18-551,Spring 2002
Group 12, Final Report

MASH
Modularly Architectured Synthesizer in Hardware

Benjamin B. Lyon (bbl@andrew.cmu.edu)
Michelle Ng (moshuen@andrew.cmu.edu)
Eduardo Neto (eneto@andrew.cmu.edu)

Table Of Contents

THE PROBLEM 3

WHAT WE DID 6

PERFORMANCE ANALYSIS 20

PRIOR ART 21

OUR FINAL DEMO 21

CONCLUSION 22

TERMS AND DEFINITIONS 23

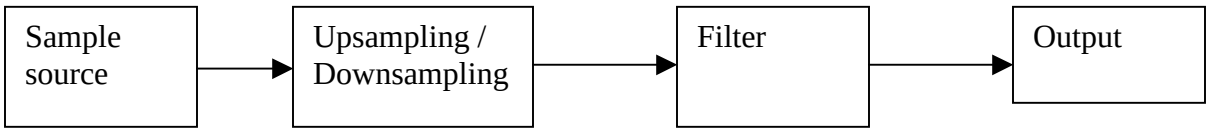
The Problem

Two major problems exist in hardware synthesizers to date. Firstly, hardware synthesizers are limited to a predefined, single synthesis method, filter order, and filter set. The closest the current state of the art comes to such an implementation is the super expensive Clavia NORD Modular[1], which allows for re-routing, but limits the user to hard-coded filter definitions. No hardware synthesizer to date (NORD included) has the ability to alter the filter/modulator set. Secondly, when experiencing DSP overload, both hardware (DSP based) and software (PC based) synthesizers miss notes or introduce significant latency when converting user input to audio output. This is highly problematic in that, for composers and musicians to get the kind of flexibility and performance they require, they have to fill their studios with multiple racks filled with different types of synthesizers, or purchase expensive software and cope with the latency issues.

Part of the reason that hardware synthesizers haven't adopted the aforementioned features is that implementing just one type of synthesis takes all the processing power that older DSPs can provide. However, with faster, cheaper, and more powerful DSPs coming out on the market, this is limitation is rapidly becoming less of an issue. While much focus has been placed on software modular synthesizers, so that they are available today[2], "soft synths" are not appropriate for real time synthesis because of the significant latencies involved, not to mention the instability of the underlying operating systems which is out of the developer's control – softsynths are simply too unstable for live use, and is the key reason why they're not widely adopted in the music industry for live performances. Hence, there is still an opportunity available in the hardware synthesizer market for the creation of a flexible "all in one" synthesizer that gracefully combines the features of the major types of synthesizers as well as provides modularity and graceful degradation in performance—all into one box.

Below are displayed the basic block diagrams for the current major types of synthesis available in hardware synthesizers today. Currently, each of the following diagrams represents a separate piece of hardware. Our goal is to be able to combine/mix/match the main subcomponents of each synthesis method together.

Sample Playback Synthesis (also known as Wavetable Synthesis)



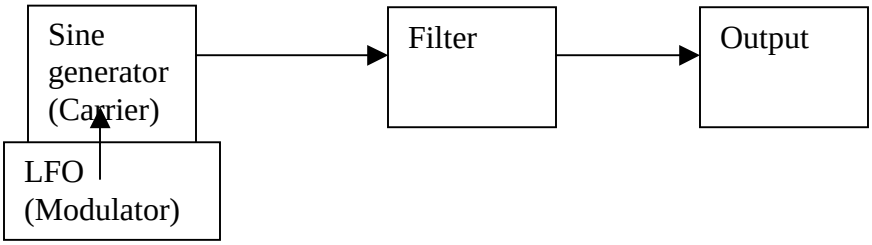
This method employs the continuous playback of a previously recorded sound sample, performing upsampling / downsampling as appropriate depending on the desired

[1] http://www.clavia.se/modular.htm
[2] http://www.software-technology.com/

fundamental frequency (ie: what note is being played on a keyboard). The filter is typically used mostly as an effect and does not play a significant role in the synthesis aspect, as the desired sound is already in sampled form.

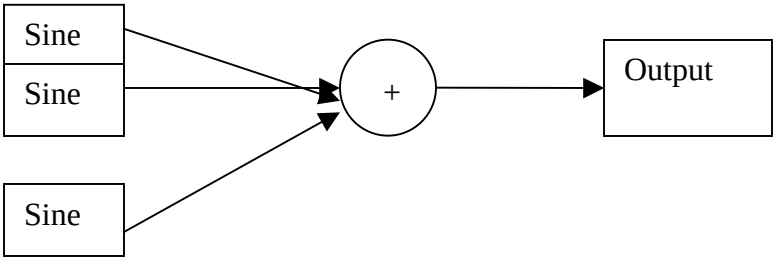
Examples: "Wavetable" sound cards, PCM sound modules.

