Modular Synthesizer

Andrew Muth, Michael Miller, Tejasvi Vishwanadha

Abstract

Traditional modular synthesizers operate entirely in the analog realm, routing independent components together through patch cables. Modules such as oscillators, filters, mixers, and sequencers allow for myriad sonic possibilities. Our modular synthesizer functions entirely in the digital domain, replacing patch bays with an internal ring buffer and analog controls with a serial interface.

Table of Contents

Abstract i
Table of Contents ii
List of Examples iii
List of Equations iv
List of Figures v
1. Overview 1
  1.2 System Overview 2
  1.3 Audio Data Considerations 3
2. Audio Units 5
    2.0.1 Arithmetic Sharing 5
  2.1 Oscillator 5
    2.1.1 Direct Digital Synthesis 6
    2.1.2 Testing and Debugging 6
  2.2 Filter 7
    2.2.2 Testing and Debugging 8
  Additionally, to test the *Filter* further, I created a brief Python script that pores over the outputted results from my simulated filter test bed (Appendix C - Code 1) 8
  2.3 Sequencer 9
  2.4 AC97 In and Out 9
  2.5 Sampler 10
  2.6 Mixer 11
  2.7 Delay 12
  2.8 Display 12
3. Command, Control, and Internal Routing 14
  3.1 Routing Overview 14
  3.2 Audio Data Routing 14
    3.2.1 Theory of Operation 15
    3.2.2 Output Drivers & Ring Cycling 16
    3.2.3 Input Receivers & Control Registers 17
    3.2.4 Testing and Debugging 18
  3.3 Control Signal Routing 19
    3.3.1 Theory of Operation 19
    3.3.2 Control Bus Implementation 19
    3.3.3 RS-232 UART System 20
    3.3.4 Serial Command Parser 20
4. Conclusion 21
Appendices 22
  Appendix A - Equations 22
  Appendix B - Examples 23
  Appendix C - Code 27

List of Examples

Example 1. Oscillator in Simulation 23
Example 2. High-Pass Filter in Simulation 23
Example 3. High-Pass Filter Reference 24
Example 4. Low-Pass Filter in Simulation 25
Example 5. Low-Pass Filter Reference 26

List of Equations

Equation 1. Biquad Transfer Function 22
Equation 2. Biquad Recurrence Relation 22
Equation 3. Low-Pass Filter Coefficients 22

List of Figures

Figure 1. Modular Synthesizer - Block Level Diagram 3
Figure 2. Delay Routing 12
Figure 3. Audio Router - Behavioral Goal 15
Figure 4. Ring Buffer - Ringing Behavior 15
Figure 5. Ring Buffer - Output System 16
Figure 6. Ring Buffer - Output Behavior 16
Figure 7. Ring Buffer - Input System 17
Figure 8. Ring Buffer - Input Behavior 18

1. Overview

A synthesizer is an electronic instrument that generates output tones by creating and combining signals generated by user-controlled inputs. A modular synthesizer is a synthesizer that produces its final output by routing independent synthesizer modules together. The modules, such as oscillators, filters, delays, and sequencers allow for myriad sonic possibilities. Traditional modular synthesizers, most famously produced by Moog, date back to the mid-1960s. These synthesizers worked entirely in the analog domain. Our modular synthesizer will internally function in the digital domain, mimicking the components of the original synthesizers and adding some that would not have been possible.

We were interested in making a modular synthesizer for multiple reasons. The highly modular scheme lends itself well to Verilog. We believed that each time we completed a module, it could be added to the synthesizer with relative ease and expand our final. We also planned to focus on the user interface, believing that it would be an interesting and fun problem to tackle. Of course, the fact that our project would generate cool music on the fly didn't hurt either.

Like their analog brethren, digital synthesizers – implemented in hardware or software – lend themselves well to a highly modular system topology with a simple data interface. In analog systems this is accomplished by implementing a number of distinct audio processing units that run in parallel while taking in and outputting a continuous non-quantized voltage waveform. Control of each processing module is generally done by a variety of switches, knobs, and sliders which provide signals to the analog circuitry within. Routing the audio data is accomplished by physical wires connecting input to output on each module, allowing arbitrary connection patterns including feedback loops and wide fan-in/out without needing any specialized hardware or system controller. In the case of our digital synthesizer the inherent division between each audio processing step worked well with the modular design paradigm enforced by Verilog. However, the twin issues of routing and audio processor control required I create a unique technical solution where traditionally human input, in the form of turning knobs or connecting cables, is the norm.

Routing digital audio signals within our FPGA presented a challenge because of the simple fact that we would no longer be able to treat each audio processing unit (APU) as a distinct block which the musician would manually connect to the next ; rather every possible routing combination would have to be achievable within the same synthesized design or the user would be stuck editing and reprogramming the synthesizer design each time they wished to change the connection layout between APUs. Similarly, the digital interface of the FPGA containing our synthesizer suggested we not implement a large set of analog knobs and switches but instead use a single control interface common to every APU. A single control module would then be responsible for communicating with the musician and providing the proper APU with its control signals based on the user's input.

The following sections of this paper look at the design methodology, tradeoffs and current implementation spec for the digital Modular Synthesizer system we built for 6.111. In particular they will focus on the Synthesizer at the system level as motivation for the audio data router and common control interface mentioned earlier, though we will also touch on the "multi-tier" abstraction concept we used to facilitate clean coding and efficient system integration. We will also discuss in detail the design specifics that went into each of the audio signal processing cores. Finally, we will look at how the engineering goals for this project changed over time from a output-oriented product to a systems

engineering-oriented synthesizer framework designed for scalability, modularity, and ease of implementation.

1.2 System Overview

As we mentioned earlier, the layout of the Modular Synthesizer we implemented relies heavily on a "tiered" paradigm that breaks up all of the necessary parts into distinct modules, those modules into sub-modules, and so on. This is true at both the concept level, where the synthesizer is made up of a number of distinct APUs, and at the hardware level where an APU is defined to be a wrapper for the network interface, control interface, glue logic, and digital sound processing (DSP) core. This abstraction was a useful design decision for two reasons: first, it took advantage of the inherent modularity present in Verilog HDL; and second, because it concentrated all of the actual digital design components into the outermost "leaves" of the Synthesizer's design tree.

For example, the control module we will look at later on is nothing more than a wrapper containing interconnections between a UART, a parser for the data from the UART, and a link to the next-highest level of the design. In the same manner the UART module is really just a wrapper containing interconnections between a serializer, de-serializer, and the next-highest level of design. It is only in the serializer and de-serializer modules that digital logic components can be found to actually implement the standard RS-232 protocol.

At the top-most level of abstraction the Modular Synthesizer design is based on four major component types which connect to the labkit peripheral hardware and each other:

1. Audio Modules (APUs)
2. Control Module (user input)
3. Debug Display Module (LEDs & character VFD)
4. Support Systems (digital clock manager, reset signal generation)

It also contains the wire nets which link the specific input and output pin names defined for the 6.111 Labkit to our modules' output and provide bus interconnections between each of the module types listed above. In every case these busses are passive wires which do not change the signal timing specs or provide storage. A block-level diagram illustrating the system is displayed below.

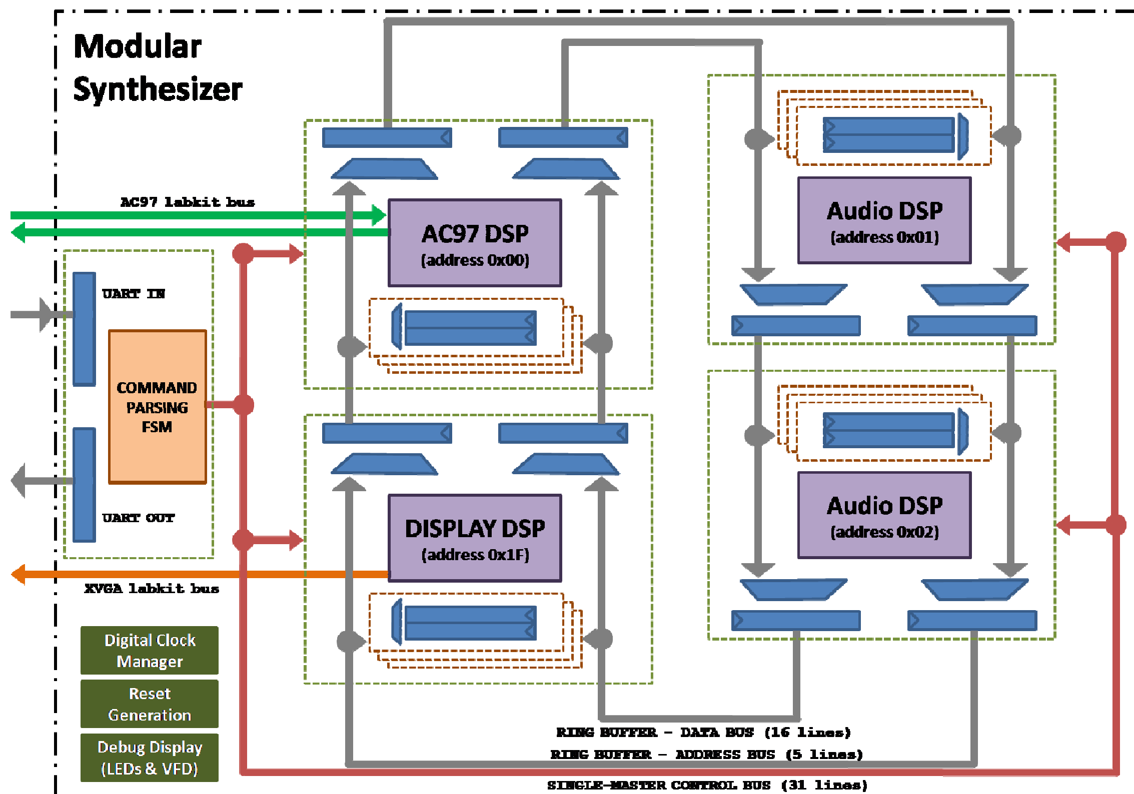


Figure 1. Modular Synthesizer - Block Level Diagram

This block diagram shows a representative instantiation of the Modular Synthesizer including two APUs listed as “Audio DSP: address 0x01” and “Audio DSP: address 0x02”; however it should be noted that the same framework and bus architecture can be extended up to greater than a hundred distinct APUs – the only limiting factor is fitting all of the devices into the FPGA without conflicting arithmetic system requirements. Noteworthy components include the audio data network (here shown in gray), the common control interface (red) and parser (orange), and a number of DSP audio processing cores (purple). Each of these components is described in detail later in this paper.

