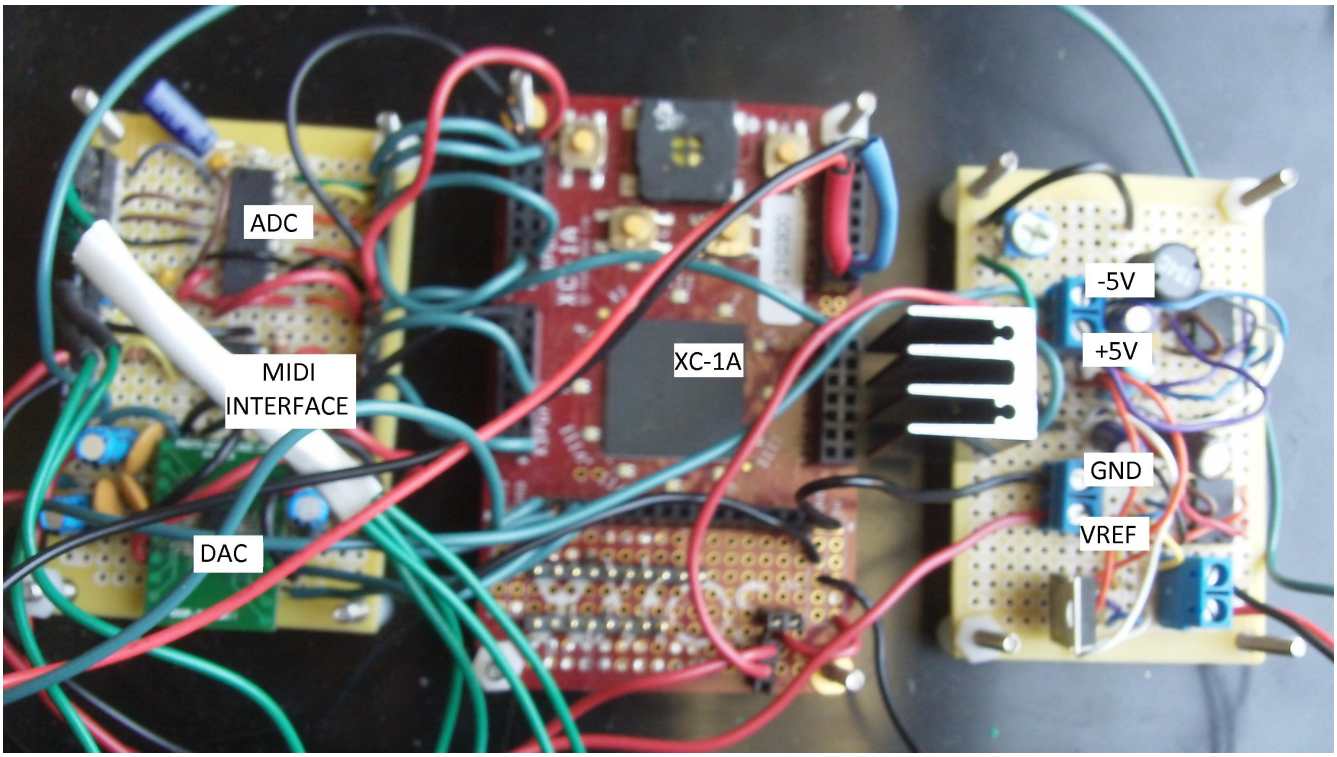

Figure 6.1: Photograph of Board Implementation

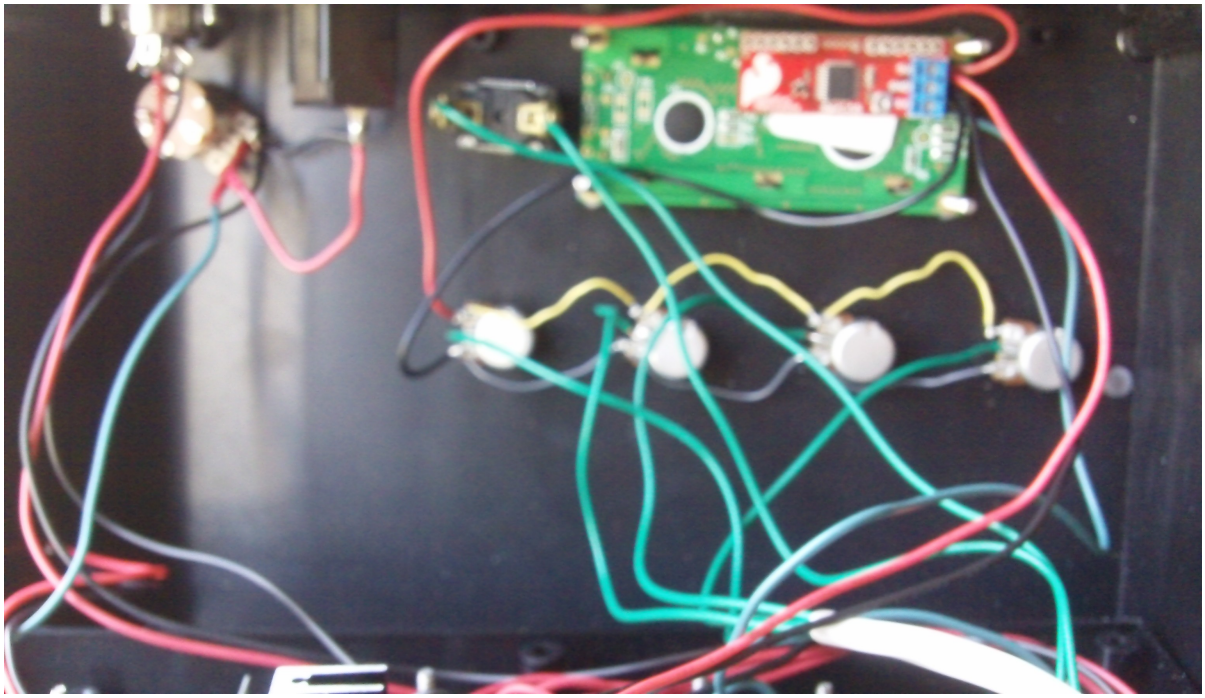

Figure 6.2: Photograph of Panel Wiring

Section 7 – Integrated System Tests and Results

Project (System-Level) Verification Testing

The project was tested by connecting a MIDI keyboard to the input of the synthesizer and a 12V DC power adapter to the power plug. The wave-shape was set to a cosine and each MIDI key was pressed. Each note was verified to be "in tune" (the correct frequency), by comparing the output to a waveform generated by a software synthesizer.

Variations in parameters and parameter modulation settings were tested exhaustively. All variations in parameter settings produced predictable results. However, when a sequence of keys are pressed consecutively, or long amplitude attack or decay settings are used, intermittent popping is present. This is likely due to discontinuities created in the amplitude envelope when the amplitude is reset to zero upon reception of a new NOTE_ON signal.

The system handles rapid succession of MIDI note signals with no issues and no noticeable latency.

Subsystem Verification Testing

Power Supply:

The power supply was first tested with a zero DAC output and no load on the DAC. Then maximum positive and negative digital values were sent to the DAC and tests were conducted with no load, and a low impedance load. For each test, the DAC output, reference, AVDD, and AVSS voltages were recorded using a multimeter. Low impedance DAC loads had no effect on supply voltages. The low impedance load loaded the DAC, but in a predictable way (the low impedance load tested is comparable to the output impedance of the DAC). These tests were conducted due to concerns over the maximum output current of the inverting regulator used to generate the negative rail.

	Maximum Positive Output (no load)	Maximum Positive Output (218 ohms)	Zero Output (no load)	Maximum Negative Output (no load)	Maximum Negative Output (218 ohms)
VOUT	4.28	3.95	0	-4.27	-3.95
VREF	4.21	4.21	4.2	4.21	4.2
AVDD	4.94	4.93	4.94	4.95	4.94
AVSS	-4.93	-4.93	-4.93	-4.94	-4.94

Figure 7.1: Power Supply/DAC Load Testing

Next, the output noise of each supply rail was tested with an oscilloscope. The rails appear to contain a significant amount of noise. Most of the noise on the positive supply lies within a **30 mV** range, but momentary, significantly larger voltage spikes are present. The negative voltage contains noise that mostly lies in a **52 mV** range, but also contains significant voltage spikes. Scope readings are shown below.