FM Synthesis



Frequency modulation synthesis employs the use of a combined unit named an operator. It consists of a sine wave generator (the carrier signal) being modulated by a low frequency oscillator (LFO), where the frequency of this LFO is a "musically meaningful" frequency such as 110Hz (A2 on the piano). The filters and envelopes are applied to further modify the sound in order to synthesize instruments such as bells, organs and certain percussion instruments. Examples: Yamaha synths developed in the 70s and 80s (DX7, TX81Z, DX100), Older sound chips used in videogames and computer sound cards (The very popular Yamaha OPL series chips used on the sound blaster and adlib cards, among others)

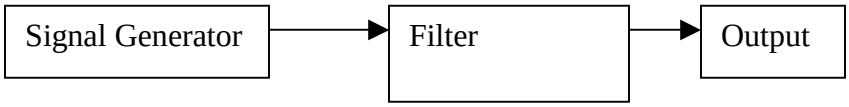
Additive/Fourier Synthesis



Stemming from the notion that any complex waveform may be generated by the addition of simple sine waves, additive synthesis employs the usage of ***many*** sine wave generators simultaneously (upwards of 100 in certain cases), each with a different amplitude and frequency in order to synthesize the harmonic components of a complex signal. Given the sheer complexity of the circuitry needed to support so many sine generators, synths employing this method are rare and extremely expensive. Likewise, implementing this method of synthesis in software is very computationally intensive. Examples: The legendary Hammond B3 organ.[3]

Subtractive Synthesis:

[3] http://www.obsolete.com/120_years/machines/hammond/



Subtractive synthesis is the most popular type of synthesis used in vintage and "virtual" DSP-based analog synthesizers. Whereas additive synthesizers are expensive to build due to the sheer quantity of oscillators needed, subtractive synths "cut corners" by using less signal generators, but with a richer harmonic component than that of a clean tone sine wave which (ideally) only has one harmonic at its fundamental frequency. Hence, by using square and sawtooth waves for example, the number of oscillators necessary to synthesize a complex sound is drastically reduced. However the side effect is that there will be frequency components that are not desired – at that point filters are used to subtract the undesired components, hence the term "subtractive" synthesis. Examples: Moog Minimoog, Roland Jupiter 8, Seq. Circuits Prophet 5, virtually all analog synthesizers[4].

Whereas virtually all hardware synthesizers in existence employ only one of the above methods, our flexible architecture would allow one to mix and match properties of each synthesis methods to develop new "hybrid" synthesis methods previously unattainable. By allowing this, the instrument would be capable of generating a wider range of possible sounds – and this along with sound quality (ie: sampling rate, bit depth, SNR of DAC) are the most desirable qualities of synthesizers, specially in the analog modeling market, where musicians are not so much interested in how well a synthesizer can emulate a real instrument as much as the wide range of sonic landscapes capable of being produced by the synthesizer.

The goal of our project is to combine the flexibility offered by new software synthesizers with the real-time functionality of current hardware synthesizers. In addition, we will add a couple "bells and whistles", namely, analog modeling. This synthesizer will have a modular architecture that will lend itself to the easy introduction of new synthesis methods, filter modules, and synthesis system paths.

[4] http://music.ashbysolutions.com/modsynth/generatn.html

What We Did

Introduction

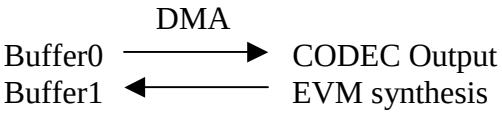
While developing our system architecture, we had two main goals in mind. First and most important, the system had to work in real time with low latency, since it was designed with live performance in mind. In other words, as notes are being played on the keyboard, the corresponding sound for a given configuration had to not only be synthesized in real time, but it also had to be responsive such that once a key is pressed, there should be no noticeable lag from that time until the sound is heard. The other goal was that the architecture had to be modular. Any type of synthesizer could then be built by combining basic DSP blocks together to form a specific signal path, be it say, additive or subtractive synthesis which both require very different signal paths. The entire system was designed and optimized from the ground up with these base requirements in mind – what follows is a detailed description of how we achieved this goal.

Real-Time/Latency Issues

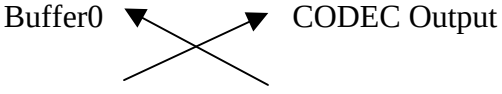
Since we were using the built-in CODEC for audio output, we first utilized the provided code from lab1, but extended it to allow for double buffering. We quickly realized that this was not satisfactory, since the interrupt was set to be executed every time a sample was consumed by the CODEC. This places too much of a burden on the CPU making any sort of advanced signal processing in real time impossible. A different system was then devised in order to offload the cpu. The logical choice was to use DMA for all audio transfers, which is precisely the way typical PCs output audio to sound cards. This way one buffer block is being transferred at once, and an interrupt is only raised when the entire block has been transferred, as opposed to a single sample. In our case, we set our block size to 256 samples (512 bytes). This means that the CPU is interrupted once every 256 samples as opposed to every single sample. Since there is overhead of setting and restoring register state with each interrupt, the benefits of using DMA become obvious. When the interrupt is raised, we toggle between buffers, such that the audio transmission is not interrupted at any time.