1.3 Audio Data Considerations

As we noted earlier, one of the keys to achieving a truly modular digital synthesizer is to define a common audio data standard whose bandwidth and amplitude space exceed that needed to create any sound. In this implementation both the source and sink of the audio path is an LM4550 AC97 codec peripheral located on the 6.111 labkit. Access to the chip is done using an edited version of the AC97 audio library interface provided in Lab 4 which provides for 16-bit PCM audio data output sampled at 48 kHz while being driven by a faster 64.8 MHz system clock.

The reasoning behind the choice to use the larger 16-bit audio data source follows directly from the nature of audio information itself. To listen to sound, it must be put into a speaker which can take only a single time-varying data channel, typically a continuous analog voltage. In the case of the AC97 decoder used in this synthesizer, I take advantage of the fact that sound can be quantized into distinct intensity levels based on the audio capabilities of the human ear. For the purposes of this implementation 16 bits of precision in the sound level, or 65,536 distinct states, were enough to reproduce almost any incoming audio stream without fidelity issues. Similarly, the range of human

hearing is limited to a narrow frequency band of approximately 20 Hz to 22 kHz, meaning that the Nyquist rate necessary to completely reproduce any sound within human perception requires a quantized sampling period of 44 kHz. The 48 kHz sampling frequency of the AC97 peripheral codec ensured that even in the worst case (very high input frequency and poor low-pass filter cutoff) one would not expect any high frequency components above the audible range to affect the reproduced sound.

Given the onboard availability of this high-quality signal and the fact that many DSP operations can occur in parallel, I chose to use the full 16-bit PCM audio stream sampled at 48 kHz without downsampling for a net data rate of 768kbps. This choice gives the synthesizer hardware an audio bitstream which is detailed enough to be downsampled to 12 or 8 bits per sample if necessary for long storage times in a sampler, yet does not require unmanageably large amounts of block RAM or ZBT implementation to work with.

2. Audio Units

At the start of this project, Mike had significant experience working with traditional audio synthesizers and digital signal processing while Andrew's area of interest was on designing a novel control interface using hardware components. Given this we divided the project up into two major components – the interface/control section which Andrew would focus on, and the more complex DSP audio section which Mike would focus on. Meanwhile, Teja would work with both of us when necessary while also attempting some of the simpler audio modules.

In the end, Mike ended up implementing and demonstrating an oscillator, sequencer, and filter DSP core while also developing an arithmetic sharing system and floating fixed-point math standard that would form the base of the Synthesizer. Teja ended up working on a variety of audio and video modules including a sampler, mixer, audio delay, and waveform display. He also extended the AC97 library from Lab 4 to provide 16-bit PCM data driven by a 64.8 MHz clock and completed wrapper files for DSP cores using the communication templates written by Andrew. Andrew ended up having to develop the internal audio routing network and input/output register memory system used by the DSP cores to communicate. He also designed the control bus architecture, found a workable serial port module, wrote the control parsing FSM which drives the control bus, and developed the wrapper coding paradigm used throughout the project.

2.0.1 Arithmetic Sharing

Since many of the modules require complex arithmetical expressions often involving multiple divisions, Mike created a scheme to share these expensive operations with every module. This scheme was successfully implemented within the *Oscillator* and the *Filter*, who perform four divides each ready pulse using the same global instance, effectively conserving sparse resources in exchange for using more clock cycles.

Within each module, most internal modules are executed sequentially. Ready levels are created based upon the expected duration of a module, a parameter. Modules sharing the divider agree to a sharing contract: as long as the ready signal remains high, that module may assert its divisor and dividend arguments. Otherwise, it must pass zero. The parent module then combines these signals into one using a bitwise or, and passes it to its dividend and divisor outputs which makes its way eventually to the divider itself. A simple extension to this technique is to remove the duration element and make each module assert a done signal as soon as it is done using the divider. By attaching the done signal of one module to the ready signal of another, each module can trigger the next to start as soon as it is finished.

2.1 Oscillator

The *Oscillator* module is a versatile signal generator that employs direct digital synthesis (DDS) to create a variety of periodic waveforms. It can currently produce six different types of waves: pulse and square; ramp and saw; triangle; and sine. The user simply specifies a `frequency` and a wave `type` (as well as a pulse width in the case of the pulse wave). The output has a fixed, default gain and the maximum amplitude is approximately the same across all signal types (-3 dB).

The *Oscillator* is written with generality in mind and is programmatically flexible. Wire widths dependent on the width of the audio or control parameters are declared in terms of `N`, a globally defined variable currently set to 16.

2.1.1 Direct Digital Synthesis

A naïve approach to creating a sine wave using a lookup table would use a counter to increment over the table. By doubling the increment on the counter, we double the frequency of the synthesized signal. However, if we want to decrease the frequency, we would need to decrease the increment below one.

Instead of a simple counter, direct digital synthesis replicates a dataset at a desired frequency with high resolution using a fixed-point `phase_accumulator`. This process keeps fractional bits of precision so that we can decrease the signal below the frequency represented in the lookup table. If we wanted to simply recreate the table from before as with the simple counter, the increment would be one represented in the same fixed-point scale as the phase accumulator.

In the final version of the *Oscillator* module, I use two generic *DDS* modules routed to different datasets. One *DDS* references a 512x16-bit sine lookup table to generate a sine wave. The table exploits sine's two points of symmetry around the vertical ($\pi/2 \rightarrow \pi$ is symmetric with $0 \rightarrow \pi/2$) and the horizontal ($\pi \rightarrow 2\pi$ mirrors $0 \rightarrow \pi$), effectively creating a 2048x16-bit table.

The second *DDS* is used to generate a ramp. In this case, a 512x16-bit memory is wholly unnecessary. Instead, the module mimics the table the *DDS* module expects but represents its data using simple addition based on the lookup address.

The abstraction between the *DDS* module and its data source leads to interesting future possibilities. It allows for the creation of a limitless number of periodic waveforms by simply changing the dataset. One could conceivably store a large number of complex datasets in memory and selectively call up these more interesting tones. As well, I am curious about the results of combining direct digital synthesis with sampling, with the samples starting and stopping at zero-crossings to hopefully smooth the looping. By generalizing the direct digital synthesis concept into an abstract dataset and synthesis module, experimenting with these new possibilities is convenient and straightforward.

Some wave types do not need to be generated, but can instead be realized from an already-created signal such as a sine wave or ramp. The pulse, square, saw, and triangle waves are created in this manner. For instance, the high-order bit of the sine wave represents a square wave with the correct frequency (although not the right amplitude). The pulse wave is created by simply setting the square wave low once `width` ready cycles have passed. The saw wave is simply the negative of the ramp wave. The triangle wave takes the absolute value of the ramp and shifts and scales it into place.

2.1.2 Testing and Debugging

I tested the accuracy of the *Oscillator*'s output in three ways: by simulation in ModelSim (Example 1); by viewing the analog output on the logic analyzer; and by ear. For both Modelsim and the logic analyzer, I successfully measured and verified the frequency of the output signal. I cross-referenced

the signals with each other and the reference 750hz sine tone provided in Lab 4 using all three methods. My sine output proved to be essentially identical, and all the other waveform `types` similarly matched in frequency, though as was expected, their tone qualities differed wildly.

Once I verified my *Oscillator*'s outputs, I continued to use it for testing other modules. I tested the Sequencer eventually by using it to drive the *Oscillator*'s `frequency` input, and it was during that test that I heard poor performance in the upper frequencies. This led me to redesign the oscillator into its final version.

My original implementation used *DDS* to generate only the sine wave. The pulse and ramp were both generated on the `ready` signal, using half the wave's period as a guide for when to flip in the case of the square wave, or as the first part of calculating the step size for the ramp. This method resulted in "pure" waveforms whose periods were an integer multiple of 48khz clock cycles. This previous method which generated the values on the 48khz cycles rather than through sampling led to poor performance, particularly in the high frequencies. As the desired frequency grew, tones with similarly high frequencies would quantize to the same exact signal.

DDS, however generates signals that are sampled from an underlying representation. The *DDS* sine wave, for instance, is not exactly a sine for most frequencies, although the representation is perfectly sinusoidal. To achieve successful digital playback at varying frequencies, the wave takes on a longer and more complex overall periodicity as the sampling point cycles over the representation. This allows us to approximate waves that are not integer multiples of the 48khz clock cycles, and hence avoid two waves with different frequencies generating the same quantized signal.

I ran into an issue involving noise as `frequency` got higher. While I was unable to fix the problem because of time constraints, it likely had something to do with the precision of the phase accumulator in the *DDS* module.

2.2 Filter

The *Filter* module is intended to act as single-pole audio equalizer unit. It implements a coefficient generator and a second-order biquad-based infinite impulse response (IIR) filter. IIR filters like the Butterworth filter are particularly suited for audio processing tasks, as they closely mimic their analog predecessors and counterparts. Butterworth filters in particular are good for audio because they have maximal flatness in the pass-band region.

Unlike finite impulse response (FIR) filters which require the generation of many constants that define an impulse for convolution, a biquad (Equation 1, Equation 2) requires only six constants. As well, while biquads are only second-order filters, they can be cascaded together in series to create higher order systems with a greater degree of numeric stability than a higher order transfer function. This property particularly lends itself to our modular framework for the synthesizer, allowing the user to string together *Filter*s to create a many-pole equalizer from this basic building block.

The *Filter* uses fixed-point arithmetic to provide an accurate rendering of the coefficients and the sum-of-products filter output (Equation 2). By converting the signed input signal to Q15.2[1] and outputting 18-bit signed coefficients formatted as Q1.15, I used the 18-bit built-in multipliers to get good performance coupled with good accuracy.

The *Filter Coefficients* sub-module generates the six coefficients for the biquad. These coefficients vary depending on the type of filter desired. Currently, low-pass, high-pass, band-pass, and notch filtering are implemented but the design is intentionally extensible. The math to generate the constants is straightforward, with the exception of a single divide used to generate the angular frequency `ω0`. The coefficients themselves are generated using sine and cosine table lookups and simple arithmetic (Equation 3).

Once the filter coefficients are available, the *Filter Scale* module divides each coefficient by `a0`. While every other divide in the synthesizer is routed to a single, global division module, the *Filter Scale* module uses its own divide more suited to its needs. I generated this divide specifically to do the 18-bit divisions. As well, it is fully pipelined to make the four divisions in this module efficient.