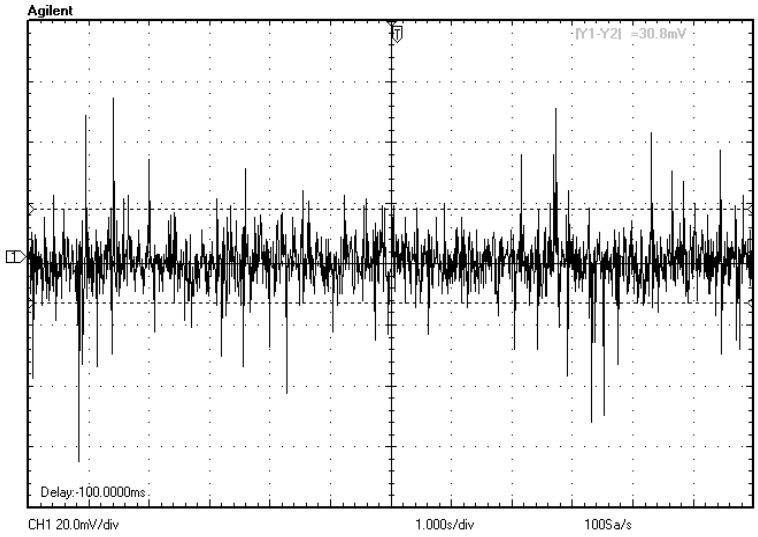

Figure 7.2: AVDD Supply Noise

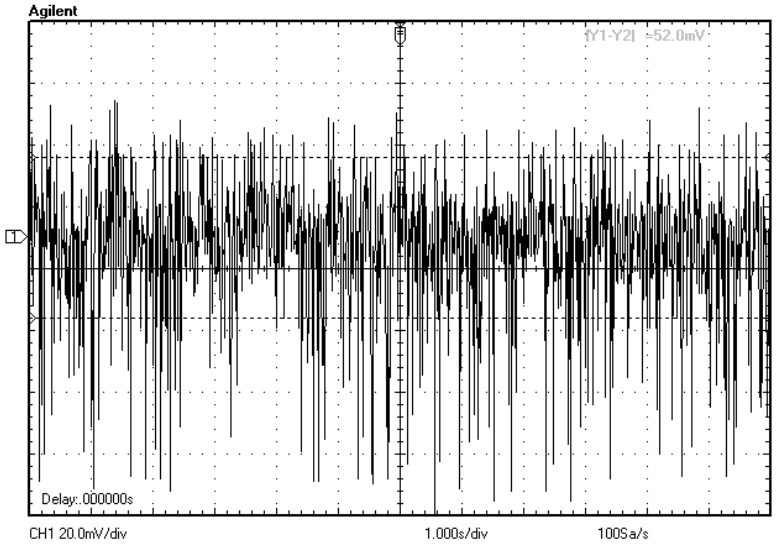

Figure 7.3: AVSS Supply Noise

Display:

The LCD display was tested by writing a "Hello, World!" string to the LCD string. Accurate and glitch-free operation was verified. The refresh rate of the display was adjusted to optimize readability.

The display started glitching for a few days after weeks of problem-free operation. The display went dim after startup and, intermittently, garbage data would be displayed on the screen followed by a blank screen. The root of this problem was not found. After a few days, correct operation resumed.

ADC:

The ADC was tested by displaying ADC values on the LCD screen. Testing verified that potentiometer

position correlated with the ADC value, and that the potentiometer could generate the full range of ADC values. However, minor fluctuations in the ADC readings were present.

DAC:

The DAC was tested by outputting a cosine wave, calculated on a once-per-sample basis, to the DAC. Correct operation was verified (the output signal was in the shape of a cosine and no overflow or clipping was present). The dynamic range (or SNR in the presence of a signal) was tested by recording a 440 Hz cosine output to a raw WAV file (resampled at 96 kHz) and generating a frequency spectrum in MATLAB. The peak of the desired harmonic was compared to the peak amplitude of the noise floor. This system exhibits a dynamic range of **60.72 dB**.

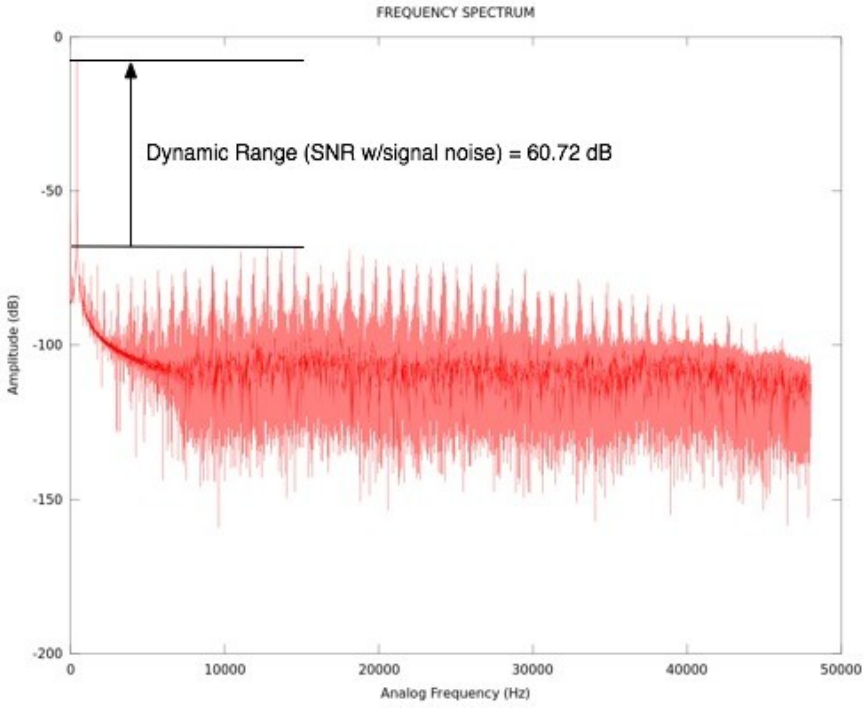

Figure 7.4: Frequency Spectrum of 440 Hz Cosine Output

Synthesis Algorithm:

The synthesis algorithm was tested at maximum *k* and *n* values, with a high fundamental frequency, to test for the presence of aliasing at parameter extremes. The output waveform was recorded into a raw WAV file (at fs = 96 kHz) and analyzed in MATLAB. Harmonics significantly above the noise floor and located above the expected cutoff of the DAC digital filter are present. Also the noise floor is significantly higher at parameter extremes. Aliasing, or other digital artifacts, are likely present, as a *k* value scaling function is not implemented, and *k* determines signal bandwidth. The dynamic range of this output waveform is **50 dB**, about a 10 dB decrease compared to the 440 Hz cosine output test.

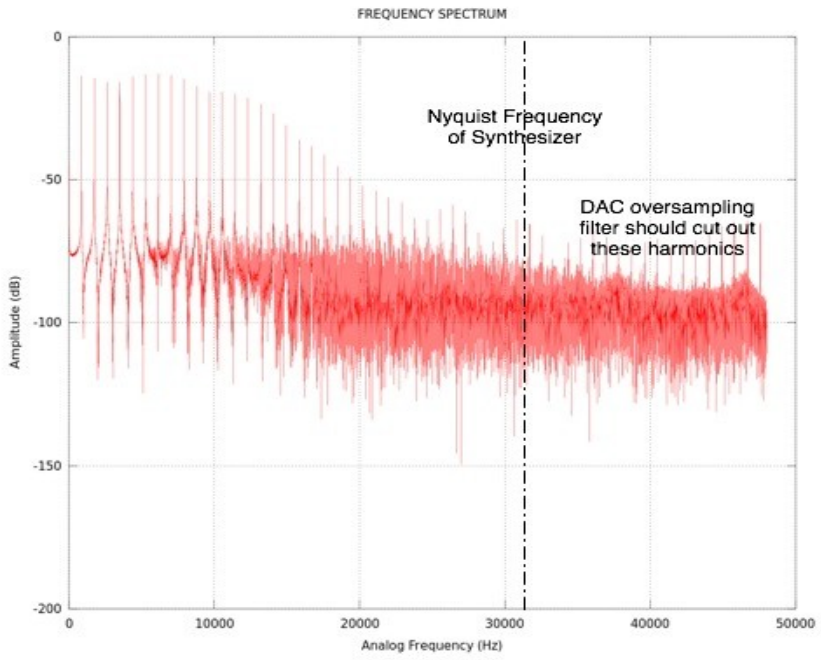

Figure 7.5: Frequency Spectrum (fo = 1760, k = 63, n = 7)

Parameter Modulation:

Envelope attack-release modulation was tested by modulating a cosine wave with amplitude attack-release envelopes. To test amplitude envelope modulation, output waveforms are recorded at minimum and maximum attack/release settings and plotted vs. time.

