The Customizable Audio Kaleidoscope

Tony Hwang, Agustya Mehta, Dennis Ramdass

6.111 – Introductory Digital Systems Laboratory – Final Project
Department of Electrical Engineering and Computer Science
TA: Amir Hirsch
Massachusetts Institute of Technology
May 17, 2007

The objective of this design project is to implement an auditory analogue to the kaleidoscope. First the FPGA kit will be used to sample from a set of stored audio waveforms. Then, these waveforms will be mixed in various aurally pleasing ways through an algorithm based in musical knowledge. Variables to consider would be pitch, tempo, volume, and timbre. The synthesized tones will then be output through a speaker. In addition to this "autopilot" mode, the user will be able to tweak limited characteristics of the final product, and the algorithm should be able to adjust accordingly in real time. Thus, the created music can be as random and pleasing as the visuals of a kaleidoscope, but appealing to the ear rather than the eye.

Table of Contents

Overview 3

User Input Interface 5

Analog / Digital Conversion 5

Switch / Button Inputs 6

Testing / Debugging 6

Control Unit 7

Random Number Generator 7

FIFO Stacks 8

Algorithm Block 8

Foundations in Music Theory 8

RandomSound 10

DecideRhythms 10

DecideOctaves 10

Rhytm2Pulse 11

Convert2notevalue 11

Testing / Debugging 12

Sound Synthesis 13

Modules 13

audio 13

ac97commands 13

ac97 14

sine / gus sample ROM 14

notes 14

address 15

mixer 16

arpeggiator 16

sfx 17

Testing / Debugging 19

Table of Contents (cont.)

Top Level Instantiation 20

Results 20

Conclusion 21

Figures

- **Figure 1:** Top-Level Block Diagram 4
- **Figure 2:** ADC Schematic 5
- **Figure 3:** ModelSim Simulation of Octave Transitions 9
- **Figure 4:** ModelSim Simulation of Notes / Rhythm 10
- **Figure 5:** Waveform of Agustya's Voice Sampled 15
- **Figure 6:** Waveform of Sine Wave Sampled 16
- **Figure 7:** Waveform of Perfect Fifth 17
- **Figure 8:** Waveform with Clipping 18
- **Figure 9:** Waveform with Bottom Threshold 18
- **Figure 10:** Waveform with Clipping and Bottom Threshold 18

Appendix (code included on 6.111 website)

Overview

A conventional kaleidoscope consists of a cylinder that contains a set of colored beads as well as a setup of mirrors. As the user peers into the peephole on one end of the cylinder and rotates kaleidoscope, the colored beads fall in a pseudo-random order and their reflections of the mirrors create a pleasing image for the user. Various characteristics of this device are attractive. Although the operation is simple, the image produced is complex and visually enjoyable. The goal of this project was to create an audio analogue of the kaleidoscope, and improve upon it.

The audio kaleidoscope is similarly capable of generating pleasing audio with minimal effort from the user. It can create music in a variety of tempi, octaves, and rhythms with up to four note chords playing at a time. The user can allow it to run without changing anything, or if he or she sees fit, can adjust in these characteristics in real time.

This implementation is divided up into two major blocks. The first is a control unit that produces notes, rhythms, and other musical variables and changes them according to an algorithm that incorporates music theory to provide some guidance in how notes are chosen, randomness to provide a unique experience to the user, and simple user inputs that allow the user to adjust the music output from the kaleidoscope without having to understand its workings. The second block synthesizes sound from the note values and durations created by the control unit and outputs 18-bit digital audio to the AC97 codec, which in turn will convert this digital signal into audible analog sound.

Figure 1 shows a simplified top-level block diagram of the system. Not all internal wires are shown, but the major signals into and out of each block are indicated along with their bit width.