Double Buffering w/ DMA Example

DMA operating on buffer 0:



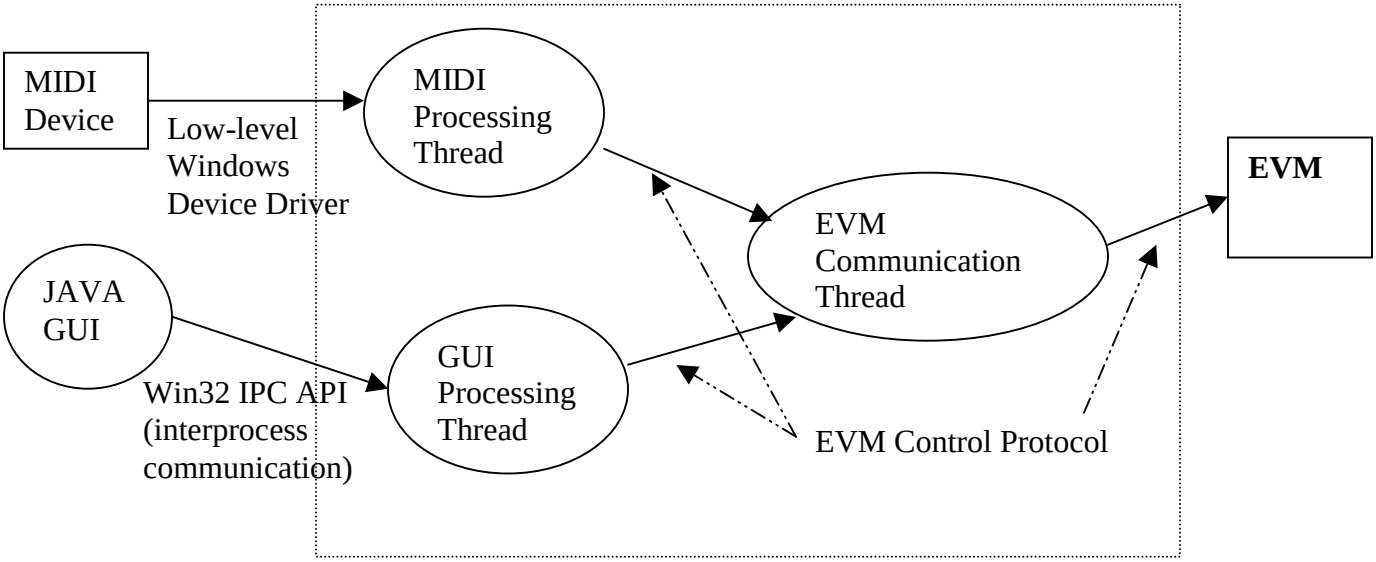
DMA raises interrupt (buffer switch occurs):



Buffer1 EVM synthesis

By using a block size of 256 samples, we immediately add 5.8ms of lantency at 44.1khz (256 / 44100). This is due to double-buffering – what is being output to the CODEC now happened 256 samples ago. Our goal was to keep latency to under 20ms since anything greater than that would be noticeable to the keyboard player. However the buffering scheme is not the only source of latency – the PC side which processes the MIDI input and communicates with the EVM also introduces latency. Hence, we optimized the PC side code by designing a low-latency multithreaded architecture for communicating with the EVM.

Multithreaded Architecture Overview



PC Side Control Program

The control loop on the PC side communicates with the EVM whenever a control message has been received from MIDI or the GUI. The processing time per command on the PC side is typically well under 5ms since all the thread does is take in the data and convert it to the evm control protocol format (explained in the next section). Profiling on the PC side has shown latency of around 3-5ms. However, latency is introduced in the evm side due to its own main control loop which is as follows:

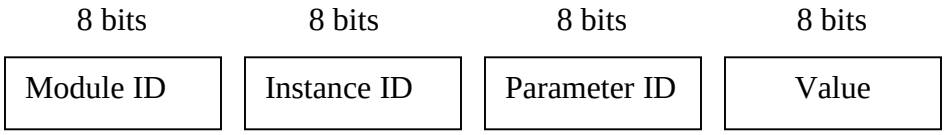
```
while (1) {
        if (go) {
                go = 0;
                processParameters();
                computeSamples();
        }
}
```

When an audio DMA interrupt comes in, the “go” semaphore is signaled meaning we’re ready to process another batch of samples. The EVM then calls processParameters() which goes through its command buffer and updates the variables of all modules accordingly. Then it calls computeSamples() which iterates through all instantiated modules and computes the samples for each. Whichever module is set as the final output module will have its buffer routed to one of the 2 DMA buffers (the opposite of the one being currently processed). Since we only process the command buffer once per control iteration (with a period of 256 / 44100), in the worst case, the command will have just missed the current iteration, yielding 5.8ms. On top of that, due to the double buffering as previously explained, we introduce another 5.8ms. And given the PC side / MIDI latency of 5ms max, we achieve a maximum latency of 5.8+5.8+5 = 16.6ms which is well under our goal of <20 ms.

EVM Control Protocol

We have two sources of control – a MIDI device and the graphical user interface. The MIDI device is used to play notes (from a keyboard) and to change control parameters in real time by controlling hardware knobs. The GUI works like a hardware knob but in software – when sliders are moved around in a given module, the corresponding control change parameter is stored in the evm control buffer. Both the MIDI and GUI processing threads process the input and stores it into a protected evm control buffer which is read by the evm communication thread and written to the EVM. The PC to EVM communication occurs over HPI. At startup, the EVM sends a 32bit message back to the PC containing a pointer to its input buffer. The PC now knows where to write the command data to in the EVM address space over HPI. Whenever the PC writes commands to that buffer, it first writes a minimal header which is simply an integer containing the number of bytes in the buffer, then the buffer data. After writing, it signals the EVM once its done writing by using the HPI interrupt 13h. This is in order to achieve buffer synchronization between the PC and EVM.