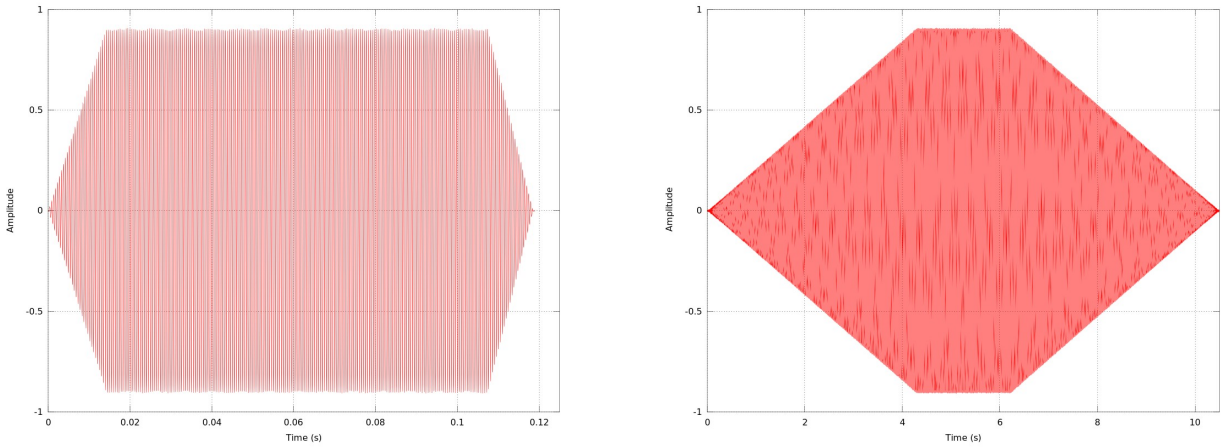

Figure 7.6: Envelope Modulation of Amplitude (short and long attack/decay)

Minimum achievable attack or decay time is **0.015 s** (above left). Maximum achievable attack or decay time is **4.25 s** (above right). The effect of a long attack envelope (4.25 s) modulating *k* on the frequency spectrum was also tested (using *n=2* @ A440). The recorded *k*-modulated waveform was rendered as a spectrogram (shown below).

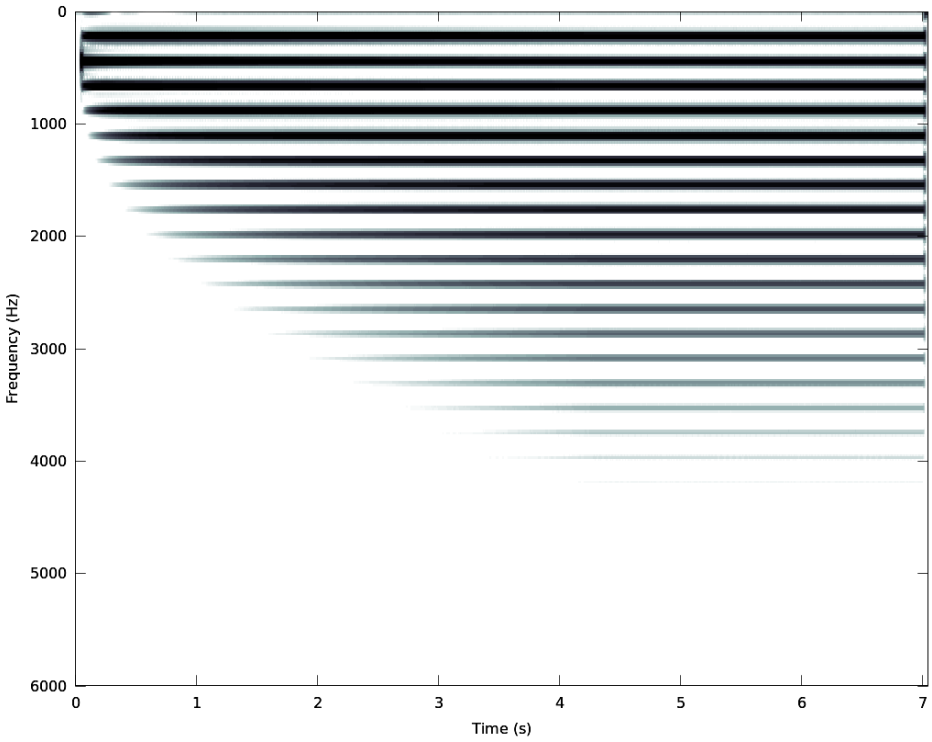

Figure 7.7: Envelope Modulation of k *(long attack, A440, n=2)*

LFO parameter modulation was tested by modulating a cosine wave with an LFO at low frequency and maximum frequency, and plotting the output waveform vs. time.

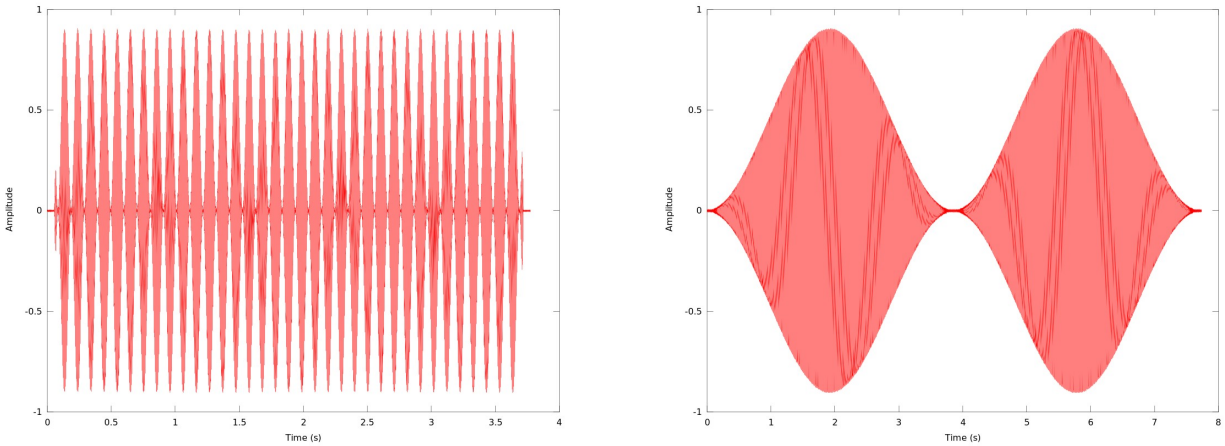

Figure 7.8: LFO Modulation of Amplitude (short and long period)

The maximum achievable LFO frequency is **10 Hz** (above left). Correct LFO waveshape can be clearly verified in the low frequency LFO test (above right).

Finally, the effect of slow LFO modulation of *k* on the frequency spectrum was tested (using *n=2* @ A440). The spectrogram is shown below.

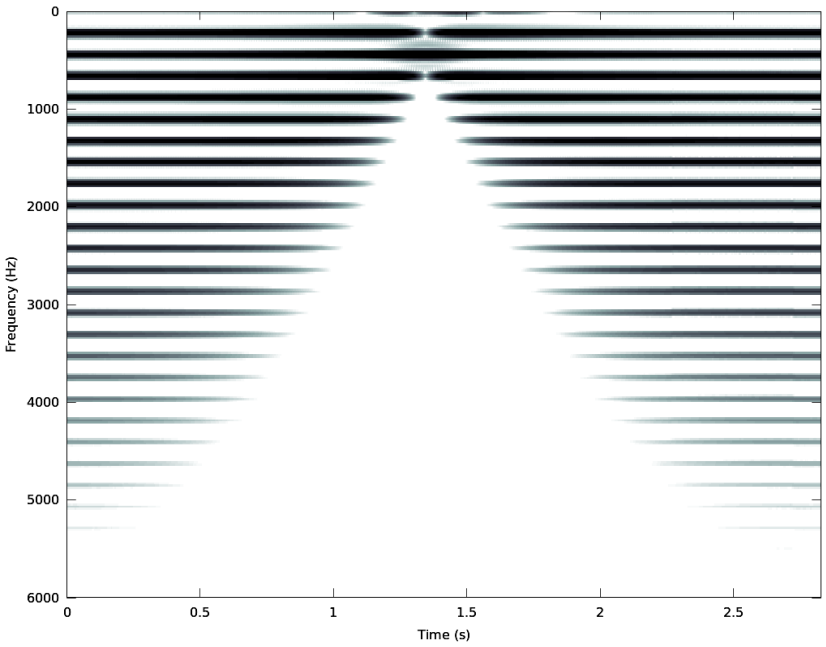

Figure 7.9: LFO Modulation of k *(long period, A440, n=2)*

MIDI:

MIDI was tested by connecting a MIDI keyboard to the synthesizer and first verifying that pressing a key starts (the correct) note, and releasing a key stops that note. This test was successful. Next, the MIDI input to the microcontroller was recorded on one channel, and the output signal was recorded on a second channel to determine the system's audio latency. Both channels were graphed using Audacity. Latency is defined as the time difference between the end of reception of the KEY data and the beginning of the attack envelope, since velocity data is not used in this application. The overall latency was found to be **1.3 ms**.