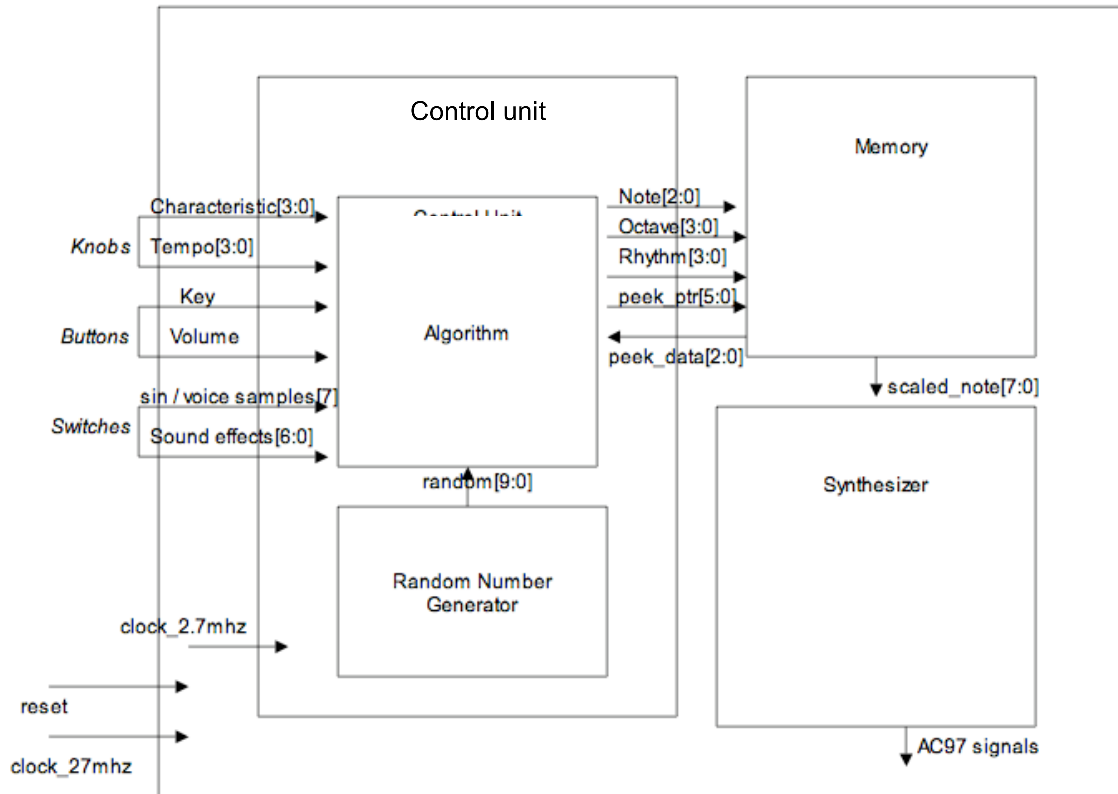

Figure 1. Top-Level Block Diagram for Kaleidoscope

User Input Interface

The control unit's calculation of notes and rhythms depends not only on the weighted probabilities of transitioning among the various possibilities, but also on the inputs of the user in real time. The user is able to adjust tempo, key, and the weights of the notes and rhythms in real time. All button inputs are debounced and synchronized to the system clock. The following section details the implementation of all inputs.

Analog to Digital Conversion

The user can adjust two analog knobs smoothly to vary the weights of the notes and the tempo. These knobs are potentiometers that divide a DC voltage that gets sent through an analog to digital converter (ADC). The ADC's output is finally sent to the user input pins on the 6.111 Labkit. Figure 2 contains a schematic of one knob's connection to the kit.