With the scaled coefficients in hand, the *Filter Accumulator* performs a sum-of-products with five terms. These terms are a product of the coefficients with the previous three samples of input or feedback of the last two samples of output (Equation 2). The module uses a single 18-bit by 18-bit built-in signed multiplier to sequentially multiply each of the five coefficient-data pairs and add them to the accumulator.

2.2.2 Testing and Debugging

While I ran out of time to sufficiently debug the *Filter* on the Labkit, the simulation results in ModelSim were quite promising. By comparing my results on various signals from the oscillator with a known result, I was able to confirm some details of my implementation's functionality. In particular, Paul Falstad's Digital Filters Java Applet proved invaluable in this endeavor[2]. Examples Example 2, Example 3, Example 4, and Example 5 illustrate these comparisons and show what appears to be a very functional low-pass and high-pass filter.

I had a lot of difficulty in particular with the fixed-point mathematics which was new to me. Addition and subtraction made intuitive sense, but division in particular, once abstracted even further through the CoreGen divider module, became very difficult for me to follow and comprehend. Ample testing was and is a necessity with complicated fixed-point arithmetic.

Additionally, to test the *Filter* further, I created a brief Python script that pores over the outputted results from my simulated filter test bed (Appendix C - Code

[1] This is notation I found useful for describing fixed-point arithmetic (http://en.wikipedia.org/wiki/Fixed-point_arithmetic): Q(#integer bits).(#floating bits). The signed bit is implied if present, and not counted.
[2] http://www.falstad.com/dfilter/index.html

Code 1 1)

The script checks the data for consistency by comparing its calculations to the calculations performed by my computer, my standard of correctness. While my filter has yet to past this rigorous test, it revealed many bugs and would surely have led me to a correct solution given more time.

2.3 Sequencer

The *Sequencer* produces a sequence of user-programmable numerical values by iteration. It acts as a sort of sequential-read random-write memory, outputting new values every `ready` pulse and writing new values whenever its `write` pin goes high. Internally, the *Sequencer* uses an array of *N*-bit wide registers to store values.

For added flexibility, the *Sequencer* includes a notion of end behavior that controls how to continue once it has output the last stored value. Current options include simply stopping, looping, or reversing. The user can also control the speed at which new values are created. The number of the values in the sequencer is parameterized. As such, the user can specify an upper-bound on the size of any sequencer instance at compile time to conserve resources.

As well, I created a file of predefined pitches ranging (in scientific pitch notation) from A3 all the way up to A6. Further pitches could be created by simply shifting an available value to raise or lower it an octave.

The design and mechanics of the sequencer are rather simple, and as a result, testing was straightforward. I simply verified that the value was latching to the correct numerical output every `ready` pulse. Unlike the filter whose low-passed saw wave might be slightly off yet look fine, there is no subjectivity involved in the correctness of the sequencer.

The *Sequencer*, once implemented, became very useful as a test tool. It allowed me to send the *Oscillator* a variety of frequencies in a repeatable test-worthy fashion much as I might write into a ModelSim test.

I found many times that while I could create a large number of tests, it was only useful when I could verify the answer with some knowledge on my part. It is hard to visualize a sine wave as it changes frequency for correctness, but it is easy to do so aurally. With my include file that mapped pitches to frequencies, I was able to create tests such that I knew what it should sound like. The combination of the *Sequencer* and the *Oscillator* became invaluable test tools for testing the Filter as well.

2.4 AC97 In and Out

The *AC97InOut* module is perhaps the simplest of the modules I created. It is essentially identical to the *Lab4audio* module used in Lab 4, but the input and output audio data have been widened to 16 bits to match CD quality audio.

2.5 Sampler

This module was actually inspired by watching a few videos by two-time UK Beatboxing champion, Beardyman. Beardyman uses two KORG KP3 KAOSS pads on stage to sample his own voice and play each back with different audio effects. My goal with the *Sampler* was to create module that stored a small sample of input data and played it back when the user told it to. The main difference in functionality between Lab 4 and the sampler is that the latter has 3 states – recording, playback, and silence – while the former only toggled between the first two states.

While functionally similar to Lab 4, two fundamental design decisions made creating the *Sampler* significantly more difficult. In lab4, we used 8-bit audio and down sampled from 48kHz to 6kHz. By storing this information in a 64Kx8 memory, we managed to store about 10 seconds of audio data.

Since we were using 16 bit data, I would need to store at least twice as much information. In addition, I didn't want to downsample because I didn't want to lose information. Therefore, maintaining the length of the sample would take eight times as much space. Unfortunately, since a 64Kx8 memory uses approximately 20% of the BRAM on the lab kit, recording a full 10 seconds of 16bit audio at 48kHz would take 320% of the available BRAM.

The first version of the sampler, which eventually became the *Big Sampler*, only sacrificed sample length to fit on the lab kit. The first iteration used a 128Kx16 memory, the largest size I could fully address with a 16-bit width, and using 80% of BRAM on the lab kit. The sample length was a little over 2 seconds. I decided to cut the memory down to 64K rows – halving the BRAM consumption at the expense of one second of audio data – in the final version of *Big Sampler*.

While 40% consumption was significantly better, I wanted to find a better way to store samples. I considered two routes – storing less audio data or storing the audio data on ZBTs. The latter was my initial choice. Each ZBT can store about 7 times as much data as all the BRAM on the lab kit. The ZBTs are each 36 bits wide, so the simplest way to maintain an easy addressing scheme is storing two audio samples per row and leaving 4 bits empty. Coincidentally, accessing the ZBT takes two clock cycles, so I'd have two samples ready every time I wanted to write a row. After doing a bit of math, I determined could store a single 16 second sample on a single ZBT.

While this seems wonderful at first, I realized that most of the samples I'd want to store would be considerably smaller – probably topping out at five seconds. Those could be broken up into smaller bits that are even smaller by isolating sounds. While 2 samplers with huge samples would be better than no samplers, I felt they would be wasting space. The only option seemed to be horizontally partitioning the ZBTs into 3 or 4 smaller chunks. At that point, however, each sampler on the kit would have to be aware of all other sampler and use time sharing to access all the necessary bits. The system didn't seem to scale well and broke out wrapper abstraction so I concluded that puzzling it out would be more trouble that its worth.

I decided instead to explore what I could do to cut down the size of the internal memory but still maintain audio quality. After a few quick experiments, I realized that the top 8 bits of audio data sampled at 48kHz sounded surprisingly good during play back. I also took advantage of the realization that specific instrument samples could be under a second long. The final result was *Small Sampler*,

which samples only the top 8 bits of incoming audio data for about half a second. It uses a 32Kx8 memory, one quarter the size of the larger version.

If I were to extend the sampler further, I would add the ability to add a delay for playback. The delay would be relatively trivial to implement. A simple counter that increments with each ready pulse would keep track of the number of cycles to delay before returning to the beginning of the memory and playing the data back.

Such a delay would save valuable memory from being wasted on silence. For example, the user could create three small samples for a snare drum, base drum, and hi-hat and space them out to create the "standard 4/4 rock beat" instead of sampling 2 full beats worth (at least 2 seconds of audio data). In fact, by using the sequencer to alternate delay values the user could alternate seamlessly between the standard beat and "double time".

2.6 Mixer

The *Mixer* module we originally proposed was to be a parameterized N-to-1 mixer module. However, since each input would be accompanied by a level value, the arithmetic for supporting N-to-1 soon got very tricky. Therefore, I focused instead on a 2-to-1 mixer that could be chained together for N-to-1 mixing.

Besides the standard clock, ready, and reset, the *Mixer* module takes four standard inputs and one parameterized input. It has only one output, the mixed signal. The `in1` and `in2` inputs are audio inputs while `level1` and `level2` tell the mixer how to combine the two streams. The `decimal` parameter tells how many bits of the incoming level values are fractional. If `decimal` is 0, the levels are whole numbers (essentially gain). If `decimal` is 15, the level is a fraction less than one. In order to average the signals, both `levels` and the `decimal` value are 1, indicating the output value would be the sum of half of each incoming signal .

The original version of the *Mixer* (before it became a 2-to-1 mixer) assumed that incoming `level` values were unsigned whole numbers. The module would then require summing the product of an unknown number of signals. In order to make sure the module did not use too many resources, the module did a single signal by level product each clock cycle and added it to the sum stored 40-bit register. I chose 40 bits because I assumed that multiplication of two 16-bit values provided a 32-bit product and that there would be at most 8 signals being mixed at one time.

When I decided to make a 2-to-1 mixer, I shrunk the register down to 34-bits. Signed multiplication of two 16-bit values could create a 33-bit value. Since there were only two signals, I knew I 34 bits was enough to contain the sum. I also retained the single-product per clock system so that Mike's modules would hopefully not contend with mine for resources.

The trickiest part was adding support for signed fractional values. Without fractions, I'd always check the top 16 bits of the product sum for non-zero value. If the value wasn't zero, I knew that signal had clipped the upper limit and the module output the highest 16-bit value. If the value of the upper bits were zero, I would know the signal did not clip and the module output the bottom 16 bits of the sum. By adding fractional values, I had to account for the signed bit and fractional bits, meaning I had to check a different set of bits for clipping.

If I represent an n-bit signed decimal value as A(a,b), where b bits correspond to fractional bits and a+b = n-1, the product A(a1,b1) * A(a2,b2) can be represented as A(a1+a2+1, b1+b2). I know that b2 is zero, since the audio data is a whole number, so the final result is A(a1+a2+1,b1). The value of b1 is the parameter `decimal`, so I knew how many bits to drop from the bottom of the product sum. The next 16 bits were valid audio data, and any higher bits were either sign information or excess indicating clipping. I tested the *Mixer* extensively in ModelSim while adding `decimal` support and decided to assume that if the "higher" bits were all the same (either all 0 or all 1) they were sign extensions and thus the audio values in the lower bits were valid. If the bits were not the same, I decided the bits were probably excess values and thus provided a clipped maximum signal.

2.7 Delay

The true purpose of adding decimal support to the *Mixer* module came from my using it twice in a *Delay* module. The *Delay* module basically stores a certain amount of and input stream in memory and then mixes this buffered value with the input stream. The module also supports feedback, mixing the output of the previous mixer with the incoming value before storage.

The `delay` input determines how many cycles of input data the module should store. This value was also the number of ready cycles to hold input data for before mixing it back in with the current input data.