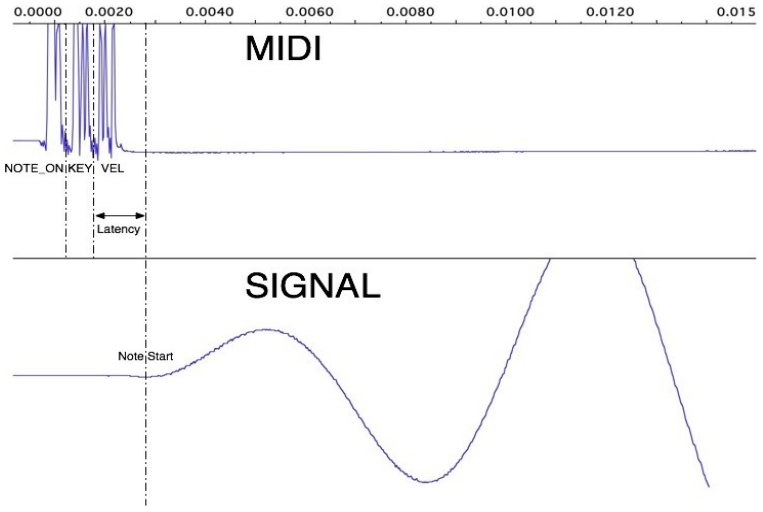

Figure 7.10: Plot for Determining Audio Latency

Summarized Test Results

Specification	Actual Value	Specification Value	% to Specification	Specification met?
Audio Latency	1.3 ms	< 10ms	769.23%	YES
Sample Rate	62.5 kHz	> 48 kHz	130.21%	YES
Dynamic Range	60.72 dB	> 90 dB	67.47%	NO
Audio Resolution	16-bit	16-bit	100.00%	YES

Figure 7.11: Performance Specification Test Results

Power Supply

Loading: The power supply is not loaded during normal operation, or during low impedance DAC loads.

Noise:

- AVDD average voltage swing: **30 mV**
- AVSS average voltage swing: **52 mV**

Display

Experienced downtime for a few days, but currently functions correctly.

Synthesis Algorithm

Parameter extremes reduce dynamic range of output signal. Aliasing likely present at parameter extremes.

Parameter Modulation

Attack-release Envelope:

- Minimum Attack/Decay time: **0.015 s**
- Maximum attack/decay time: **4.25 s**

LFO Envelope:

- Maximum LFO frequency: **10 Hz**

Section 8 – Conclusions

This project was successful in meeting most functional requirements initially proposed. The system is capable of generating spectrally dense waveforms with extensive spectral shaping capabilities and resonance control. Parameter modulation functions are available for all oscillator parameters. The user interface provides a minimal, yet intuitive menu system for the adjustment of parameter settings, with visual feedback. A line output is available, and there is no perceptible audio latency. The product is small, portable, and relatively durable.

However, the oscillator algorithm appears to exhibit aliasing, especially at parameter extremes. There is currently no method of bandwidth control implemented, so high values of k in combination with large carrier to modulator ratios or large fundamental frequencies have no guarantee of being band-limited. Even with a cosine output, undesirable harmonics above the average noise floor are present.

As far as initial performance specifications, the product meets or exceeds audio resolution, audio latency, and sample rate requirements. However, the dynamic range is significantly below specifications. Also, undesirable popping is intermittently present due to discontinuities present in the amplitude modulation envelope.

Proposed Solutions to Design Shortcomings

Noise

The noise degrading the dynamic range of this system is likely due to a combination of digital and analog noise sources. A significant amount of noise is present on the analog rails, but the dynamic range test also determined that undesirable harmonics are present in the cosine signal.

To address analog sources of noise, the following improvements are proposed:

1. **Use a center-tapped transformer to generate supply voltages:**

   Inverting switching regulators introduce a significant amount of ripple into the negative rail. Instead of inverting a positive DC supply, a wall mount center-tapped transformer coupled with full-wave rectification and filtering can be used to obtain positive and negative unregulated voltages, which can be regulated using LM7805/LM7905 linear regulators. The part count would not be significantly increased, because the switching regulator and its associated components would be eliminated.

2. **Implement an RF choke between the positive supply and digital circuitry:**

   Digital switching introduces noise into the rail it draws current from. Currently, a single positive supply for analog and digital circuitry is used in this design. Digital switching noise can be drastically reduced by introducing an RF choke into the digital supply path. This solution eliminates the need for separate digital and analog supplies while reducing the effect of digital switching on the analog supply.

3. **Redesign board layout:**

   The introduction of a ground plane could help reduce ground noise. Switching from prototyping boards to a two-sided PCB design would help reduce EMI effects and allow for one side to be used as a dedicated ground plane. Digital and analog ground planes should be separated using a small slit in the exposed copper. This would allow for a low enough

impedance to keep digital and analog grounds at the same potential while providing enough separation between the planes to keep digital ground noise from affecting the analog ground plane. At the very least, more attention should be directed towards supply and ground paths, and further experimentation should be performed

Digital sources of noise must also be addressed, as they appear to have a more significant effect on the degradation of dynamic range. The following improvements are proposed:

1. **Test other methods of cosine calculation:**

   Currently a fixed-point cosine approximation is used to calculate cosine values sample-by-sample. It is possible that this algorithm introduces approximation error, which could introduce undesirable harmonics into the cosine signal. Other methods of calculation, including table lookup and soft floating point, should be explored.

   Also, the cosine signal should be tested for phase distortion due to inconsistencies in the phasor signal. This is unlikely to be the cause of noise because the phase is wrapped around when reaching 2*PI by subtracting 2*PI from the current phase. Also the cosine output waveform doesn't appear to have any perceptible discontinuities. However, this is still worth exploring.

2. **Increase sample rate and/or implement digital filter with 20 kHz cutoff:**

   At high frequencies and parameter extremes, aliasing effects become present. Even with the currently implemented cosine signal, aliasing is likely present. The presence of aliasing can be reduced by increasing the sample rate, reducing the harmonics present above nyquist. Also, a digital filter with a 20 kHz cutoff can be implemented to attenuate harmonic content present up to (nyquist frequency + 20 kHz).

3. **Implement bandwidth control:**

   Currently, no method of bandwidth control is implemented. The index of modulation, *k*, controls bandwidth. With increasing frequency, or increasing carrier-to-modulator ratio, the maximum value of *k* necessary to avoid aliasing is reduced. Parameter *k* should be scaled according to a function of *n* and *fo* in order to ensure all generated waveforms are band-limited. This can be implemented using table lookup, or derivation and calculation.

Undesirable Popping

More work is necessary in eliminating discontinuities in attack-release envelopes. Some experimentation on implementing a fast decay from current envelope value down to zero upon a NOTE_ON trigger was performed, but has been so far unsuccessful.

Further Improvements

1. **Implement digital effects:**

Digital effects such as reverb, delay, and phaser are desirable to electronic musicians because they introduce additional sound sculpting capabilities. The XMOS XS1-G4, with its high performance specifications and hardware parallelism, should have plenty of spare resources to implement these types of algorithms.

2. **Switch to stereo output:**

Stereo output would allow for the incorporation of a sense of depth and space. Parameter modulation

could have a stereo component (different rates, phases, or amplitudes of modulation on left and right channels). Also, stereo sound is useful for expanding the functionality of digital effects, if implemented.

3. **Incorporate polyphony:**

This design is a monophonic design, meaning only one note can be played at a time. Incorporating polyphony (the ability to play multiple notes simultaneously) would allow the user to play chord parts rather than just leads and melodies.

4. **Add USB MIDI functionality:**

USB MIDI functionality eliminates the need for a dedicated MIDI keyboard or computer/MIDI audio interface. MIDI data from a DAW (digital audio workstation) would be able to be sent directly from the computer, over USB, to control the synthesizer. Many computer musicians interface with their MIDI controllers and devices using only USB. This would allow the product to fit into a broader USB MIDI solution.