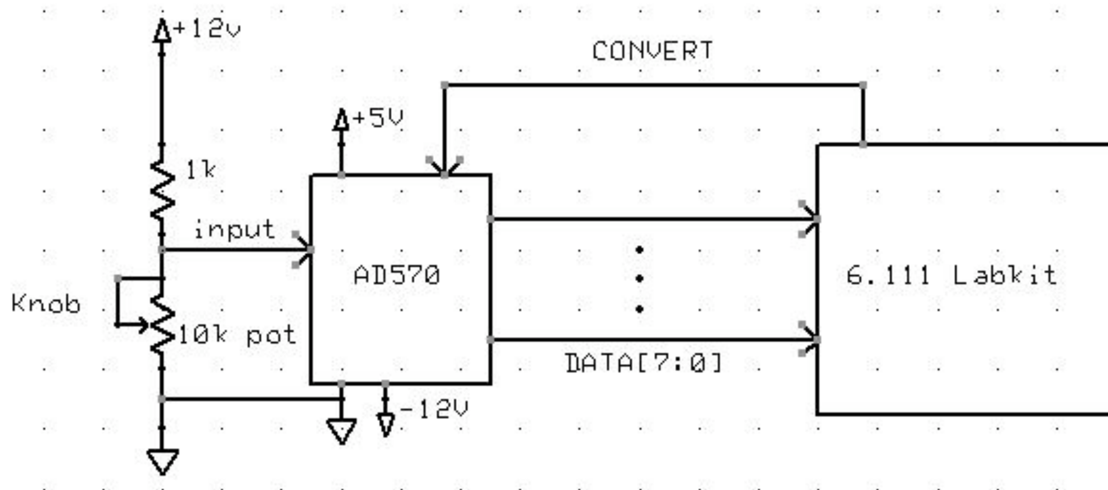

Figure 2. Circuit schematic for knob input. The potentiometer forms a voltage divider with the 1k**Ω** resistor to bias the input to the right range.

The AD570 has a sampling rate of 25 µs. It operates on +5V and -12V supplies, and begins conversion of its analog input upon receiving a convert pulse. To cause the AD570 to convert the input frequently enough so that the user's inputs would be received at an acceptable resolution, a clock divider module was instantiated to send a CONVERT pulse every 370 µs for a

duration of 1.5 μs (verified by data sheet). The converted digital data has an 8-bit resolution, which is plenty considering that we only needed eleven levels of variation for our inputs.

Whenever one of the characteristics is adjusted, a signal is sent to the algorithm so that it knows to reset and incorporate the change.

Other inputs (buttons, switches)

The scale key is adjusted up and down by two buttons on the lab kit. There are also eleven levels that correspond to each key separated by half-steps.

The switches are used to control the outputted sound's characteristics. Switch 7 is used to toggle between output of the synthesized sound wave and the recorded audio sample. Switches 6 to 0 are used to control the degree of distortion added to the chosen sample. Both key and sound characteristics are independent of the algorithm's process of note selection: key is simply added to the root note value supplied by the control unit (chords built upon this root will be transposed accordingly), and distortion effects are simply characteristics of the sound itself.

Testing and Debugging

The analog interface was relatively straightforward so not much debugging was necessary. First, the voltage divider was built and the oscilloscope was used to verify that the divided voltage was falling in the correct range. The desired voltage range was 0 to 10 V because the AD570 is capable of converting this range to digital values. Although theoretically the divider should have been built with a 2kΩ resistor and a 10kΩ potentiometer ($V_{divider}$ = 12 * (R_{pot}/(R_{pot}+2) volts), in practice we were seeing around a 7.75 V maximum. Switching to a 1kΩ resistor solved this problem. After testing the voltage divider, the AD570 was wired up and its output bits were examined on the logic analyzer. It was verified to read logical relative values as the potentiometer was turned.

Control Unit

The Control Unit designed consisted of 3 blocks: the random number generator, FIFO stacks, and the block implementing the algorithm to choose notes. The algorithm block uses a random number from the random number generator and data from the FIFO stacks in conjunction with a set of predetermined musical rules and tendencies to produce non-deterministic pleasing musical output. This output is put on to the FIFO stacks to be read at an appropriate time. The control unit is clocked on a 2.7MHz clock (which was implemented using a clock divider module) since the 10-bit random number output from the random number generator changes every 10 cycles of the 27MHz clock.

Random Number Generator

True random number generation in hardware can be quite a challenge. Initially, the possibility of sampling noise (perhaps Johnson or Shot noise from a resistor) to produce random values was contemplated, but in the interest of spending time on more relevant parts of the project, it was decided that our system could use pseudo-random numbers with minimal negative impact on kaleidoscope performance.

These pseudo-random numbers were registered using a Linear Feedback Shift Register (LFSR). The LSFR outputs a random sequence of runs of zeros and ones by shifting out a bit at a time from a fixed-length register and shifting in an exclusive-or value of values at specific positions in the register. These positions called “taps” are optimally determined by some governing polynomial. The initial value of the LSFR is called the seed and can be any value except all bit values being zero. The seed is created in the submodule RandSeed; this module counts up from initial system reset at every clock edge, providing a relatively arbitrary seed.