The `wetdry`, `gain`, and `feedback` inputs were basically `level` values for the two *Mixer*s used in *Delay*. A wet signal would just be the delayed signal while a dry signal would just be the current input. I assumed the `wetdry` signal to be a fully fractional signed value between -1 and 1. When zero, the signal is dry and when the absolute value is 1 the signal is completely wet.

The product of the audio data and the `wetdry` input is mixed with the output of the 2nd mixer to determine the output of the module. The levels for this *Mixer* are the `gain` input, applied to the product, and the inverse of the `gain`, applied to the other *Mixer*'s output.

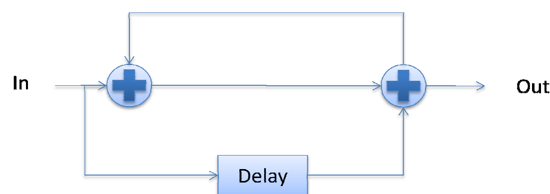


Figure 2. Delay Routing

The other *Mixer* handles feedback, mixing the input and the output of the module. The level for the output is `feedback` while the input has a level of 1-`feedback`.

I did not finish developing the *Delay* module for this project, but believe that my current implementation is pretty close to working. I am not sure that I am using the correct address for the memory I created, and would probably convert the logic to make use of a 16-bit wide register array, allowing me to select different addresses for reading and writing each ready signal.

2.8 Display

The original purpose of the *Display* module was to provide a visual component to the project. We had wanted a way to display the waveforms from any module—both for our own debugging purposes and for users to know what they are doing. Over time, however, the debugging purpose became less useful, allowing me have a bit more fun playing with the visualization aspect.

We decided early on that we wanted to turn the display counter-clockwise 90 degrees. In this configuration, we have 1024 pixels worth of vertical space, which would allow us to display signed values between positive and negative 512. Turning clockwise also ensures that the monitor's vertical scan progresses from right to left.

While the module receives 16 bits of input, only the top 10 bits are ever used. Values are sampled from the input data every 32 ready pulses, and added to a 10x768 array of values. When pixels are drawn to the screen, the pixels on each row of the screen check against the value in the corresponding row of the array. The value of the row is shifted upward by 512 to fit between 0 and 1024. For each row, pixels whose horizontal coordinate lies between 512 and shifted array value are colored.

The coloring scheme is also determined from the 10 bits stored in the array. *Display* was meant to be run on Andrew's Nexsys II kit. His kit uses only expects 8 bits of color information, with 3 for red, 3 for green, and 2 for blue. As a result, I chose to make the output `pixel` 8 bits wide. The 6.111 Lab Kit uses 8 bits per color. I decided to break up the `pixel` into the 3-3-2 format for different red, green, and blue values and then repeat each value until it was 8 bits long. In this way, the color of each row also corresponds to the value of the incoming data.

If I were to improve the module, I would like to have found a way to sync the screen and updates so the waveform would have a smooth continuous motion across the screen. In addition, downsampling makes the output image rather jagged and the output values could have used filtering (much like Lab 4).

3. Command, Control, and Internal Routing

3.1 Routing Overview

As was mentioned in the System Overview, there are two signal requirements a digital Modular Synthesizer will require which need a different implementation than one would find in an analog synth: control and audio data. While seemingly straightforward to create, the little complexities of both of these data routing problems ended up costing myself and the project in general several weeks of planning, coding, and debugging. However, once the bus architectures I designed were fully functional they allowed Mike and Teja's DSP components to be simply "dropped" into a wrapper template I wrote to encapsulate the audio processing and network components into a single audio module (APU).

3.2 Audio Data Routing

The Audio Data Routing system needed by the Modular Synthesizer had two major hurdles to overcome before it could be deployed, namely flexibility and scale. The need for flexibility represented the desire for a user to be able to change the pathway taken by audio data between the source (*i.e.,* the AC97 input, tone generator, or sampler memory) through a variety of audio modules towards the AC97 output.

At first glance the simple solution to this problem is to simply connect each module to all of the others using individual data busses. With that in place and a set of multiplexers used as selectors, it would be possible to connect any arbitrary data pathway at runtime. However, this concept suffers from the second hurdle: scale. If each APU instantiated within the system can talk to each of the others, the number of wires grows with order $n!$, where n is the number of APUs built. If the unidirectionality of Verilog nets is taken into account and modules can talk to themselves, this growth order rises to n^n. The implication of this is that unless the number of instantiated modules is **very** low, the total number of direct interconnects quickly grows outside the bounds an FPGA can handle.

The next iteration of my Audio Data Routing system was to have each APU output and input a standard 16-bit serial signal updated at 48 kHz synchronous with the new sample `sync` signal. These serial I/O pins would then be connected to hardware pins located on the 6.111 labkit, allowing the human user to simply plug wires between the appropriate inputs and outputs to form connections.

While this approach may seem feasible (after all, it is only a digitized version of the analog patch panel used on other synthesizers), it was suggested to me by Professor Terman et al. to look into an internal "virtual patch panel" based on a continuously-looping network with a ring topology. This approach, which I would later implement as the Audio Ring Buffer, would avoid all of the noise and potentially intermittent errors associated with using hardware connections to drive high-bandwidth digital connections.

3.2.1 Theory of Operation

In a nutshell, the Audio Ring Buffer is designed to give each APU access to an addressed version of every other APU's data. It also has to meet this requirement quickly in comparison to the 48 kHz `sync` signal provided by the AC97, since this signal is essentially our synthesizer's metronome.

The following diagram illustrates the behavioral goal I had in place for a single module while designing the Audio Ring Buffer. The key realization I had while working this timing specification out was the fundamental truth that any audio DSP operation would take a non-zero number of clock cycles between the time that its new input is available on its input and its output has the proper result. Given that the system clock `CLK` is many times faster than the `SYNC` signal, as long as the DSP can ensure that its output is valid by the time the next SYNC signal comes along this only forces a 1-`SYNC` latency from audio input to output (illustrated by the red arrows).

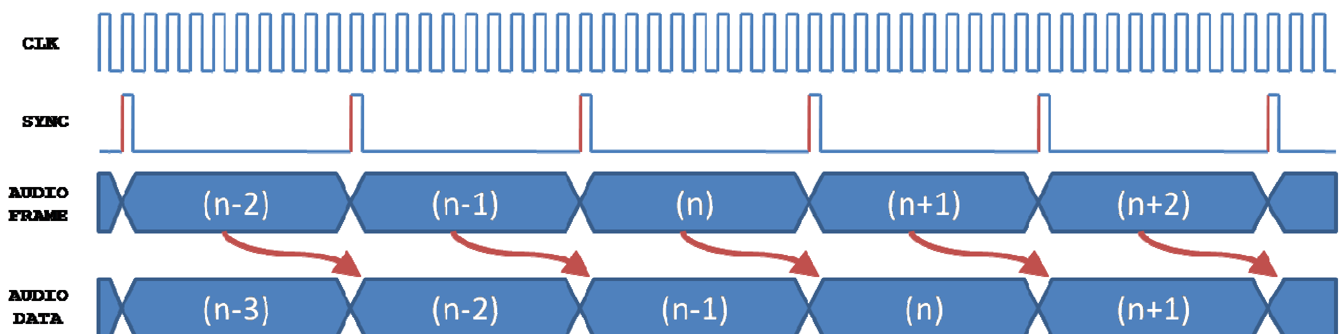


Figure 3. Audio Router - Behavioral Goal

From here I realized that if all of the APUs were connected linearly in a circular fashion and assigned a unique address, it would be possible to "rotate" their data value around the bus by inserting clocked registers between each audio module. If clocked on the system clock, these registers would effectively pipeline the bus and cause the data to loop around the ring bus once every *n* cycles for *n* modules on the bus. Combined with an address bus also clocked on the system clock, this idea yielded the following functional spec:

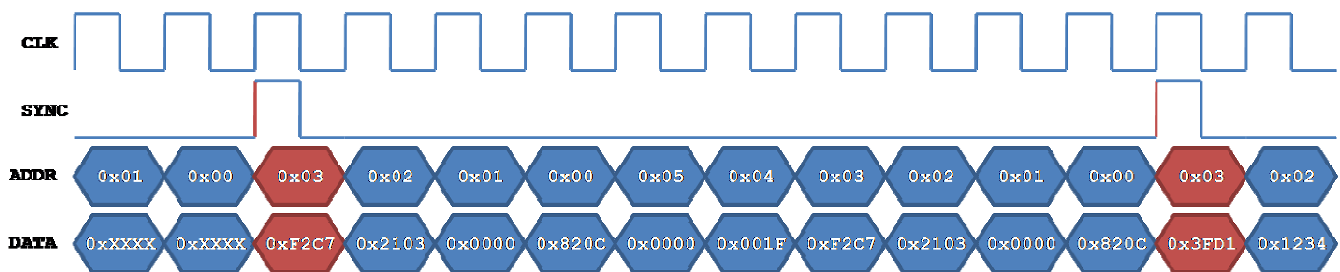


Figure 4. Ring Buffer - Ringing Behavior

Figure 4 also illustrates (on the red clock cycle) the fact that data will have to be injected onto the bus in order for it to be useful. As we saw in figure 3, if we require each DSP core to produce a valid answer by the rising edge of the next `SYNC` signal, we could use this pipelined ring buffer to copy data we know is valid at the rising edge of `SYNC` onto the bus at that time. If all of the modules on the bus do this at once on a common `SYNC` and rotate on a common `CLK`, it is possible to essentially reset the entire Ring Buffer bus on very `SYNC` to contain only fresh values from the previous audio frame.

3.2.2 Output Drivers & Ring Cycling

The net result of the bus design theory mentioned above is shown in Figure 5, which depicts the hardware necessary to create a constantly "ringing" Ring Buffer which snaps to a new state every SYNC. As is evidenced in the image, it is a very small design using only 2 muxes and 2 registers. The module address programmed onto the address component of the ring buffer is a parameter set at compile time.