Section 9 – Bibliography

1. Lazzarini, Victor, and Joseph Timoney. "New Perspectives on Distortion Synthesis for Virtual Analog Oscillators." *Computer Music Journal* 34.1 (2010): 28-40. *Project Muse*. Web.
2. Lazzarini, Victor, and Joseph Timoney. "Theory and Practice of Modified Frequency Modulation Synthesis." *J. Audio Eng. Soc.* 58.6 (2010): 459-71. Web.
3. Välimäki, Vesa, and Antti Huovilainen. "Oscillator and Filter Algorithms for Virtual Analog Synthesis." *Computer Music Journal* 30.2 (2006): 19-31. *Project Muse*. Web.
4. *XS1 Family Product Brief*. XMOS. Web. <http://www.xmos.com/published/xs1-family-product-brief>.

Appendix A: Parts List and Cost

Budget

- XMOS XC-1A: $99
- Audio DAC: $0 (samples)
- ADC: $7
- MIDI Interfacing: $5
- Power Supply: $8
- Enclosure: $14
- Display, Knobs: $40
- Connectors, misc. : $15
- Prototyping Supplies: $15

Estimated Cost: $203

Parts List

Description	Part Number	Source	Quantity	Price	Extended Price
XMOS XC-1A Development Board	XCARD XC-1A-ND	Digikey	1	99.000	99.00
Radio Shack Breadboard PCB	276-150	Radio Shack	2	2.190	4.38
10k Rotary Potentiometer	COM-09939	SparkFun	4	0.950	3.80
10k Audio Taper Potentiometer	COM-09940	SparkFun	1	0.950	0.95
Pushbutton	N/A	spare part	1	0.000	0.00
LCD Display	LCD-09052	SparkFun	1	14.950	14.95
Serial-enabled LCD backpack	LCD-00258	SparkFun	1	16.950	16.95
1/4” audio jack	SC1316-ND	Digikey	1	2.520	2.52
Female MIDI jack	CP-1250-ND	Digikey	1	1.630	1.63
ABS Plastic Project Enclosure	SR194B-ND	Digikey	1	13.990	13.99
Nylon nuts	PRT-10231	SparkFun	16	0.250	4.00
Nylon standoffs (x4)	PRT-09155	SparkFun	3	1.950	5.85
16-bit 16x oversampling serial DAC	DAC8580	TI Sample	1	0.000	0.00
10 uF capacitor	P5134-ND	Digikey	6	0.200	1.20
0.1 uF capacitor	478-3150-1-ND	Digikey	8	0.162	1.30
100 uF capacitor	1013PHCT-ND	Digikey	1	0.380	0.38
150 uF inductor	445-3751-1-ND	Digikey	1	1.280	1.28
PCB terminal connectors	PRT-08432	SparkFun	3	0.950	2.85
Machine screws	PRT-08988	SparkFun	16	0.050	0.80
Potentiometer knobs	COM-10001	SparkFun	5	1.500	7.50
PCB mount trimpot	271-282	Radio Shack	1	1.690	1.69
Schottky diodes	11DQ04-ND	Digikey	2	0.480	0.96
5V reference	REF02	TI Sample	1	0.000	0.00
5V linear regulator	LM7805CT-ND	LM7805CT	1	0.650	0.65
4-channel 12-bit SPI dac	ADS7841P-ND	Digikey	1	6.300	6.30
Optocoupler	6N138QT-ND	Digikey	1	1.340	1.34
Adjustable Negative Regulator	LM337TFS-ND	Digikey	1	0.710	0.71
QSOP-20 to DIP adapter	PA0029	Proto-Advantage	1	7.990	7.99
220 ohm resistor	P220CACT-ND	Digikey	3	0.150	0.45
100 kohm resistor	SFR16S0001003FR500	Digikey	1	0.230	0.23
			TOTAL:		203.65