Since a 10-bit random number was needed by the algorithm, it was decided that the LFSR should be 16 bits long. The LFSR was implemented using a Xilinx Coregen LFSR module since it was more convenient to design an optimal 16-bit LSFR with a pre-built module containing a lookup of the optimal tap polynomials rather determining it manually. The seed of the LFSR is calculated by determining the number of cycles on the 27MHz clock from the time the system is reset to the user system reset.

FIFO Stacks

In this project, there are essentially two clock domains – the clock with which the control unit operates and the 44 kHz (actually, 44 kHz * 255) AC97 bit clock that audio is being sampled with. In order to avoid loss of data due to the inability for audio to be sampled at the clock speed of the control unit, a FIFO (First-in First-out) was used as a high performance parallel interface between the two different clocking domains.

The FIFO implemented in the project possessed all the properties of a standard FIFO: data can be written onto the stack and read off the stack at the command of read and write signals and the stack full and empty constraints being fulfilled. FIFOs are used to store note, octave and rhythm values. The implementation of the algorithm requires "data-peeking", i.e. looking at a certain position without taking the value off the stack, for the FIFO. This meant that the traditional FIFO model had to be modified to allow outputting of its read and write pointers as well as a "peek" at a certain value at a time. Although this behavior is better suited for a RAM, only a small number of cells compared to the size of the stack needed to be "randomly accessed"; the FIFO was used because it better suited the general needs of the system.

Since user inputs needed to implemented in real-time and these FIFOs contained values generated well before the values are used to generate sound output, the FIFOs have to be cleared whenever the user input value changes.

Algorithm Block

Foundations in Music Theory

The control unit is capable of generating scale interval values 1 through 7, octave values 0 through 16, and choosing from a sequence of predetermined rhythms. These values are generated based on algorithms that are founded primarily on basic rules of musical composition.

Notes are chosen through tables of transition probabilities. These probabilities are founded in traditional Western music theory. Each scale consists of seven scale intervals. Certain scale intervals transitioning to other intervals traditionally have had certain meanings in music theory. For example, certain progressions sound like the resolution or ending of a musical phrase, while others imply uncertainty. By weighting each note in a transition probability matrix and accounting for these musical rules, it is theoretically possible to have an algorithm generate a moderately structured progression of notes. The issue of determining an algorithm to

automatically generate pleasing progressions has no single concrete solution, though this project hopes to come close to something that is both flexible but follows some basic rules in order to sound “good” to a Western ear.

The octave generation algorithm depends on the generated notes as well as the history of octaves that have occurred in the recent past. It calculates the octave score for four notes at a time by weighting the effect of three relevant characteristics: a measure of how high the note intervals are (Note Variation Sum), a measure of the deviation between each of the four notes (Delta Variation Sum), and finally an octave history (Octave History Sum). The note values (1-7) are added to calculate Note Variation Sum, the differences between the notes are added to determine Delta Variation Sum, and finally a comparison between the octave value for the current note and the octave value four notes previous. These characteristics are weighted and added together to calculate Unscaled Octave Variation Sum. Finally, this sum is scaled to either change the current octave up or down by one, or remain the same. The scale is designed such that the octave does not change too frequently, but still in a manner that some variation will occur, a guideline that many composers follow. A ModelSim screenshot of signals relevant to octave generation is shown in Figure 3.

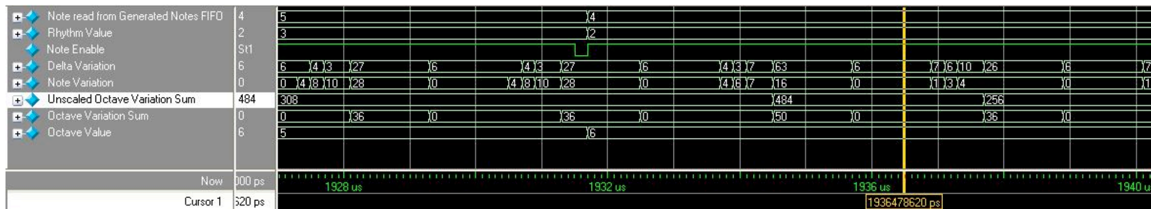

Figure 3. ModelSim waveform demonstrating octave generation algorithm

Independent of note and octave generation is the rhythm generation algorithm. Its intended design would have taken into account note interval values, though its implementation would be an extremely complex problem beyond the scope of this project. Thus, the algorithm transitions among a set of preprogrammed rhythms applying a similar concept to the note generation algorithm. The possible rhythms are all even divisions of the quarter note beat (quarter, eighth, sixteenth notes, as well as the same kinds of rests). The reason more complex rhythms (such as triplets) were avoided was the inconvenience of dividing by uneven numbers. The semi-random selection of combinations of these rhythms results in a satisfying amount of