The evm control buffer stores commands in our “evm control protocol” – a low overhead control protocol with the following format:



Each control command is 32 bits (one integer) long. The upper byte contains the module ID, which describes the type of module, such as a delay effect or a waveform generator module. With one byte we can have up to 256 different types of modules. The next byte determines the instance ID of that module. This allows the system to contain multiples of a single type of module (up to 256 of each type of module). That allows the system to have for example, 4 waveform generators going through 2 mixers, each going through 2 delay modules, being finally mixed together with one more mixer module. The

third byte contains the parameter ID, which is simply the ID of a variable to be changed. In the case of an oscillator (waveform generator) for example, ID 0 represents a note table from 0 to 127 (mapped onto its corresponding frequency value). The final byte is the value of the parameter given by the previous byte. For example, if we want to modify the waveform type of the first oscillator to a square wave, the command would be:

Module ID: 2 Instance ID: 0 Parameter ID: 3 Value: 2

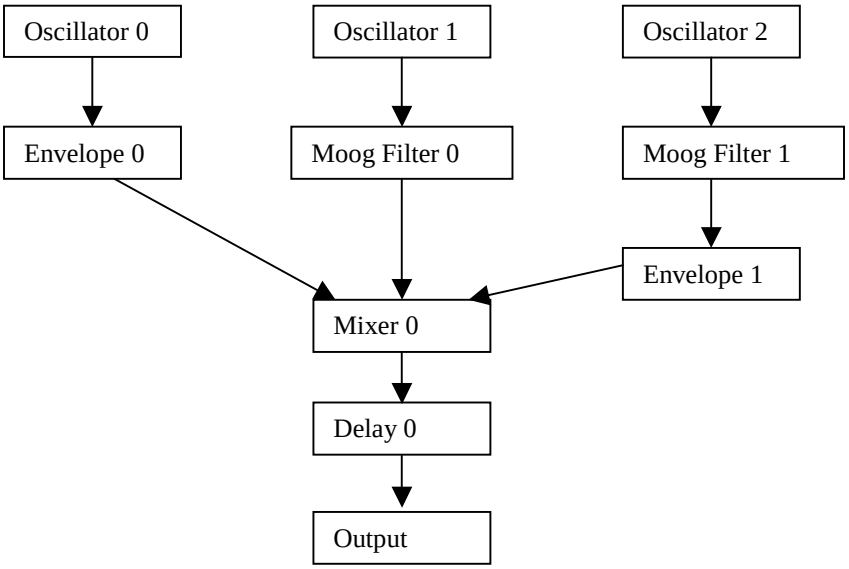
Module ID 2 is the ID for an oscillator (waveform generator) module. Instance ID 0 means it's the first oscillator created. Parameter ID 3 means "waveform type" and value 2 means square wave (0 = sine, 1 = saw, 2 = square, 3 = noise).

Here's an example table of parameter ID's for the waveform generator module (each module has its own parameter ID table depending on its own requirements):

ID	Value meaning
0	Note value from 0 to 127, maps onto musically meaningful frequencies from 8.176 Hz to 12543.85 Hz
1	Trigger value (note on / off) 1 true 0 false (whether keyboard key is currently pressed)
2	Amplitude value from 0 to 127, normalized to –MAXVOLUME ←→MAXVOLUME (uses velocity value from keyboard presses to implement touch sensitive velocity – the harder a key is pressed, the louder the note sounds)
3	Waveform type (0 = sinewave, 1 = sawtooth wave, 2 = square wave, 3 = white noise)
4	Pulse width value from 0 – 255 normalized to 0 –1 (eg: 63 = 0.5 pulsewidth) this is similar to the "duty cycle" function of hardware waveform generators.
5	Finetune value from 0 – 255 normalized to –FINETUNECOEFF ← → FINETUNECOEFF (set to +/- 0.4 by default). This allows for slight tuning and detuning of the oscillators such as when the pitchwheel of the keyboard is modified regardless of the note played. Example: if the note played is A4 (440Hz) and this value is set to +0.1, the resulting freq. becomes 440Hz * 1.1 = 484Hz. The effect is similar to the "wobbling" of "wah wah" effect of bending a guitar string while playing the guitar.

Modular Architecture API

Our major goal besides low-latency and real time performance is that the system should have a modular architecture such that DSP blocks could be composed together in order to achieve any type of audio synthesis desired. We have develop a simple API to do this programmatically, such that modules could be easily created, routed and mixed together. The easiest way to demonstrate this, is to provide an example such as the one below:



In the above configuration, the following calls would be made on the EVM in order to construct the synthesizer:

```
initModuleMap(9);
createModule(0, OSC, 0);
createModule(1, OSC, 1);
createModule(2, OSC, 2);
createModule(3, ENVELOPE, 0);
createModule(4, MOOG, 0);
createModule(5, MOOG, 1);
createModule(6, ENVELOPE, 1);
createModule(7, MIXER, 0);
createModule(8, DELAY, 0);
routeModule(ENVELOPE, 0, 0, OSC, 0, 0);
routeModule(MOOG, 0, 0, OSC, 1, 0);
routeModule(MOOG, 1, 0, OSC, 2, 0);
routeModule(ENVELOPE, 1, 0, MOOG, 1, 0);
routeModule(MIXER, 0, 0, ENVELOPE, 0, 0);
routeModule(MIXER, 0, 1, MOOG, 0, 0);
routeModule(MIXER, 0, 2, ENVELOPE, 1, 0);
routeModule(DELAY, 0, 0, MIXER, 0, 0);
setOutput((getModulePtr(8))->routeOutput[0]);
```

The first call initializes the module map, and its only argument is the total number of modules in the system. Then the module instantiations begin by calling:
createModule(mapID, moduleID, instanceID);
mapID is the global module unique identifier, module ID is the module type (above the names are defined in module.h) and instanceID is the instance of that type.