Appendix B: Schedule

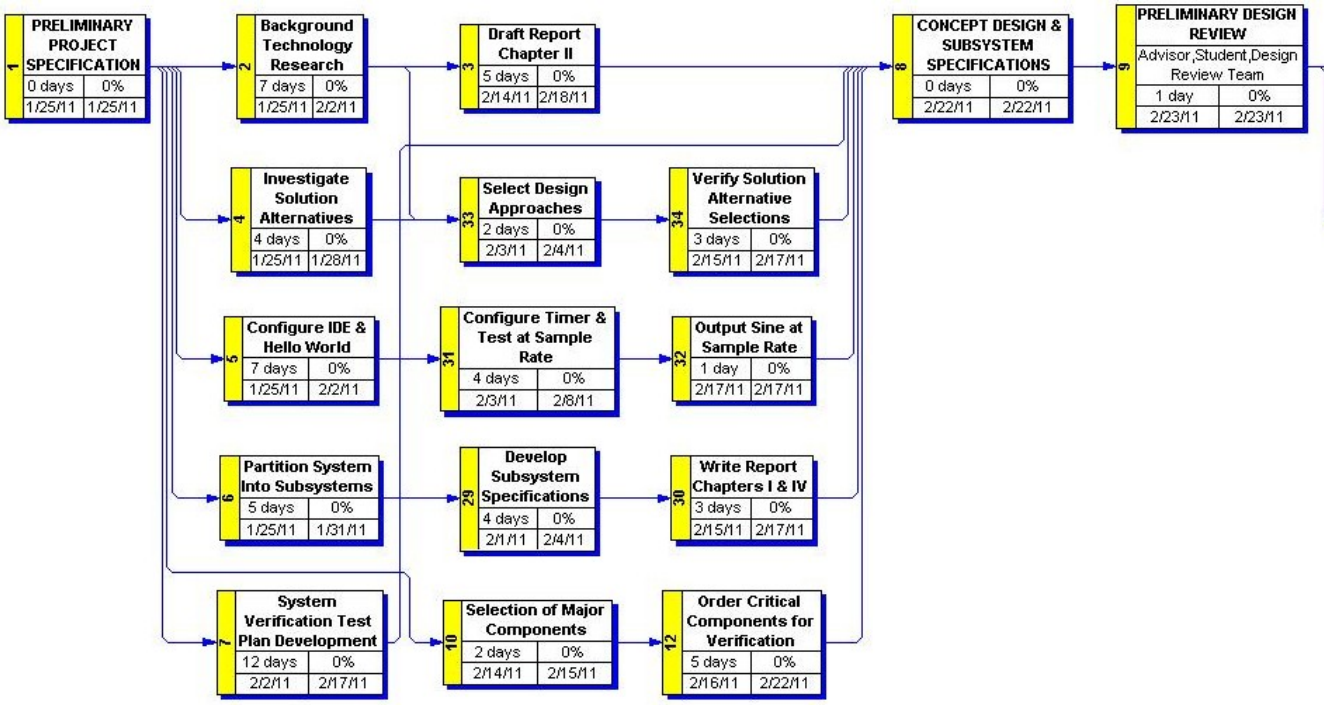

Figure C.1: Schedule from 1/25/11 to 2/23/11

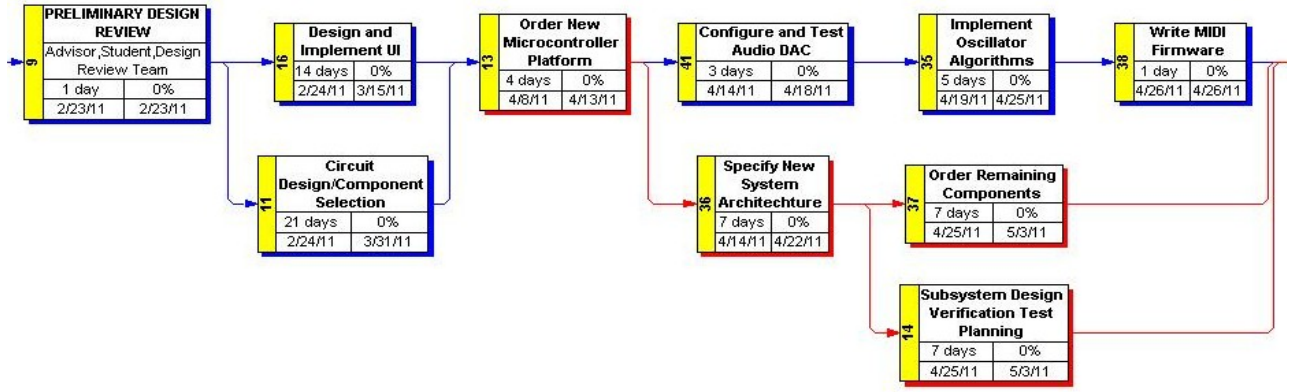

Figure C.2: Schedule from 2/23/11 to 4/25/11

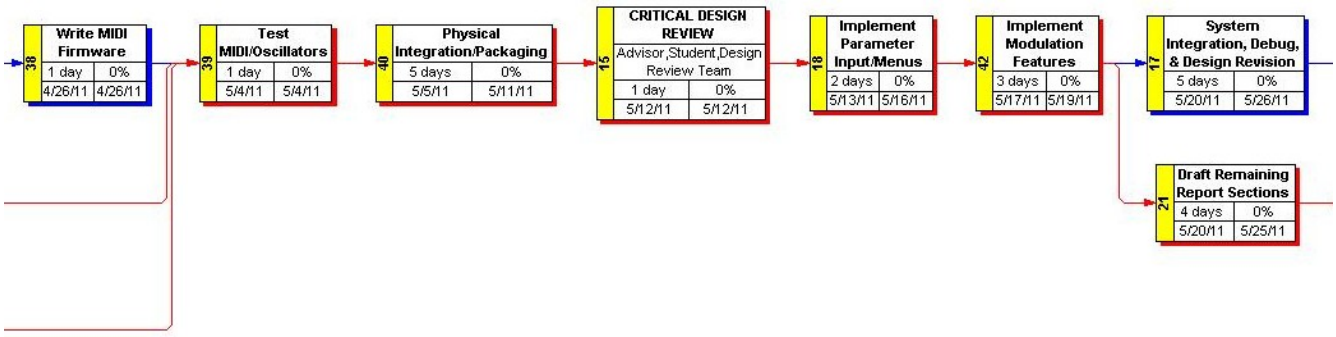

Figure C.3: Schedule from 4/25/11 to 5/26/11

Figure C.4: Schedule from 5/26/11 to 6/10/11

Appendix C: Program Listing

(Code contained in installed libraries or modules not shown)

```
/*
 * main.xc
 *
 *  Created on: Apr 15, 2011
 *      Author: justintomlin
 */


#include <platform.h>
#include <xclib.h>
#include <print.h>
#include <xs1.h>
#include "mathf8_24.h"
#include "audio_out.h"
#include "xlog_server.h"
#include "midi.h"
#include "buffer.h"
#include "tuning.h"
#include "uart.h"
#include "oscillator.h"
#include "spi_master.h"
#include "adc.h"
#include "control.h"

#define CLK_DIV 50


on stdcore[0] : struct aud_interface rj_aud =   {
                XS1_CLKBLK_1,
                XS1_CLKBLK_2,
                XS1_CLKBLK_3,
                XS1_PORT_1M, // aud_dout
                XS1_PORT_1N, // aud_sclk
                XS1_PORT_1O};// aud_fsync

on stdcore[3] : struct spi_master_interface spi = {
                XS1_CLKBLK_1,
                XS1_CLKBLK_2,
                XS1_PORT_1A, // mosi
                XS1_PORT_1D, // sclk
                XS1_PORT_1B};// miso

on stdcore[3] : out port chip = XS1_PORT_1C;

on stdcore[3] : port midi_rxd = XS1_PORT_1E;

on stdcore[0]: port p_tx = XS1_PORT_1P;

on stdcore[1]: in port btn = XS1_PORT_1A;


int main()
{
        chan midi, midi_buf, notes, disp, params, btn_s, noteon, adc, st;
        streaming chan control;
```

```
        streaming chan c_out;
        par {
                on stdcore[3] : read_midi(midi_rxd, midi);
                on stdcore[3] : buffer(midi, midi_buf);
                on stdcore[3] : {
                        spi_init(spi, 100);
                        adc_read(spi, chip, adc);
                }
                on stdcore[3] : lcd_disp(disp);
                on stdcore[1] : parse_midi(midi_buf, notes, noteon);
                on stdcore[1] : button_event(btn, btn_s);
                on stdcore[1] : param_update(adc, btn_s, params, disp, st);
                on stdcore[1] : controls(params, st, control, noteon);
                on stdcore[2] : oscillator(notes, control, c_out);
                on stdcore[0] : output_setup(rj_aud, CLK_DIV, c_out);
                on stdcore[0] : xlog_server_uart(p_tx);
        }
    return 0;
}

                                                /*

 * oscillator.c
 *
 *  Created on: May 4, 2011
 *      Author: justintomlin
 */

#include "oscillator.h"
#include <platform.h>
#include "mathf8_24.h"
#include <xs1.h>

#include <xclib.h>
#include "adc.h"
#include "tuning.h"
#include "float.h"
#include "cosn.h"

void oscillator(chanend incr, streaming chanend control, streaming chanend c_out)
{
        int y;

        unsigned long ph = 0;
        unsigned long shift_ph = 0;
        int inc = 0;
        int shift_inc, shift = 0;
        unsigned revdat = 0;

        int k = 1 << 24;
        int a = 0;
        int n = 1 << 24;
        int A= 1 << 24;
        int curnote, note = 0;

        while(1) {
                select{
                        case incr :> note:
                                if (note >= MIN_MIDI_NOTE && note <= MAX_MIDI_NOTE) {
                                        // Note on
                                        inc = midi_freq[note - MIN_MIDI_NOTE];
```

```
                                curnote = -note;
                                }
                        break;
                default:

                        //stream in oscillator parameter control signals
                        control :> A;
                        control :> k;
                        control :> n;
                        control :> a;
                        control :> shift;

                        //implement frequency shift relative to fundamental frequency
                        shift_inc = mulf8_24(shift, inc);

                        //wrap around phases
                        if(ph >= PI2) ph -= PI2;
                        else if (ph < 0) ph += PI2;
                        if(shift_ph >= PI2) shift_ph -= PI2;
                        else if (shift_ph < 0) shift_ph += PI2;

                        //implement fixed-point oscillator algorithm
                        y = mulf8_24(mulf8_24(expf8_24(mulf8_24(k, cosf8_24(ph) - ONE)),
mulf8_24((ONE-a), cosf8_24(mulf8_24(n, ph + shift_ph))) + mulf8_24(a, cosf8_24(mulf8_24((n + ONE), ph +
shift_ph)))), A) >> 10;

                        //prepare data for MSB-first serial transmission
                        revdat = bitrev(y) >> 16;

                        //stream data to output
                        c_out <: revdat;

                        //increment phase (fixed point frequency)
                        ph += inc;

                        if(shift_inc != 0) shift_ph += shift_inc;
                        else shift_ph = 0;
                        break;
                }
        }
}

/*
 * control.xc
 *
 *  Created on: May 11, 2011
 *      Author: justintomlin
 */

#include "control.h"
#include <platform.h>
#include <xs1.h>
#include <print.h>
#include <xclib.h>
#include "uart.h"
#include "tuning.h"
#include "mathf8_24.h"

void lcd_disp(chanend data)
{
        chan p;
```

```
        int state;
        par{
                int_out(p);
                {
                        char decBuffer[] = {0,0,0};
                        int param0, param1, param2, param3;

                        while(1)
                        {
                                select{
                                        case data :> state:
                                                data :> param0;
                                                data :> param1;
                                                data :> param2;
                                                data :> param3;

                                                break;
                                        default:

                                                p <: state;
                                                p <: param0 >> 3;
                                                p <: param1 >> 3;
                                                p <: param2 >> 3;
                                                p <: param3 >> 3;

                                                break;
                                }
                        }
                }
        }
}

void int_out(chanend p)
{
        char decBuffer[] = {0,0,0,0};
        int pos=0;
        int val[4] = {0,0,0,0};
        int state = 0;

        while(1)
        {
                select{

                        case p :> state:
                                for(int i=0; i<4; i++) p :> val[i];
                                break;

                        default:
                                printchar(254);
                                printchar(128);

                                // choose menu title:
                                switch(state){
                                        case 0:
                                                printstr("   oscillator   \n");
                                                break;
                                        case 1:
                                                printstr(" amp modulation \n");
                                                break;
                                        case 2:
```

```
                                        printstr("  k modulation  \n");
                                        break;
                                case 3:
                                        printstr(" a / shift mod  ");
                                        break;
                                }

                                for(int j=0; j<4; j++)
                                {
                                        // convert int to string:
                                        decBuffer[2] = (val[j] % 10) + '0';
                                        val[j] = val[j]/10;
                                        decBuffer[1] = (val[j] % 10) + '0';
                                        val[j] = val[j]/10;
                                        decBuffer[0] = (val[j] % 10) + '0';
                                        decBuffer[3] = ' ';
                                        pos = 64 + 4*j;
                                        printchar(254);
                                        printchar(128 + pos);

                                        for(int i=0; i<4; i++)
                                                printchar(decBuffer[i]);
                                }

                                break;
                        }
                }
        }

void button_event(in port btn, chanend btn_s)
{
        int x = 0;
        int i = 0;
        int count = 0;
        unsigned time;
        timer t;
        btn:> x;
        btn_s <: 0; // initial state = 0

        while(1) {
                btn when pinsneq(x) :> x; //wait for button change
                t :> time;
                count = !count;
                if (count == 1) //check if change is 0 to 1 transition
                        btn_s <: (++i)%4;
                time += 100000; //wait 1 ms before checking again (for debouncing purposes)
                t when timerafter(time) :> void;
        }
}

void param_update(chanend adc, chanend b_st, chanend params, chanend disp, chanend st)
{
        int state;
        int adc_val[4] = {0,0,0,0};
        int o_params[4][4];
        int cv_flag[4] = {0,0,0,0};

        //initialize 2D parameter array (o_params[state][value index])
        for(int i=0; i<4; i++) for(int j=0; j<4; j++) o_params[i][j] = 0;

        while(1)
```

```
	{
		select{
			case b_st :> state:
				st <: state; //send state to control
				for(int i=0;i<4;i++) cv_flag[i] = 1; // set current value flag

				break;
			case adc :> adc_val[0]: //on reception of first adc value, read in
				adc :> adc_val[1];  //the next three
				adc :> adc_val[2];
				adc :> adc_val[3];
				break;
			default:

				for(int i=0; i<4; i++)
				{
					// if knob turned to position of past value of parameter after
state
					// change, update oscillator parameters with current adc values
					if((adc_val[i] < o_params[state][i] + 50) && (adc_val[i] <
o_params[state][i] + 50) && (cv_flag[i] = 1))
						cv_flag[i] = 0;
					if(cv_flag[i] == 0) o_params[state][i] = adc_val[i];
				}
				//send parameters to control
				for(int i=0; i<4; i++) params <: o_params[state][i];
				//send parameters to display
				disp <: state;
				for(int i=0; i<4; i++) disp <: o_params[state][i];

				break;
		}
	}

}


/*                        param:
 *                     0      1      2      3
 *  state: -------------------      menu:
 *       0      |    k      n      a      shift | fundamental
 *       1      |    A      f      at   dec   |    A_mod
 *    2  |    A      f      at      dec   | k_mod
 *       3      |    A      f      A        f    | a_lfo/shift_lfo
 *               -------------------
 */
void controls(chanend param, chanend st, streaming chanend control, chanend noteon)