variation in rhythm. Figure 4 shows the transitioning of rhythms, where each number corresponds to a pattern of notes.

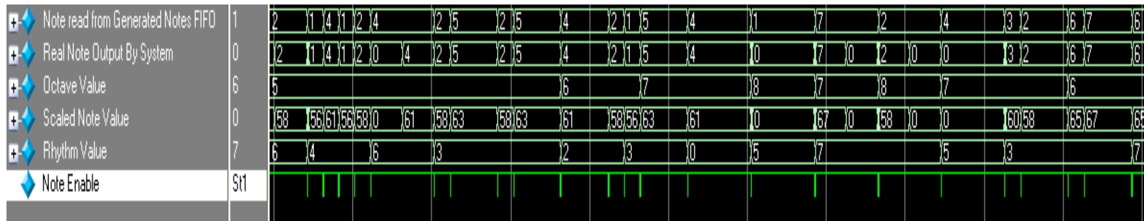

Figure 4. Rhythm and note transitions

The algorithm block consists of 5 modules:

RandomSound

This module generates random notes based on the preset probabilities of note transitions as well as the user "randomness" input. A Finite State Machine (FSM) is used to implement this by representing each note both as a state and an output. Given the state, the user "randomness" input and a short note history (implemented as a history of 2 notes), the probability of transitioning out of the state to every other state is calculated by a *reweight* module and used by the FSM to determine next state (note). The short note history is used to limit the occurrences of repeated notes and cycling of 2 notes consecutively. The output of this module is put onto the Generated Notes FIFO.

DecideRhythms

This module works essentially the same way as RandomSound in its implementation of a control FSM which transitions state based on probabilities calculated by a *reweight_rhythms* module. A difference is that a rhythm history is not needed to determine the next rhythm. The output of this module is put onto the Generated Rhythms FIFO.

DecideOctaves

This module is deterministic and uses scoring to calculate the octave at which a set of 4 notes (a 4-note bar) should be played in. In order to do this, the algorithm "peeks" at 4 values off the Generated Notes FIFO as well a value 4th from the top of the Generated Octaves FIFO in order to make its calculations. Since this module is deterministic, the states and state transitions

in this module's FSM are different in structure from RandomSound and DecideRhythms. The following states are present in the FSM:

INITIALISATION: The system goes into this state right after reset and stays in this state for 4 clock cycles while a constant value is written as the first 4 octaves in the system.

FILL_STACK: In this state, internal copies of a notes stack and octaves stack are filled with values the algorithm needs to look at to determine octaves. The system stays in this state for 4 clock cycles as the algorithm "peeks" at 4 octave and note values.

SUM_DELTAS_AND_NOTES: In this state, some measures of variation are calculated from values in the internal copy of the notes stack. The system stays in this state for 4 clock cycles as the values are accessed and summed.

GENERATE_OCTAVES: In this state, a scoring formula to determine octaves.

WRITE_STATE: In this state, the corresponding octaves for the next 4 notes are written to the Generated Octaves stack. The system stays in this state for 4 clock cycles.

Rhythm2Pulse

This module uses the values stored on the Generated Rhythms FIFO to calculate the timing signals for reading from the Generated Notes and Generated Octaves FIFOs. A *note_enable* signal is generated to read from the notes and octaves FIFO and a *rhythm_enable* signal is generated to read the next rhythm. This module used an FSM structure to control the timing of these signals where each state corresponded to the given rhythm and the system stayed in a given state for as long as needed to pulse *note_enable* as many times as necessary for that rhythm. When the FSM is ready to transition state, it pulses the *rhythm_enable* signal to read the next rhythm.