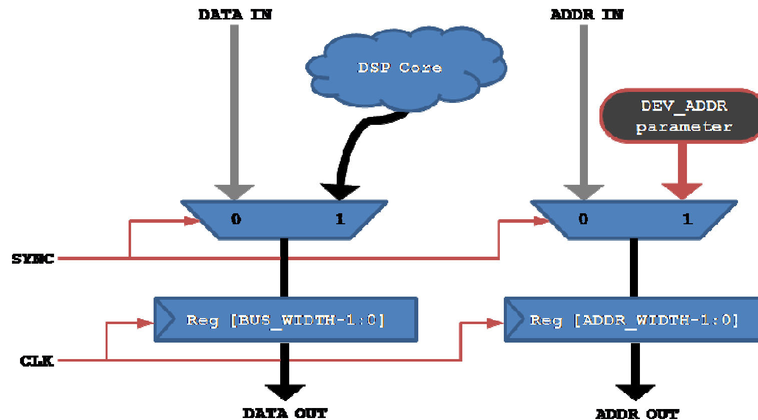


Figure 5. Ring Buffer - Output System

Figure 6 expands on the ringing behavioral spec seen in figure 4 to include a variety of signals located within each audio module. As is shown with the uppermost set of red arrows, the value located on the DSP unit's output is loaded onto the data bus. At the same time, the associated module's address is loaded onto the address bus. Both will then rotate continuously until the next SYNC rising edge, which will load the bus with new data. This functionality is actually coded in *network_flow_controller*.

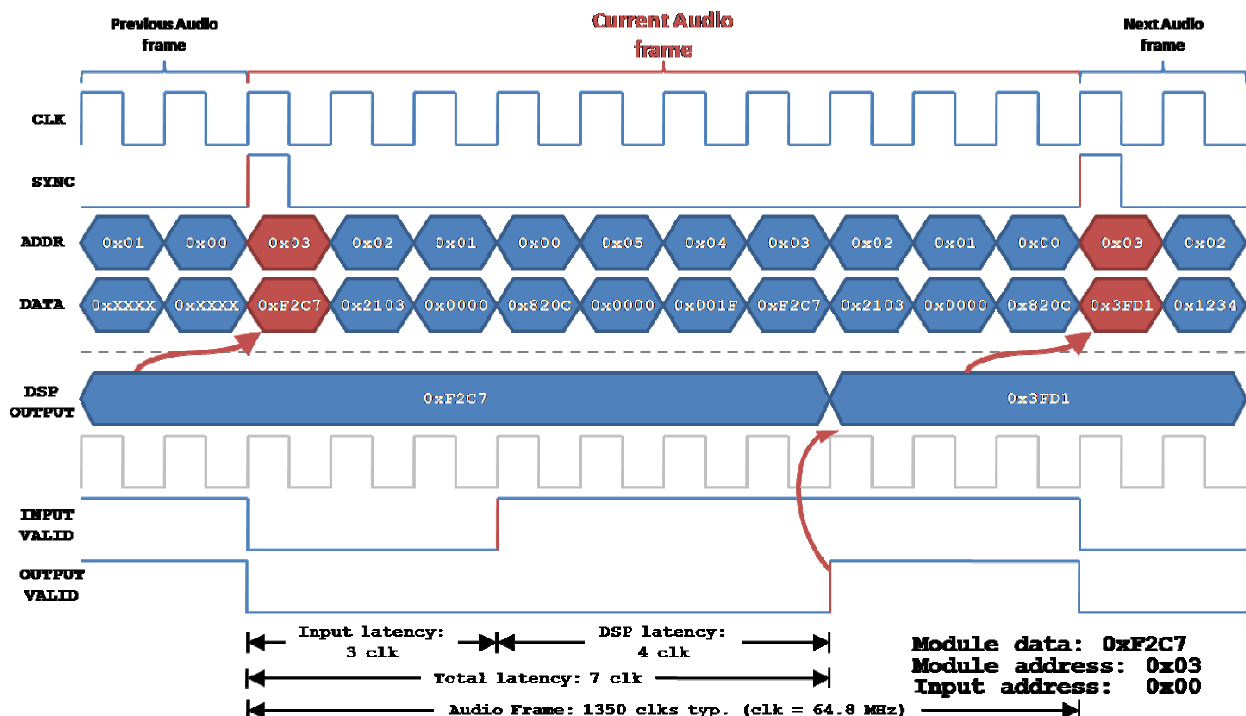


Figure 6. Ring Buffer - Output Behavior

3.2.3 Input Receivers & Control Registers

While the implementation developed so far provides the ability to write values to a ring buffer, this information would effectively be getting lost unless another module has the ability to read the addresses and data values on the bus. The implementation of this system is shown below.

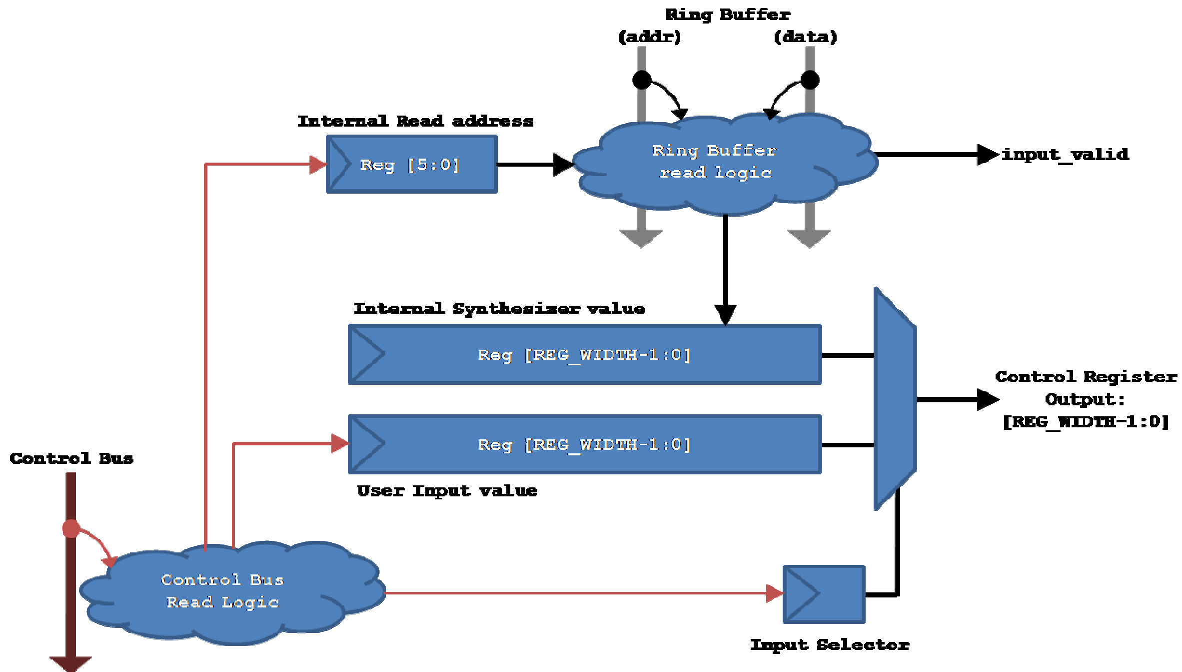


Figure 7. Ring Buffer - Input System

As the figure shows, each *control_register* is designed to output a (nominally 16-bit) value and a single-bit signal `input_valid`. Internally, *control_register* contains 4 globally-accessible registers which can be programmed over the control bus (mentioned in §3.3). Two of these modules are (nominally 16-bit) registers which store a value programmed by the user and a value received from the ring bus.

Additionally, there is a smaller register which can be programmed by the user and contains the audio module's address which *control_ register* will read in on every `SYNC` pulse. Once `SYNC` has been asserted, *control_register*'s `INPUT_VALID` pin will drop to indicate that it is currently waiting to see the ring buffer's address value equal the value stored in the small register. When it sees that value, the control logic stores it into one of the 16-bit registers. This process occurs on every `SYNC` regardless of whether or not that value will be passed along to DSP core. That is controlled by the value of a single-bit register, `INPUT_SELECTOR`.

More information on the control bus architecture and now a *control_register* instance can be programmed is available in §3.3, "Control Signal Routing".

3.2.4 Testing and Debugging

By themselves, both *network_flow_controller* and *control_register* function without needing any external control inputs beyond `CLK`, `SYNC`, and (in the case of *control_register*) a `CONTROL_BUS`. Moreover, the ring buffer implementation I designed is fault tolerant because it separates the input and output stages into two distinct, independent modules which do not communicate with one another directly. This greatly enhanced my ability to test and debug the network control modules because it meant I had to provide a much simpler testbed without worrying about concurrency or timing issues.

The block diagram shown below as figure 8 demonstrates how keeping each of the logical components in the ring buffer implementation as individual low-level modules yields audio modules designs which are solely wrappers and interconnects. In the image below, the only components needed in the audio module are 3 instances of *control_register*, 1 *DSP_Core*, and 1 *network_flow_controller*. Everything else is simply wires.