{
	int state = 0;
	int o_param[4][4]; // [state][param#]
	int cvflag[4] = {0,0,0,0};
	unsigned time;
	timer t;
	unsigned A, k, n, a, shift_inc;
	int lfoAph, lfokph, lfoaph, lfosph, Aatt, Adec, Katt, Kdec, trig, trig1 = 0;
	int Aenv, Kenv = 0;
	int Adec1 = 250000;

	for(int i=0; i<4; i++) for(int j=0; j<4; j++) o_param[i][j] = 0;

	t :> time;
```

```
	while(1)
	{
		#pragma ordered

		select {
			case st :> state:
				for(int i=0; i<4; i++) cvflag[i] = 1;
				break;
			case param :> o_param[state][0]:
				param :> o_param[state][1];
				param :> o_param[state][2];
				param :> o_param[state][3];
				break;
			case noteon :> trig:
				trig1 = 1;
				//Kenv = 0;
				break;
			default:

				Aatt = ONE/((o_param[1][2]<<6) + 625);
				Adec = ONE/((o_param[1][3]<<6) + 625);

				Katt = ONE/((o_param[2][2]<<6) + 625);
				Kdec = ONE/((o_param[2][3]<<6) + 625);

				if(lfoAph >= PI2) lfoAph -= PI2;
				else if (lfoAph < 0) lfoAph += PI2;

				if(trig == 1)
				{
					if(trig1 == 1)
						{
							Aenv -= Adec1;
						}
					else
					{
						if(Aenv < ONE)
							Aenv += Aatt;
						if(Kenv < ONE)
							Kenv += Katt;
					}
				}
				else if(trig == 0)
				{
					if(Aenv > 0)
						Aenv -= Adec;
					if(Kenv > 0)
						Kenv -= Kdec;
				}
				if(Aenv > ONE) Aenv = ONE;
				if(Aenv < 0)
					{
					Aenv = 0;
					trig1 = 0;
					}
				if(Kenv > ONE) Kenv = ONE;
				if(Kenv < 0) Kenv = 0;

				A = mulf8_24(Aenv, ONE - mulf8_24((o_param[1][0] << 12), mulf8_24(HALF,
ONE+cosf8_24(lfoAph))));
```

```
                                k = mulf8_24(Kenv+1, mulf8_24(mulf8_24(o_param[2][0] << 12,
cosf8_24(lfokph)), o_param[0][0] << 17) + (o_param[0][0] << 17));
                                n = ((o_param[0][1] >> 9) << 24);
                                a = mulf8_24(ONE - mulf8_24((o_param[3][0] << 12), mulf8_24(HALF,
ONE+cosf8_24(lfoaph))), o_param[0][2] << 12);
                                shift_inc = mulf8_24(ONE - mulf8_24((o_param[3][2] << 12), mulf8_24(HALF,
ONE+cosf8_24(lfosph))), o_param[0][3] << 12);

                                control <: A;
                                control <: k;
                                control <: n;
                                control <: a;
                                control <: shift_inc;

                                lfoAph += (o_param[1][1] << 2);
                                lfokph += (o_param[2][1] << 2);
                                lfoaph += (o_param[3][1] << 2);
                                lfosph += (o_param[3][3] << 2);


                                break;
                        }
                }
        }
}
/*
 * adc.xc
 *
 *  Created on: May 5, 2011
 *      Author: justintomlin
 */

#include "adc.h"


void adc_read(spi_master_interface &spi, out port chip, chanend param)
{
        unsigned time;
        timer t;
        int adc0, adc1, adc2, adc3;
        chip <: 1;
        t :> time;
        while(1) {
                // read in four ADC values
                chip <: 0;
                spi_out_byte(spi, 0b10010111);
                adc0 = (spi_in_short(spi) >> 3);

                chip <: 1;
                chip <: 0;
                spi_out_byte(spi, 0b11010111);
                adc1 = (spi_in_short(spi) >> 3);

                chip <: 1;
                chip <: 0;
                spi_out_byte(spi, 0b10100111);
                adc2 = (spi_in_short(spi) >> 3);

                chip <: 1;
                chip <: 0;
                spi_out_byte(spi, 0b11100111);
                adc3 = (spi_in_short(spi) >> 3);
```

```
                // output current adc values to channel
                param <: adc0; param <: adc1; param <: adc2; param <: adc3;
                chip <: 1;

                // slow down ADC value data rate
                time += DELAY;
                t when timerafter(time) :> void;
        }
}
/**
 * Module:  module_spi_master
 * Version: 1v0
 * Build:   21a5617cdbae74496f778b2a7da0f0661babbd36
 * File:    spi_master.xc
 *
 * The copyrights, all other intellectual and industrial
 * property rights are retained by XMOS and/or its licensors.
 * Terms and conditions covering the use of this code can
 * be found in the Xmos End User License Agreement.
 *
 * Copyright XMOS Ltd 2010
 *
 * In the case where this code is a modification of existing code
 * under a separate license, the separate license terms are shown
 * below. The modifications to the code are still covered by the
 * copyright notice above.
 *
 **/
////////////////////////////////////////////////////////////////////////////
// Select lines are intentionally not part of API
// They are simple port outputs
// They depend on how many slaves there are and how they're connected
//

#include <xs1.h>
#include <xclib.h>
#include "audio_out.h"

void output_setup(aud_interface &i, int spi_clock_div, streaming chanend c_out)
{

        // configure ports and clock blocks
        configure_clock_rate(i.blk1, 100, spi_clock_div);
        configure_out_port(i.sclk, i.blk1, 0);
        configure_clock_src(i.blk2, i.sclk);
        configure_out_port(i.mosi, i.blk2, 0);
        configure_out_port(i.fsync, i.blk1, 0);

        clearbuf(i.mosi);
        clearbuf(i.sclk);
        clearbuf(i.fsync);
        start_clock(i.blk1);
        start_clock(i.blk2);
        i.sclk <: 0xFFFFFFFF;
        set_thread_fast_mode_on();

        spi_out_aud(i, c_out);

        set_thread_fast_mode_off();
}
```

```
void spi_out_aud(aud_interface &i, streaming chanend c_out)
{
        while(1)
        {
        // MSb-first bit order - SPI standard

    unsigned dout = 0;
    c_out :> dout;

        // MSb-first bit order - SPI standard

    i.mosi <: (dout>>8) & 0xFF;

        i.sclk <: 0xAA;
        i.fsync <: 0x00;
        i.sclk <: 0xAA;

        i.mosi <: (dout) & 0xFF;

        i.sclk <: 0xAA;
        i.fsync <: 0x7F;
        i.sclk <: 0xAA;

        }
}
// Read MIDI messages.
//
// A MIDI interface is a current-loop serial port running at 31250 baud. The
// normal way of implementing a MIDI input is to use an optocoupler, a diode,
// and a couple of resistors. Mine looks like this:
//
//                                        ,------o +5V
//                                        |
//                                       | | 680R
//                                       |_|
//                 ____                   |
//      pin 4 o--[____]--+----------.     +------o output to XC1
//                220R   |          | OC |
// MIDI                  - IN4001   V _|/
// input                 ^          -  |\,
//                       |          |    |
//      pin 5 o----------+----------'    `------o ground
//
// The optocoupler I found in my junk box is a CNY 75B, which is really a bit
// slow for MIDI; get a faster one (and check the resistor values against the
// datasheet) if you're building this.
//
// Copyright (c) 2008 Adam Sampson <ats@offog.org>
//
// Permission to use, copy, modify, and/or distribute this software for any
// purpose with or without fee is hereby granted, provided that the above
// copyright notice and this permission notice appear in all copies.
//
// THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
// WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
// MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
// ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
// WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
// ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
```

```
// OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

#include <xs1.h>

#include "midi.h"
#include "tuning.h"

// The length of a MIDI bit, in timer ticks.
// (MIDI is 31250 bits per second.)
#define MIDI_BIT_TIME ((100000000) / 31250)

// Read bytes from a MIDI interface connected to a port.
void read_midi(in port rxd, chanend cout) {
	while (1) {
		int m = 0, t;

		// Wait for a start bit.
		// The MIDI input is normally 1.
		rxd when pinsneq(1) :> int _ @ t;

		// Sample near the end of the bits from now on.
		// (My optocoupler's a bit laggy...)
		t += MIDI_BIT_TIME / 2;

		// Read 8 bits, shifting them into m.
		for (int i = 0; i < 8; i++) {
			t += MIDI_BIT_TIME;
			rxd @ t :> >> m;
		}
		m = (m >> 24) & 0xFF;

		// Discard the stop bit.
		t += MIDI_BIT_TIME;
		rxd @ t :> int _;

		cout <: m;
	}
}

// Parse MIDI messages and produce a stream of notes.
// A note is a positive int for note on, and a negative int for note off.
void parse_midi(chanend midi, chanend notes, chanend trig) {
	while (1) {
		int m, key, vel, curnote;

		midi :> m;

		switch (m & 0xF0) {
		case 0x80: // note off
			midi :> key;
			midi :> vel;

			notes <: -key;
			if(key == curnote)
				trig <: 0;
			break;
		case 0x90: // note on
			midi :> key;
			midi :> vel;

			curnote = key;
```

```
                        if (vel == 0) {
                                // For some reason, my Studiologic keyboard
                                // sends "note on" with velocity 0 instead of
                                // "note off".
                                notes <: -key;
                        } else {
                                notes <: key;
                                trig <: 1;
                        }

                        break;
                default:
                        // Data (or something we don't handle) -- ignore until
                        // we get a command to resync.
                        break;
                }
        }
}
// An N-place FIFO buffer.
//
// Copyright (c) 2008 Adam Sampson <ats@offog.org>
//
// Permission to use, copy, modify, and/or distribute this software for any
// purpose with or without fee is hereby granted, provided that the above
// copyright notice and this permission notice appear in all copies.
//
// THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
// WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
// MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
// ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
// WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
// ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
// OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

#include "buffer.h"

static void buffer_l(chanend cin, chanend req, chanend resp) {
#define SIZE 512
        int buf[SIZE];
        int wpos = 0, rpos = 0, used = 0;

        while (1) {
                #pragma ordered
                select {
                case (used < SIZE) => cin :> buf[wpos]:
                        wpos = (wpos + 1) % SIZE;
                        ++used;
                        break;
                case (used > 0) => req :> int _:
                        resp <: buf[rpos];
                        rpos = (rpos + 1) % SIZE;
                        --used;
                        break;
                }
        }
#undef SIZE
}

static void buffer_r(chanend req, chanend resp, chanend cout) {
        while (1) {
```

```
                int value;

                req <: 0;
                resp :> value;
                cout <: value;
        }
}

void buffer(chanend cin, chanend cout) {
        chan req, resp;

        par {
                buffer_l(cin, req, resp);
                buffer_r(req, resp, cout);
        }
}
```