Convert2NoteValue

This module simply calculates the scaled note output for the system on a scale from 0 to 127 based on the note value read off the stack and the octave value.

Testing and Debugging

Testing of the control unit was done almost exclusively through simulation using ModelSim. This was done because testing independent of the sound synthesis modules was needed to ensure correct behavior. Simulation also proved a much faster and more flexible testing tool. Each submodule of the control unit was tested individually and then everything was wired up and tested as a unit. Simulation allowed easy debugging and tweaking of the algorithm. Screenshot of the algorithm outputting notes, octaves, rhythms and scaled notes randomly are shown in Figures 3 and 4 in earlier parts of this section.

Much of the algorithm was first coded in software platform using Java and MATLAB; MATLAB was useful in that it is capable of producing actual sound output; the initial testing that was done using this software-based system thus did not require the exceedingly long compile times required of implementing the algorithm in hardware.

Sound Synthesis

The sound synthesis block operates independently from the control unit. It consists of a number of submodules, which are described below.

audio

The "audio" module takes in the system clock and contains ports for all of the signals for the AC97 codec (audio_reset_b, ac97_bit_clock, ac97_sdata_out, ac97_sdata_in, ac97_synch), as well as the digital audio input supplied by the system, the volume, and the "ready" pulse, which is described under the ac97 module. It serves to abstract away the details of the operation of the ac97 and ac97commands modules but supplying these with the 18-bit audio input data for the left and right channel and their corresponding wires. The module outputs signals processed by its submodules in order to drive sound on the labkit.

ac97commands

"ac97commands" contains the default command data to configure the registers to unmute the sound output from the headphone jack. It also adjusts the volume register for this output (which this particular system is using) based upon user input from the upper-level audio module.

ac97

The control for the ac97 codec takes in audio data and command data from the audio subsystem and produces valid AC97 codec signals so that the sound on the labkit can be driven. Briefly, AC97 works by producing a succession of 44 kHz (or whatever rate the audio is sampled at) digital data signals. Each signal begins with ac97_synch being high, signaling the beginning of a frame; the data signal ac97_sdata_out is a one-bit signal. This is divided into 255 bits: a tag at the beginning to signify which parts of the signal are valid, and a succession of 20-bit frames that control the register values which control the audio chip and supply audio data to the chip. The ac97commads module provides the desired instructions to be written to sdata_out during the control frames; the 18-bit audio left and right data is then sampled. (Only the first four frames are used in each 44 kHz cycle, and even though the frame is 20 bits, only 18 bits of audio data are used). The ac97 data signals are clocked by the ac97_bit_clock, which operates to produce the desired sample rate of 44 kHz; upon system startup, the module waits a number of system

clock cycles before starting the bit clock to make sure that the AC97 is reset clocked properly. The ac97 module also produces a one 27 MHz clock-cycle width pulse called “ready” which signals that an ac97synch pulse has begun, in order to communicate with the rest of the system which uses the 27 MHz system clock rather than the ac97 bit clock.

sine

“sine” is a 16384 location ROM containing one period of a sine wave. Each sample is 18 bits wide. 16384 (2^14) was chosen as the number of samples to use because it far exceeds the number of samples required to perfectly construct a 44 kHz sampled wave at the lowest frequencies that the system would realistically use. The samples were created in MATLAB by sampling a period of a sine wave into 16384 parts, multiplying it by 131071.5 and shifting down by .5 (effectively putting the range between 131071 and -131072) and then converting to 18-bit two’s complement.

gus

“gus”, aside from being one of the coauthors of this paper and this project, is the name for the ROM that stores 16384 samples of a period its namesake’s voice singing “ahh”. The sound was recorded on a PC in Wavelab and a single period of the audio was cut out and stretched to 16384 samples. (This software is designed to incorporate relatively advanced interpolation, which is why I used it, rather than simply “connect the dots” linearly.) These samples were sent to Wavelab (using the wavread command), stretched to a magnitude of 2^17 bits (between 131071 and -131072), and then converted to 18 bit two’s complement to be stored in the ROM.