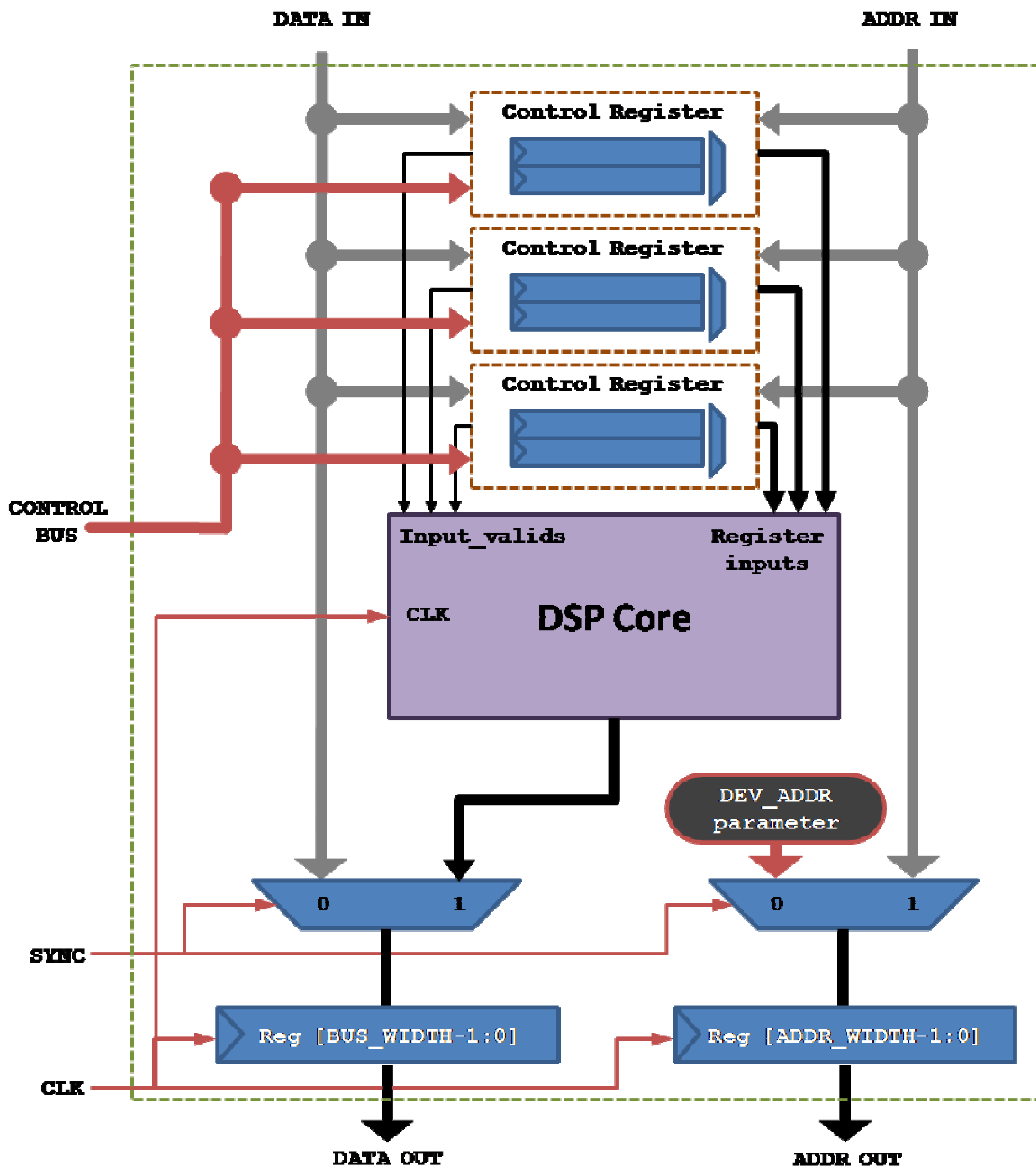


Figure 8. Ring Buffer - Input Behavior

3.3 Control Signal Routing

As I mentioned earlier, the control system used by the Modular Synthesizer relies on a common command and control interface shared among all of the audio modules to set the parameters of each module's behavior. In essence it is like a virtual bank of knobs or sliders which can be commanded to communicate with instantiated *control_register*, although in its current form it is much more like having a virtual keyboard that can communicate with every module.

3.3.1 Theory of Operation

As was noted earlier, each *control_register* instance requires a connection to a common control bus which provides the setup information it needs to function. The required information falls into three categories which correspond to the register where each of them is stored:

1. External (User-Provided) Data Value
2. Valid_Address
3. Input Selector

Between these three values (and Internal Data Value, read from the bus at location Valid_Address) each *control_register* would know when to properly read from the ring buffer and what data to output. Therefore, I designed a 31-bit wide data bus that connects every control register in the Synthesizer to a single master controller, which takes in user input from a serial port and commands a specific register to a certain state.

3.3.2 Control Bus Implementation

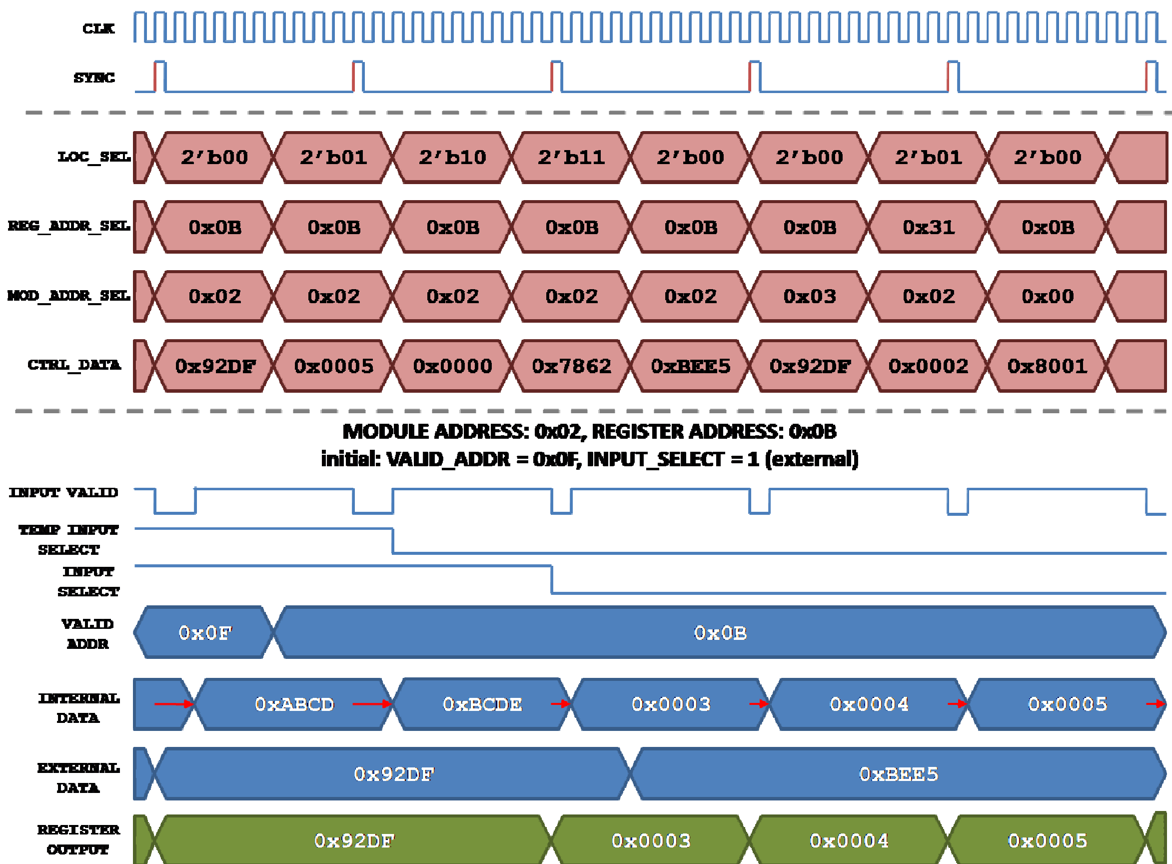
The control bus I implemented which is connected to every *control_register* is 31-bits wide. However, these bits are broken down into four distinct ranges which each control a certain component of the system. These parts are:

1. Control Bus Data (CTRL_BUS_DATA)
2. Control Bus – Module Address (CTRL_MOD_ADDR)
3. Control Bus – Register Address (CTRL_REG_ADDR)
4. Control Bus – Location Selector (CTRL_LOC_SEL)

Each *control_register* then implements a simple conditional logic tree to determine if it is the module which is being commanded to a new state:

```
If (control bus module address == MY_MODULE_ADDRESS) {
    If (control bus register address == MY_REGISTER_ADDRESS) {
        If (control bus location selector == 0)
            External_data_value ← control_bus_data[15:0];
        If (control bus location selector == 1)
            Valid_Address_value ← control_bus_data[4:0];
        If (control bus location selector == 2)
            Location_Selector← control_bus_data[1:0];
        If (control bus location selector == 3)
            Do nothing since there aren't 4 registers to set
```

The following figure demonstrates the behavior of a *control_register* with an initial setup as listed responding to a bunch of user input over the control bus.



3.3.3 RS-232 UART System

As it quickly became apparent within the first week of starting the Modular Synthesizer project that developing a custom LCD touchscreen terminal for user input would be too complex to be possible, I decided to fall back to a simpler serial interface wherein the user would enter an 8-digit value in hexadecimal values (0-9,A-F) which maps to the associated 31-bit value present on the control bus. I implemented a simple state machine which ensures that the user only presses valid hex keys and does not enter the value to the bus until all 8 bytes have been received.

Regarding the actual serializer and de-serializer components themselves, I chose to use the TX and RX Verilog files provided for public use at www.fpga4fun.com. I felt that while writing my own copy of a UART protocol transceiver would be fun, the UART was only being used as a means to get my data into the synthesizer. Therefore, I chose to simply use premade modules and instead focus on helping integrate the system as well as work on a complex finite state machine which could parse human-readable text into control bus values.

3.3.4 Serial Command Parser

The serial parser consisted of a very simple FSM focused around an counter which ranged from 8 (no characters entered on the serial port) to 0 (every character entered, so load value onto the control bus). As this was a very simple implementation of the control interface, it worked without incident at 9600 bps 8N1.

4. Conclusion

The focus of our project shifted quite a bit from our original goal. We had originally had a product-oriented project in mind – with emphasis on the audio and user interface. Our actual final project was much more of a framework. We had originally believed that the audio routing problem was solved for us – we could just serialize audio data in and out of our modules via the user pins on the lab kit. Much of our UI design time was sacrificed as we implemented the ring network and did all of the routing within the hardware.

We also underestimated the time it would take to complete the audio modules. We had believed that while some modules were complicated we would have the time to implement quite a few of the audio modules. As we found out, it is very difficult to implement complicated fixed-point or floating-point arithmetic. We did not expect to run into hardware limitations while doing our calculations.

That said, the end result was much more of a framework that we or any other person could build on in the future. Had we had the audio routing network we have now before we started, we could have focused on the end product interface much more. In addition, the knowledge we gained while implementing the more difficult audio modules would help us finish off the easier ones we had planned much more quickly.