After the modules are created they must be “connected” by assigning references to their input and output buffers. The function is:
routeModule(in_moduleID, in_instID, input_num, out_moduleID, out_instID, output_num);
The first three parameters indicate the input module type and instance ID, and the input “number” if the module has more than one input (such as a mixer). The next three parameters indicate where the signal is coming from – the module type, instance ID and output number for that module. So in the above example, to route the output of oscillator 2 into the input of moog filter 1, the following call is made:
routeModule(MOOG, 1, 0, OSC, 2, 0);

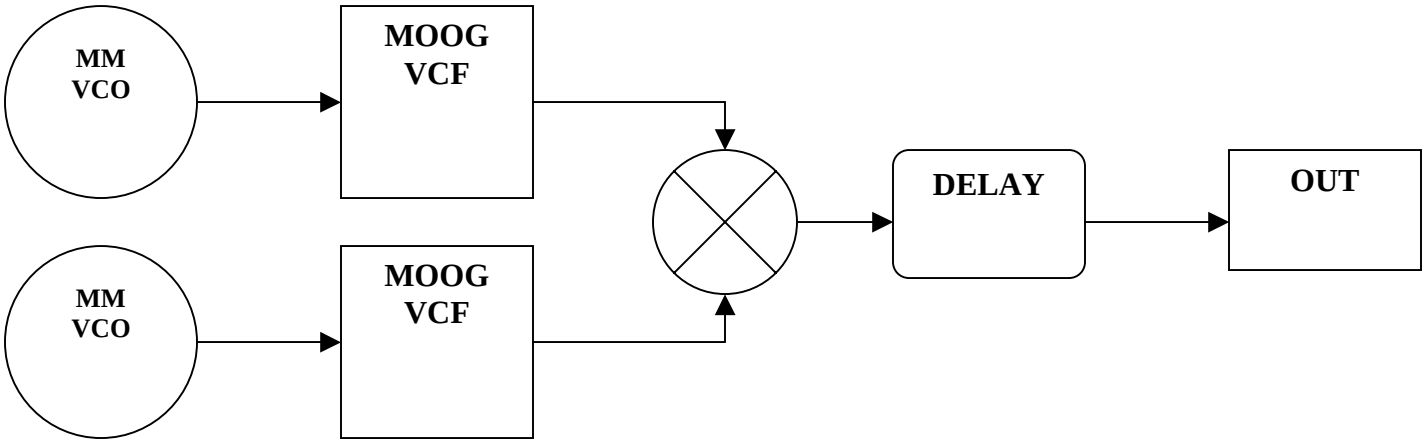
Once all modules are routed together, one module is selected as the final output module. This is the module that will be connected to the EVM DMA audio subsystem. The following call is used for this:
setOutput(float *out);

Where “out” points to the output buffer. In the above example:
setOutput((getModulePtr(8))->routeOutput[0]);

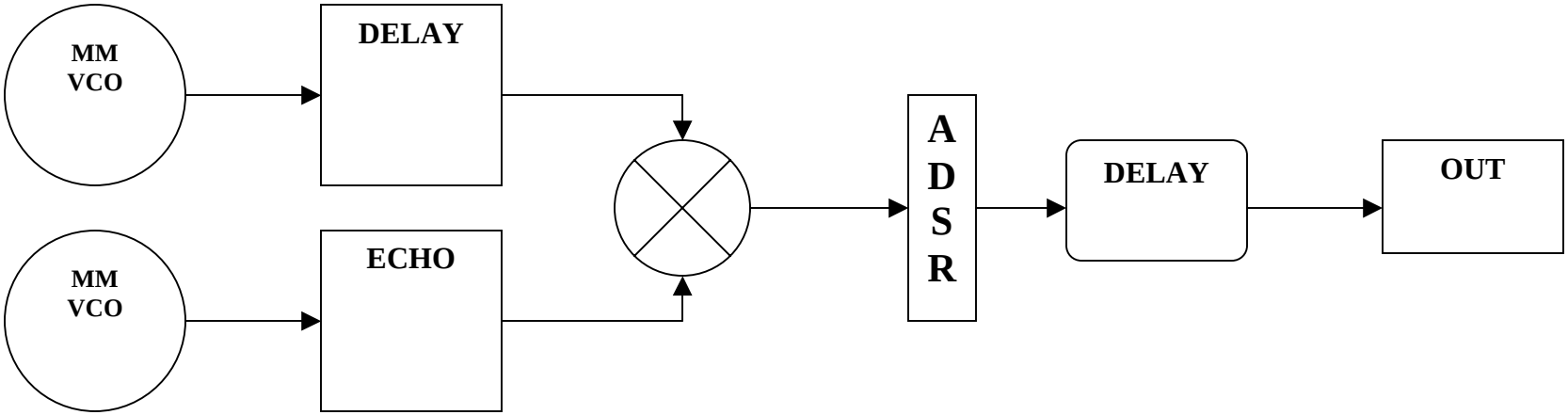
The buffer is acquired from module with mapID 8 (the DELAY 0 module). The getModulePtr() function returns a handle to the module given its unique identifier. The module structure contains an array of output buffers depending on how many outputs it has (most of the times it’ll only have one output, but say, modules that output in stereo or more channels will have more than one). So the above call accesses the first output (output 0) of the delay 0 module. The buffer pointer is at: (getModulePtr(8))->routeOutput[0].