notes

“notes” is a 128-location ROM. It is a lookup table that translates logical note values from the control unit into frequencies. The values for frequency are actually the number of samples to skip when playing the 16384 sample stored waveform in order to produce the desired output frequency. (e.g., a “frequency” sample skip value of 128 would produce a complete waveform in 16384 / 128 = 128 samples; since the sample rate is 44 kHz (samples / second), the output frequency would be 44,000/128, or approximately 344 Hz. The frequency value is stored

as 14 bits, and the value associated with each note was calculated in MATLAB by taking a base known frequency (in this case, 440 Hz, which corresponds to A above middle C, or A4) and multiplying it by 2^(n/12), where n is the number of notes away from A4. Note 0 would correspond to C0, note 12 to C1, etc. Since "note 0" is ridiculously low and beyond the range of human hearing, the frequency value here was actually replaced with 0; in this way, note 0 signifies a rest because the system will skip "0" samples, i.e., stay at the same sample.

address

"address" takes in the frequency skip value from the note ROM and outputs the address to sample from in the 16384-location sample ROM; it is simply a counter which increments by "frequency" on each "ready" enable pulse. It only resets to 0 on reset or if the address is 0 (since address 0 of the stored ROMS is 0 or -1, this means that during a rest, there won't be any annoying DC offset during mixing with other notes), in this way, it "loops" around the stored samples at likely a different number each time, maintaining the same distance from each previous sample to provide a fairly accurate rounding interpolation.

Figures 5 and 6 show the logic analyzer output of the waveforms of the Gus voice and sine wave samples sampled at middle C, showing that the addresses progress as desired.

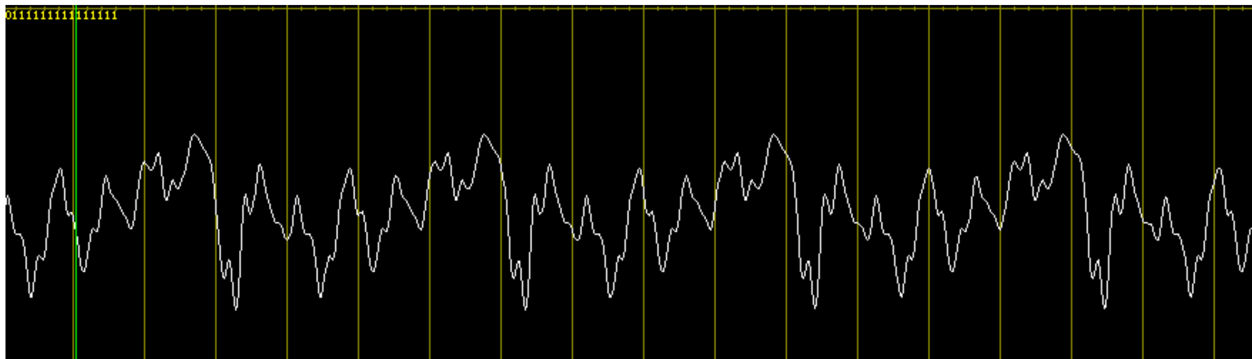

Figure 5. Waveform of Gus's voice at middle C

\

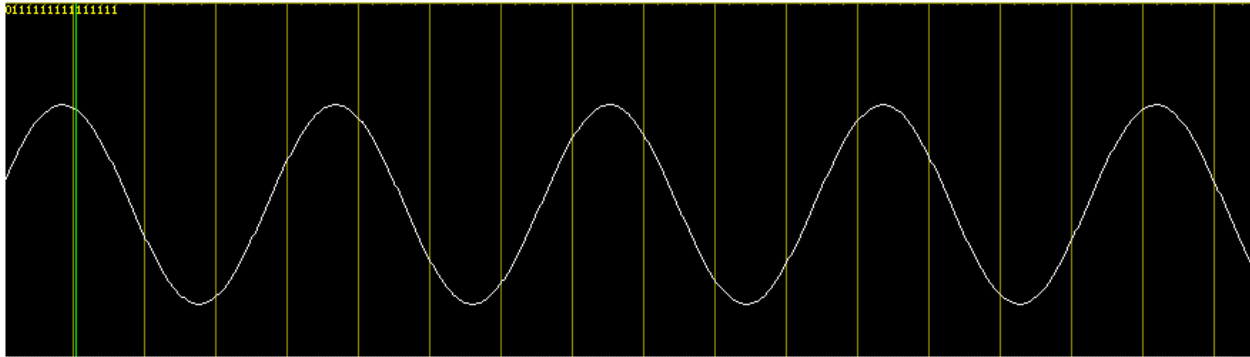

Figure 6. Waveform of sine wave at middle C

mixer

The mixer module simply adds the 18-bit audio data from two separate instantiations of the address, note, and sample memory to produce one audio signal that is the sum of them. To avoid distortion, the signals are scaled down by 1/2 before being summed. Since only four channels of audio are being used in this system (only 4 notes are ever played simultaneously), only a 2-input mixer is needed, since both the left and right audio channels are used.