Appendix D: Analysis of Senior Project Design

Project Title: Multicore, Multithreaded, Phase-synchronous FM Sound Synthesizer

Student's Name: Justin Tomlin

Advisor's Name: Dr. Pilkington

Summary of Functional Requirements:

- At least one oscillator capable of generating spectrally dense waveforms with wave-shape options and minimal undesirable artifacts (aliasing, heterodyning, etc...)
- MIDI control
- Spectral shaping with resonance capabilities
- Amplitude and oscillator parameter modulation envelopes
- Low-frequency oscillator modulation of amplitude and oscillator parameters
- Parameter adjustment via knobs, buttons, and display
- Mono line-out
- No perceptible latency
- Powered by “wall-wart”
- Small/portable
- Durable

Primary Constraints:

The primary setback in this project was initially using a microcontroller development platform that was unable to interface with any DAC on the market (that I could find) that would fulfill the performance specifications outlined for this design. After further research, it was found that a DAC that communicates via I2S, left- or right-justified audio format was necessary to output high quality audio. The Copper AVR32 development system initially chosen contains a model of the UC3 microcontroller that doesn't have the serial synchronous controller peripheral necessary for sending data in these formats. Bit-banging was far too CPU-intensive, and impractical due to tight timing requirements. To be able to interface with suitable peripherals, the microcontroller development platform was switched to the XMOS XC-1A.

Economic Considerations:

See Appendix B for cost estimates and bill of materials.

Originally it was estimated that the project would be ready for its first demonstration by 4/27/11. In actuality, the project took until 5/31/11 to finish development and testing was carried out through the last week of the quarter. This is primarily due to trouble with the first microcontroller platform chosen.

If manufactured on a commercial basis:

N/A, not ready for commercial manufacture.

Environmental:

There is relatively little environmental impact associated with the use of this device. The device runs on a 400 mA wall transformer, so the power requirement isn't too large. As far as manufacture, lead-free solder was used on this project, and RoHS compliant components were chosen whenever available.

Manufacturability:

To manufacture this product, custom PCBs to house the components should be developed (for durability, cost savings, and ease of manufacture. The XC-1A development board should be replaced with a custom board to house the XS1-G4 microcontroller and its necessary hardware circuitry. Wire jumpers should be replaced with IDC headers, cables, and plugs for durability and ease of assembly.

Sustainability:

The device in its current form has a high probability of developing loose panel connections due to the use of jumpers to make connections between boards and the front panel. This can easily be remedied through the use of headers and cable assemblies.

This device has relatively low impact on the sustainable use of resources. It is intended to be used long-term as a electronic musical instrument.

Addressing noise issues would improve the quality of this design. Redesigning the board layout and power supply would help attenuate analog sources of noise. Experimentation with cosine calculation methods, digital filtering, increase in sample rate, and increase in resolution would help attenuate digital sources of noise. Finally digital effects and stereo would be highly marketable design upgrades.

Upgrading this design is relatively simple. The resources on the XS1-G4 are far from exhausted, so there is room for additional functionality.

Ethical:

The ethical concerns relating to the design, manufacture, or use of this project are relatively minimal.

Social and Political:

The social and political concerns relating to the design, manufacture, or use of this project are relatively minimal.

Development:

During the course of this project, I have learned and developed many tools and techniques for design. In developing this project, my prototyping and troubleshooting skills have vastly improved. I also have a much better idea of issues that need to be considered when researching data sheets during paper design, which would have helped me avoid my initial design setback. I have become much more proficient in C programming for embedded systems. I have also gained experience working with microcontrollers that include an architecture that supports hardware parallel processing. Through working with threads and thread communications, I have developed a much better understanding of event-driven software design. Finally, I have become much more acquainted with some of the considerations that must be taken in digital audio hardware and software design.