MASH Signal Flow

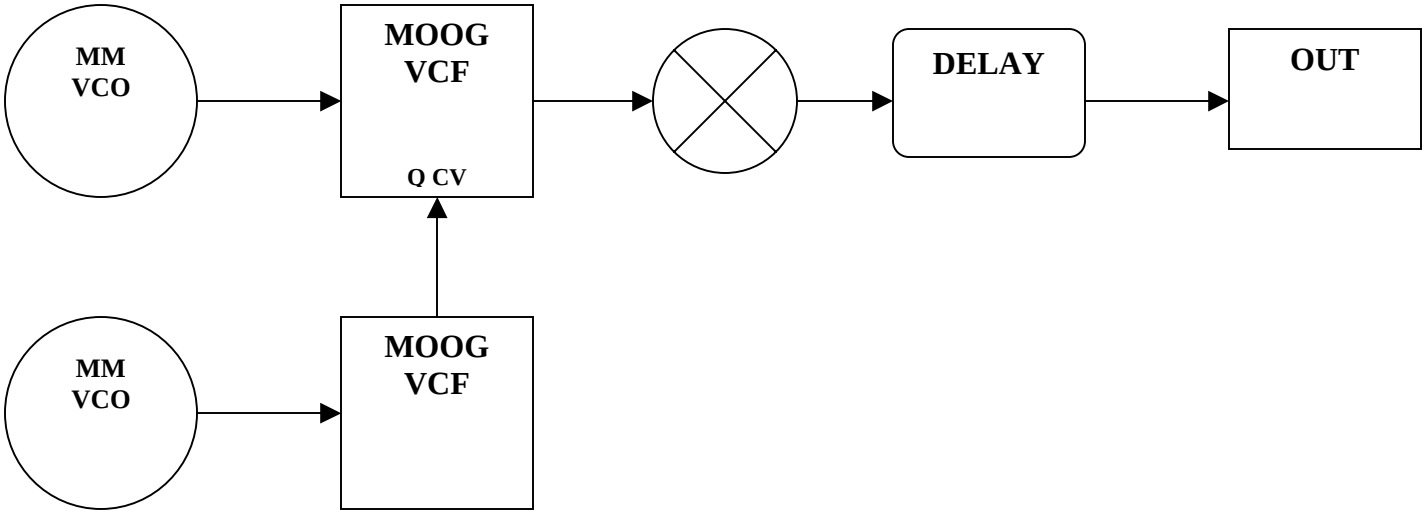
Example Signal Path Configurations / Module Instantiations



Configuration involving two oscillators (Usually an even and an odd harmonic generator) that are mixed after filtration, with delay added to give some reverb. This configuration is great for making organ tones.



Dual oscillator configuration with independently controlled delay and echo paths for each oscillator. This yields a different beat frequency for each harmonic set. Great for analog modeling of higher order distortion effects, or creating heavy vibrato.



Dual oscillator configuration with the second oscillator modulating the resonance of the filter that takes the first oscillator as an input. Great for making metallic sounds, such as cymbals.

Synthesis Modules

MM VCO

The Multi-Mode Voltage Controlled Oscillator (MMVCO) Produces 4 different types of signals

- Sine
- Saw
- Square
- Noise

Different base signals are needed in audio synthesis because they contain different harmonic content. For example, a square wave provides the fundamental frequency as well as successive odd harmonics. We can then select which harmonics we want later via filtration techniques.

Our first approach to generating these signals was writing a single period of the selected wave type to the module's output buffer and then adjusting a read pointer in order to only send the portion of the buffer containing the sample to the CODEC.

This approach had two weaknesses. Firstly, we discovered that in order to meet the real time deadlines, we were going to have to process many samples at once, so an algorithm with no memory, as indicated above was simply untenable. Additionally, the size of the data in the buffer was a direct function of the frequency of the sample, therefore the processing load of the oscillator module and each successive module in the signal path would vary with frequency. This was an unacceptable outcome as well.

The general solution was reached by filling the module output buffer with the selected waveform and then using a pointer to keep track of where in the waveform the last data point in the buffer resided. This way, the next update would start where the last one left off. The result was continuous, non-skipping audio, while letting us arbitrarily alter the frame size and sampling rate. This became important later when adding polyphony.

Wavetable MMO

The Wavetable Multi Mode Oscillator (WMMO) is similar to the MMVCO, except instead of calculating values for the waveforms in real time, it generates a look-up table on initialization and then grabs the values for the selected waveform from that look-up table.

The advantage if the WMMO is that it requires significantly processor time. The disadvantage lies in the inherent inaccuracies that come into play when you try to modulate a Wavetable value.

Our Implementation adds the same basic waveforms to what our MMVCO offers, in addition to the following:

- Inverse sine
- Triangle
- PWM square (pulse width modulated square)

Wave-shaper VCO

This was our concept of a "make your own signal" generator. Essentially, the user can set the amplitude of the signal within each period. We provide four "poles" that let the user do that. Just like the other oscillators, the frequency and overall amplitude are independently adjustable as well.