Appendix A - Equations

$$H(Z) = \frac{b_0 + b_1 Z^{-1} + b_3 Z^{-2}}{a_0 + a_1 Z^{-1} + a_2 Z^{-2}} = \frac{\frac{b_0}{a_0} + \frac{b_1}{a_0} Z^{-1} + \frac{b2}{a_0} Z^{-2}}{1 + \frac{a_1}{a_0} Z^{-1} + \frac{a_2}{a_0} Z^{-2}}$$

Equation 1. Biquad Transfer Function
A generic biquad transfer function.

$$y[n] = \frac{b_0}{a_0} x[n] + \frac{b_1}{a_0} x[n-1] + \frac{b_2}{a_0} x[n-2] - \frac{a_1}{a_0} y[n-1] - \frac{a_2}{a_0} y[n-2]$$

Equation 2. Biquad Recurrence Relation
The generic biquad expressed as a recurrence relation between the input signal *x* and the filter output *y*.

$$\omega_0 = f / f_{sampling}$$

$$\alpha = \frac{\sin(\omega_0)}{2 \cdot Q}$$

$$H(s) = \frac{1}{s^2 + s/Q + 1}$$

$$b_0 = b_2 = \frac{1 - \cos(\omega_0)}{2}$$

$$b_1 = \frac{1 - \cos(\omega_0)}{2}$$

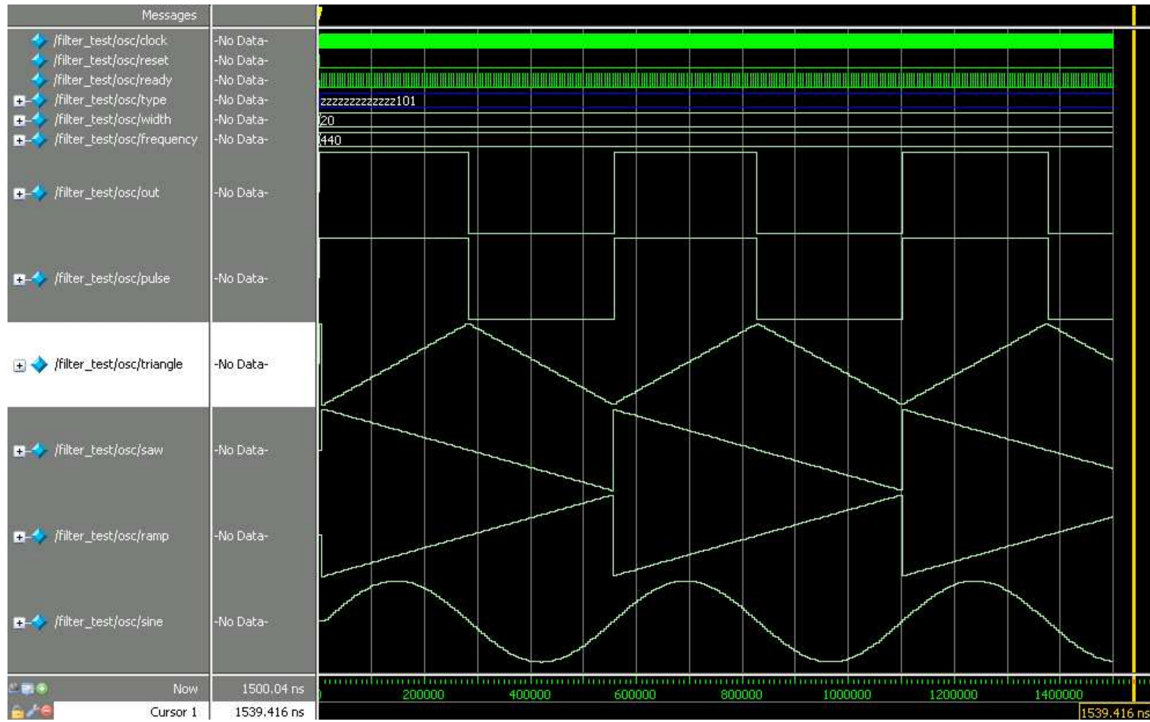
$$a_0 = 1 + \alpha$$

$$a_1 = -2 \cdot \cos(\omega_0)$$

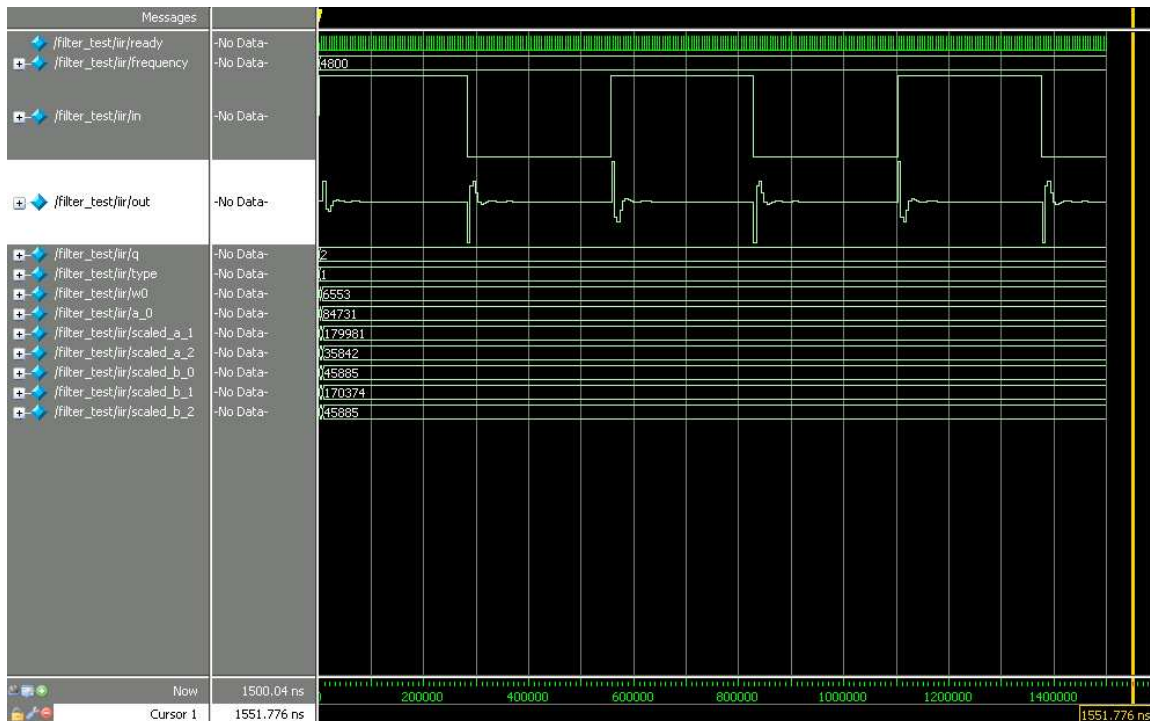
$$a_2 = 1 - \alpha$$

Equation 3. Low-Pass Filter Coefficients
Coefficients for a low-pass biquad filter.