Moog VCF

This filter is a digital equivalent to the Moog Voltage Controlled Filter, aka, the Moog Ladder Filter (MVCF).

Our MVCF implementation allows the user to select the following modes:

- low-pass
- high-pass
- band-pass

The z-domain transfer function of the MVCF is the following:

- G(z) = ((p+1)(z+0.3)/1.3(z+p))^4

It should be noted that this filter is a resonator. This is part of what has made it so popular in the audio world. Altering the Q of the filter can yield ringing and emphasis effects at the cutoff frequency of the filter. As a matter of fact, setting the Q high enough can result in destabilizing the filter altogether and creating an oscillator, oscillating at the cutoff frequency. In the analog world, a reset could only be achieved by bleeding out the charge from the capacitive and inductive elements of the circuit. In the digital world, we just reset the filter if it's gone unstable.

We were surprised at how well the algorithm[5] we used mimics the real world MVCF.

Butterworth VCF

This is the digital equivalent to a 4 pole RC passive filter. We generate Butterworth second order sections and then transform them from the s domain into the z domain to arrive at our digital filter coefficients. We pretty much followed the guidelines set out at the beginning of 18-551 as well as an algorithm we found on the web[6].

[5] Algorithm from http://www.musicdsp.org
[6] Algorithm from http://harmonycentral.com

Our major modification of the algorithm was to separate the initialization, parameter update, and frame calculation sections in order to minimize processor load intensity. As is intuitively obvious, you only want to initialize each instance of the filter once, calculate the filter coefficients only when the user has requested a change, and calculate a new frame of samples only when the last frame has been read. Needless to say, the last of these operations occurs with the highest duty cycle, not separating the former two from the latter saves a lot of processing time.

This IIR filter behaves as a LPF, as long as the resonance is kept small. As the Resonance value increases, the filter "rings" more near the cutoff frequency, causing a band pass effect as a result of the added peaking

ADSR VCEG

The ADSR VCEG is one of the most used and fundamental modules in sound and music synthesis. We abbreviate the term above to ADSR. It stands for "Attack Delay Sustain Release".

The ADSR is a time sensitive voltage controlled amplifier. The ADSR is what provides the amplitude shaping that lets us create the following types of effects:

- Swell
- Fade
- Percussive attack
- Doppler effect

The way it works is that we break up every "note on" period into 4 distinct regimes.

- First attack
- Then decay
- Then sustain
- And lastly, release (note off)

The algorithm looks something like this:

If(in attack) output = a*input
If(in decay) output = d*input
If(in sustain) output = s*input
If(in release) output = r*input

But not exactly, because we process frame by frame instead of sample by sample – However, that's the idea. In reality, we have to keep track of the time that's passed since the note was triggered, the amplitude of the previous sample, and whether the "note off" event has arrived. This is done in a manner similar to the way we keep track of our sample position in the MMVCO.

Echo, Delay

The delay and echo modules delay the input signal and have built in configurable feedback.

Delay:
Our delay module can mix the input and delayed signal into the output, as well as feedback the output into the input. We represent the original signal and the delayed signal as “Dry” and “Wet” respectively. Our algorithm looks like this:

```
delayLineOutput = *readPtr;
output[i] = dryLevel * input[i] + wetLevel * delayLineOutput;
*readPtr = input[i] + feedbackGain * delayLineOutput;
```

The delay time and read head position can be modified by input CV's. The read head is the place in the buffer the output sample is taken from, relative to the write head. You’ll note that just as in the MVCF and the MMVCO, we have to know the results of the previous sample to computer the current one. Depending on the delay time, we may have to know the value of a sample as many as 44,100 positions back, given a maximum delay of 1 second!

The delay module can be used as the base of a number of effects, such as phasers, flangers and complex echoes.

If the output is fed back into the input, you get a similar effect to the echo, but you can add cool effects by routing the signal back through a low-pass filter (for example).

Echo:

This is a generic, no frills echo where you can control just the feedback and the delay. It is essentially a stripped down version of our delay that runs a lot faster at the cost of less functionality.

Mixer

Our mixer is essentially a four channel adder, with scalable inputs. The output is simply the sum of the input-channel gain product.

Out = K1*input1 + K2*input2 + K3*input3 + K4*input4

GUI

GUI Development

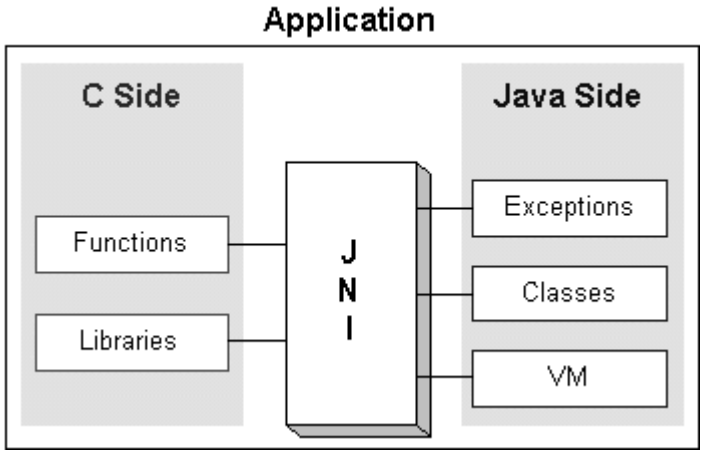
In order to allow users to efficiently control the various features of an audio synthesizer, an intuitive interface is essential. Thus a well designed graphical user interface is more important in our project than a typical signal processing project in this class. Several factors influenced the GUI development process. First, the GUI should have the same easy to use look and feel of current software synthesizer interfaces.

Secondly, the GUI must have a method of interacting with the underlying system C code to pass the control parameters as input and routing of the order of the various modules.

Our GUI module is composed of various smaller modules to display each of the control windows. RadioButtonListener and SliderListener Java classes were utilized to detect changes in the GUI components. A desktop pane is used to place the GUI component buttons. Upon clicking a GUI component button, an instance of the corresponding GUI window is instantiated. We have developed seven GUI windows: ADSR envelope, Butterworth Filter, Delay, Echo, Mixer, Moog Filter, and Wave Generator.

We have chosen to use the applet classes in the Java language to implement our GUI. The biggest challenge was to ensure that the graphical applet windows and controls within them created in Java are able to pass the parameters and drive the system module written in C. The first thought was to utilize the Java Native Interface (JNI) to have Java programs implement various methods in native code to drive C programs. This involves writing java program shells to create a dynamic link library file to call the equivalent C implementation of a particular function.

The JNI Bridge Between JAVA and C



In addition, we have chosen to use Inter Process Communication (IPC) to allow information flow between C programs and Java programs. IPC is a system that lets two or more processes communicate with each other. The two ways to achieve IPC are shared memory and message passing. We have utilized an IPC pipe object to communicate between the C and Java programs via message passing. The pipe is first open in a C program, waiting for a connection. The driver program in the GUI implementation, MASH.java, initiates a pipe object and allows data output to be sent via the pipe. Each GUI component consists of a moduleID, instanceID, parameterID and a value field. A byte array is declared to obtain the various states of the sliders and radio button groups in the GUI window. This byte array is later passed through the C program through the pipe object to allow communication with the other modules of this project.

Performance Analysis

Areas	Code Size	Incl. Average	
computeOscSamples	1300	13677	cycles
computeEnvelope	824	1991372	cycles
processParameters	1120	3848	cycles
computeSamples	600	7429395	cycles
TOTAL Frame Cycles		7429395	cycles
TOTAL Frame Time		0.052005765	seconds
TOTAL Sample Time		1.02E-04	seconds
AVAILABLE Sample Time		1.25E-04	seconds
TOTAL Head Room		2.34E-05	seconds

Above is a performance analysis summary of one of our configurations. As you can see, we meet our real time deadline here, so the audio doesn't skip. Altering polyphony, module number, module type, and sampling rate all have significant impacts on performance.

We found that through using Wavetable based oscillators, that we saved a lot of time, however, we decided to show the worst case scenario here, to demonstrate our ability to run a fully featured analog synthesizer emulator in real time.

A couple words on the profiler: It's performance was erratic at best, often summing the processes going on in multiple functions all under one heading, and often the wrong one. Restarting the EVM often yielded different profiler numbers. In general, The Profiler was accurate for profiling SMALL functions, that didn't call other functions.

Prior Art

Algorithms:

18-551
There have been no projects in 551 that attempted to implement a synthesizer.
A couple of projects implement tone generators and rudimentary midi parsers.
This code was not real time compatible and therefore unusable.
N
Elsewhere
http://www.firstpr.com.au/dsp/pink-noise/
All about noise generation
http://www.harmonycentral.comn
Source for basic filters oscillator algorithms
Examples of MIDI parsers– nothing real time
http://www.musicdsp.org
Source for basic filters oscillators
Analog filter algorithms
Most code either “pseudo-code”, or in LISP, C++

In general, we were able to find algorithms and java skeleton code (for the GUI). All the C programming and optimization of these algorithms had to be done on a module-by-module basis.

Our Final Demo

Our final demo showed off the working synthesizer and demonstrated the following features:

- Analog synthesizer modeling
- MIDI or Oscillator INPUT with AUDIO (via CODEC) Output
- The advantages of our modular design
- Our working intuitive GUI
- Real time performance (imperceptible latency) improvement over “softsynths”(latency greater than half a second)
- Our ability to make cool sounds as well as music

Conclusion

MASH is a significant step beyond current hardware synthesizers. Aside from arbitrary polyphony (only dependent on the speed of the DSP), MASH's combination of hardware real time capability and software based GUI ease of use platforms simply hasn't been done before. The closest matches are software synthesizers that, while inexpensive and easy to use, are useless in performance environments, and $4,000+ difficult to use monophonic digital hardware synthesizers.

One thing we did discover during this project is that to do real time audio successfully on the EVM, one has to both double buffer as well as talk via DMA to the CODEC. In the end, resolving these real time issues became the single greatest hurdle to overcome in implementing MASH—and overcome it we did.

Terms and Definitions

MMVCO	Multi Mode Voltage Controlled Oscillator
ADSR VCEG	Attack Delay Sustain Release Voltage Controlled Envelope Generator
VCF	Voltage Controlled Filter
IIRF	Infinite Impulse Response Filter
Polyphony	Multitimbral value
LPF	Low Pass Filter
HPF	High Pass Filter
BPF	Band Pass Filter

Acknowledgements:

Ed Neto – who essentially played the role of EPM for the entire project.