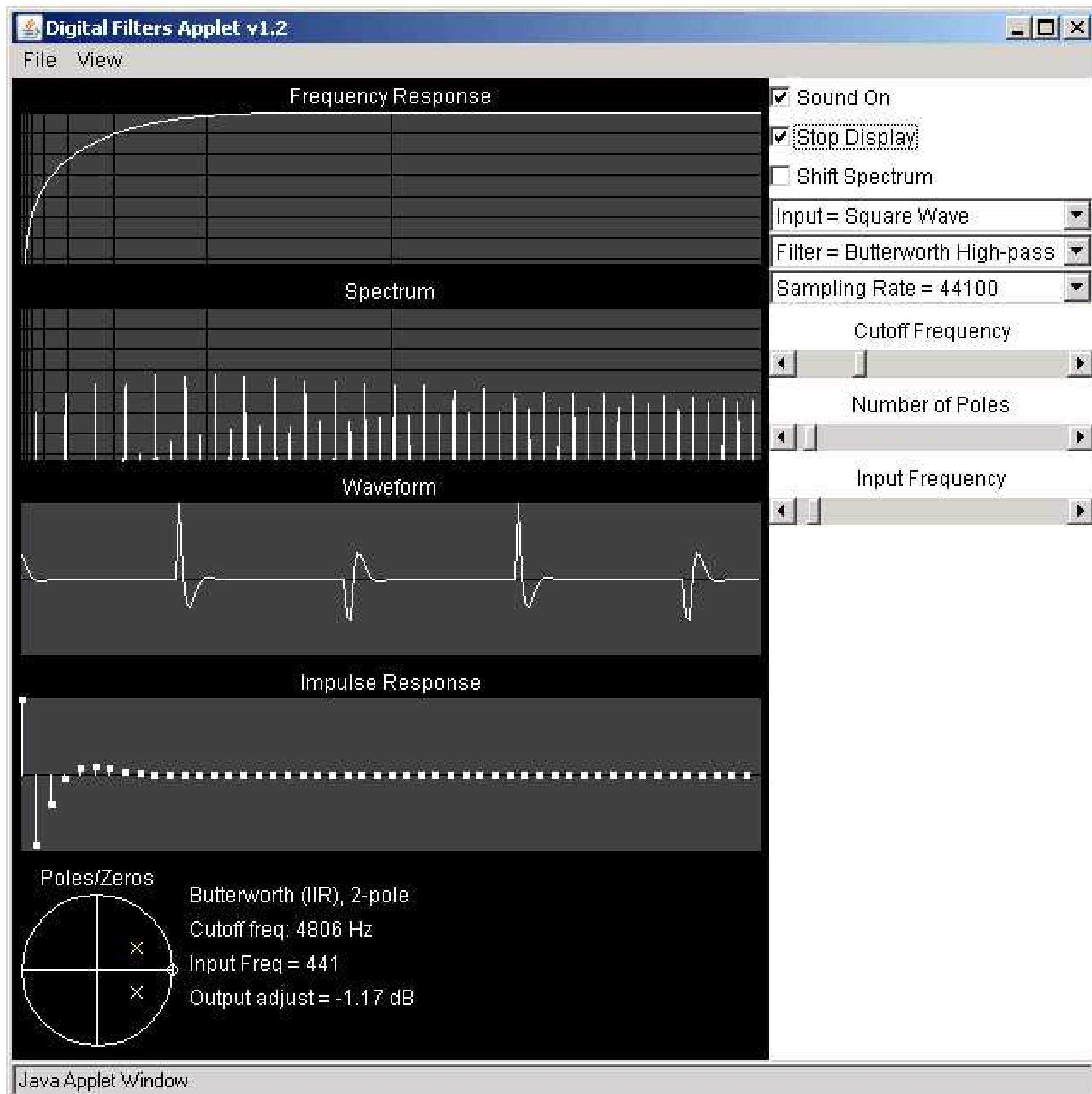
Appendix B - Examples



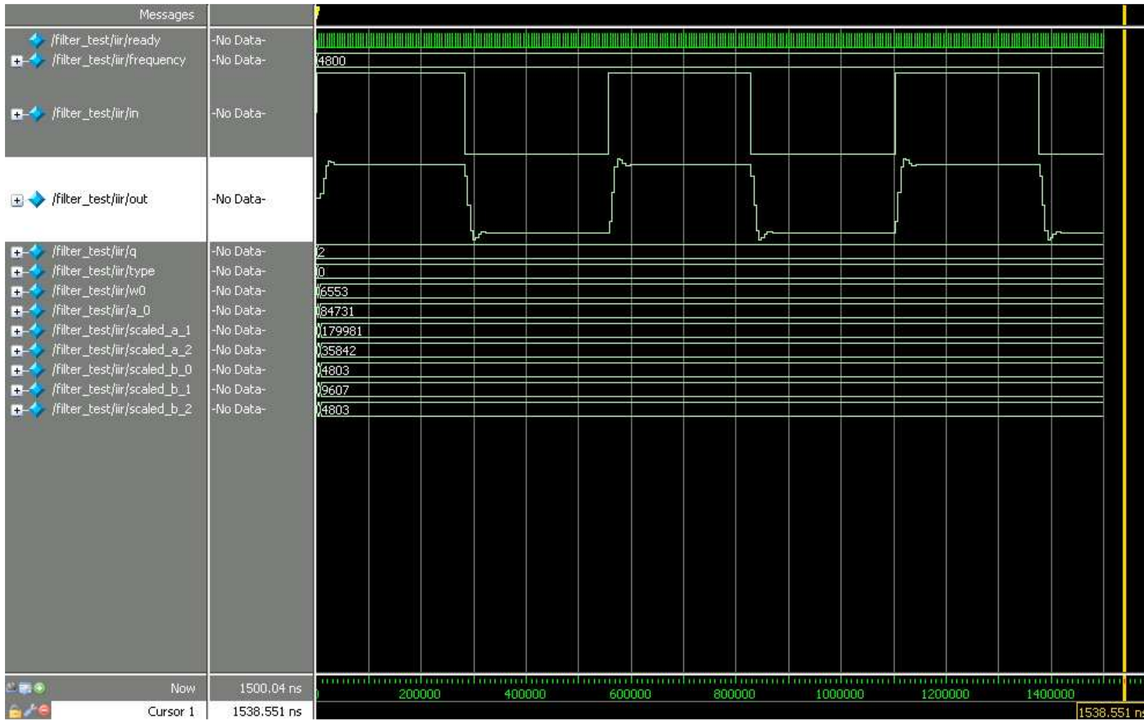
Example 1. Oscillator in Simulation
The Oscillator module’s different waveforms generated for 440hz.



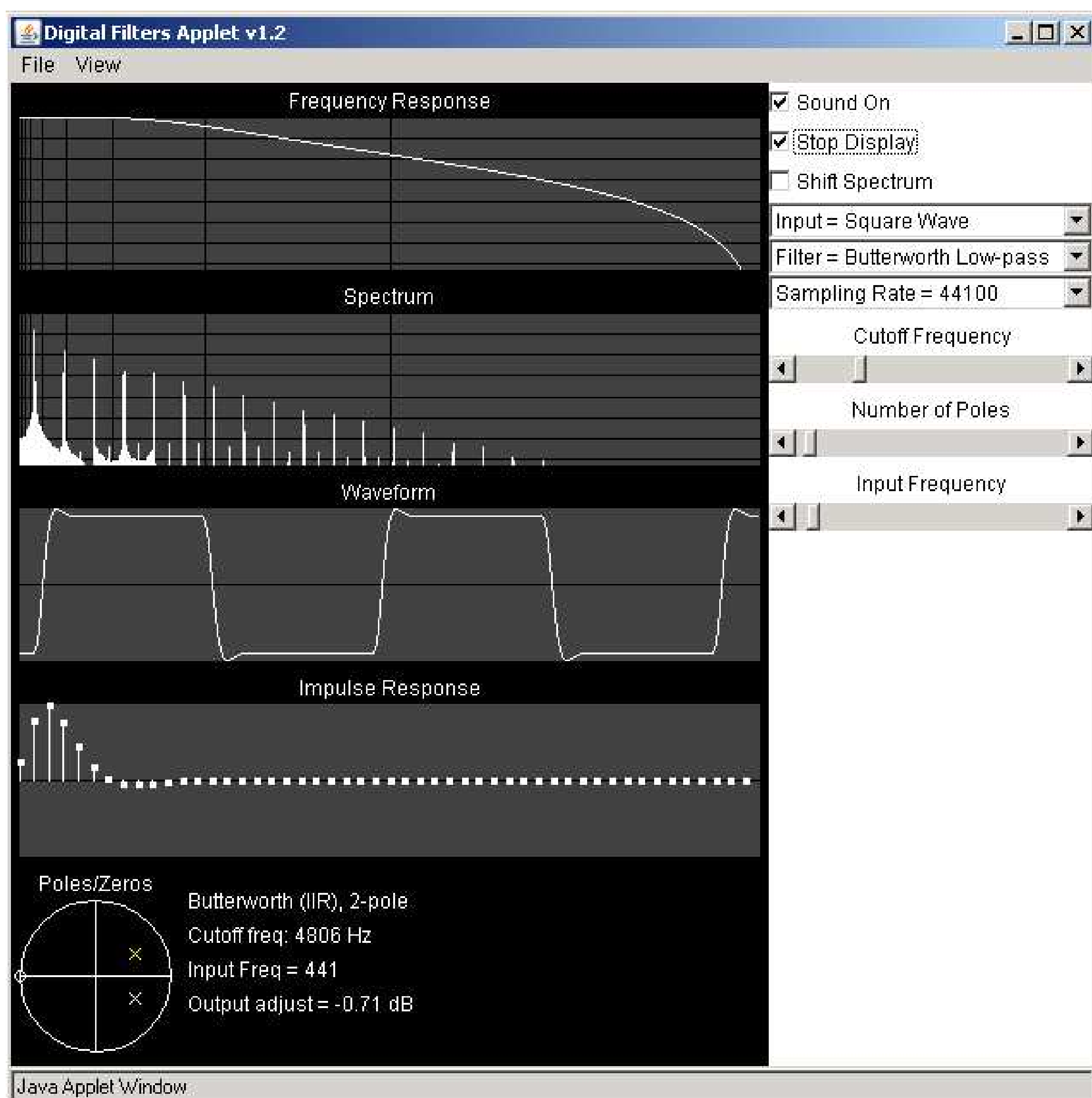
Example 2. High-Pass Filter in Simulation
The output of a high-pass filter with cutoff frequency 4800hz acting on a 440hz square wave.



Example 3. High-Pass Filter Reference
Paul Falstad's Java applet displaying a 441hz square wave with a high-pass filter at 4806hz. Note the similarity of the waveform to Example 2.



Example 4. Low-Pass Filter in Simulation
The output of a low-pass filter with a cutoff frequency of 4800hz on a 440hz square wave.



Example 5. Low-Pass Filter Reference
Paul Falstad's Java applet generating low-pass filtering a 441hz square wave with cutoff frequency 4806hz. Note the similarity to Example 4.

Appendix C - Code

Code 1. Python script for verifying the correctness of the filter test bed.

```
import re
import math

""" converts x, an unsigned decimal representation of
    a fixed-point binary number to a signed decimal
        interpretting x as Qinteger.fraction for unsigned
        or Q(integer-1).fraction for signed """
def fixed_point(x, integer, fraction, signed=True):
        i = 0
        ans = 0
        i = integer
        str = int2bin(x, integer+fraction)
        for k in str:
                i -= 1
                if (k == '1'):
                        if (signed and i == integer-1):
                                ans -= 2**(i)
                        else:
                                ans += 2**(i)

        return ans

""" returns the binary of integer n, using count number of digits """
def int2bin(n, count=16):
        return "".join([str((n >> y) & 1) for y in range(count-1, -1, -1)])

""" check to see if two numbers are equal to within a
    tolerance value of one another. """
def approximatelyEqual(x, y, debug=True, tol=0.01):
        b = abs(x-y) < tol

        if (not b):
                raise Exception("Values "+repr(x)+" and "+repr(y)+" do not match.")
        else:
                if (debug):
                        print ":: ", x, "~=", y

f=open("output.dat")

lines = f.readlines()

data = [0 for i in range(0, len(lines)-2)]

# tokenize the numbers
for i in range(0, len(lines)):
        if (i >= 2):
                d = re.split("[^a-zA-z0-9-]+", lines[i])
                data[i-2] = []
                for v in d:
                        if v != "" and v != "x":
                                data[i-2].append(int(v))

w0 = 2*math.pi*4800.0/48000.0
c = math.cos(w0)
s = math.sin(w0)
alpha = s/4
actual_b0 = (1 - c)/2
actual_b1 = (1- c)
actual_b2 = actual_b0
actual_a0 = 1 + alpha
actual_a1 = -2*c
actual_a2 = 1 - alpha

a0 = 75133
a1 = -106228 #155916
a2 = 55939
```

```
b0 = 6211
b1 = 12422
b2 = 6211

print "a0:",
approximatelyEqual(actual_a0, fixed_point(a0, 2, 16))
print "a1:",
approximatelyEqual(actual_a1, fixed_point(a1, 2, 16))
print "a2:",
approximatelyEqual(actual_a2, fixed_point(a2, 2, 16))
print "b0:",
approximatelyEqual(actual_b0, fixed_point(b0, 2, 16))
print "b1:",
approximatelyEqual(actual_b1, fixed_point(b1, 2, 16))
print "b2:",
approximatelyEqual(actual_b2, fixed_point(b2, 2, 16))

# in, x[0], x[1], x[2], y[0], y[1], y[2], accumulator
i = 0
x_actual = [0,0,0]
y_actual = [0,0,0]
actual_accumulator = 0

for d in data:
	input = d[0]
	x0 = d[1]
	x1 = d[2]
	x2 = d[3]
	y0 = d[4]
	y1 = d[5]
	y2 = d[6]
	accumulator = d[7]

	print "(",i,"):", ("input",input), ("x0",x0), ("x1",x1), ("x2",x2), ("y0",y0), ("y1",y1),
("y2",y2), ("accum",accumulator)
	print "accumulator:",
	approximatelyEqual(b0*x0 + b1*x1 + b2*x2 - a1*y1 - a2*y2, accumulator)
	approximatelyEqual(actual_accumulator, fixed_point(accumulator, 18, 18), tol=50)
	approximatelyEqual(y_actual[0], fixed_point(y0, 16, 2), tol=50)
	i += 1

	x_actual[2] = x_actual[1]
	x_actual[1] = x_actual[0]
	x_actual[0] = input

	y_actual[2] = y_actual[1]
	y_actual[1] = y_actual[0]

	actual_accumulator = actual_b0*x_actual[0] + actual_b1*x_actual[1] + actual_b2*x_actual[2] -
actual_a1*y_actual[1] - actual_a2*y_actual[2]
	y_actual[0] = actual_accumulator / actual_a0
```