arpeggiator

The arpeggiator module is somewhat misleadingly named, as a true arpeggiator incorporates many of the tasks provided by the control unit in providing its own progressions. Regardless, this module takes in a single root note (either from the user directly when in the demo version, or from the control unit) and also input on how many notes to play, what kind of quality to play them with, and what kind of inversion to use. It builds chords (or in the case of two notes, intervals); it can select between 1 to 4 notes to play. When playing multiple notes, the quality of the chord or interval has one of eight possibilities, which are as follows. For two-note intervals, the system plays major seconds, minor thirds, major thirds (the same interval as a diminished fourth, since we are using an equal-tempered scale, i.e. 12 evenly spaced notes per octave), perfect fourths, augmented fourth/diminished fifths, perfect fifths, minor sixth/augmented fifths, and major sixths. For three-note chords, the system plays suspended second, major, minor, suspended fourth, diminished, “power” (open fifth with octave above), augmented, and sustained fourth and sixth chords. For four-note chords, the system can play

augmented minor sevenths, minor sevenths, major sevenths, diminished sevenths, minor sevenths with flattened fifths, dominant sevenths, augmented major sevenths, and major sixths. An attempt was made to have the chord types correspond loosely with one another in different note-number chords, but this has a number of holes due to the fact that only 8 chord types were used. The (maximum of four) output notes are sent to be mixed together and processed by sound effects.

Figure 6 shows a perfect fifth; note that the summed waveform is not as symmetric as a truly perfect fifth would be because we are using an equal tempered scale, so fifths are off slightly from their ideal 2:3 ratio.

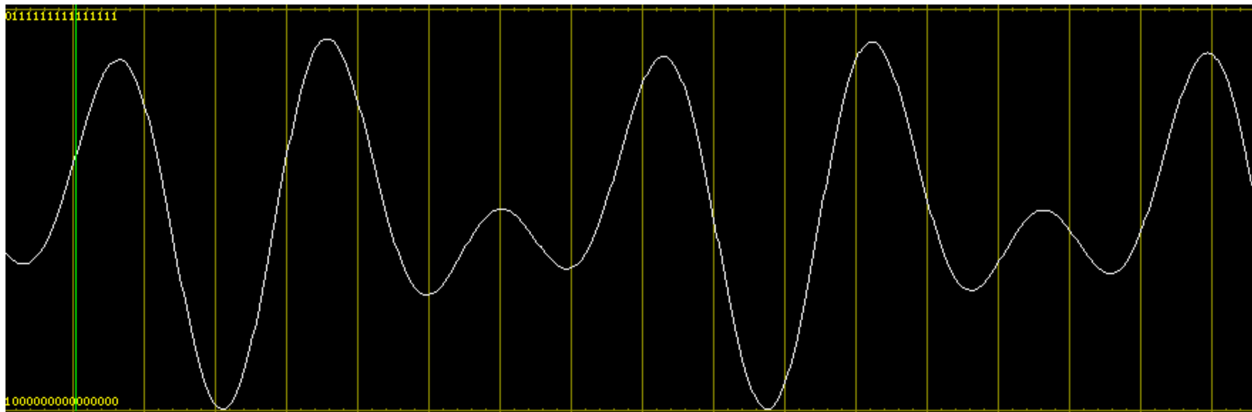

Figure 7. Middle C with G

sfx

The sound effects module can distort a waveform. It takes in inputs of high-threshold and low-threshold values, and clips the waves at the high-threshold values (at the top and bottom, i.e. the threshold is a magnitude), and zeroes the wave below the low-threshold value. The result is an interesting distortion, as can be seen in the following figures.

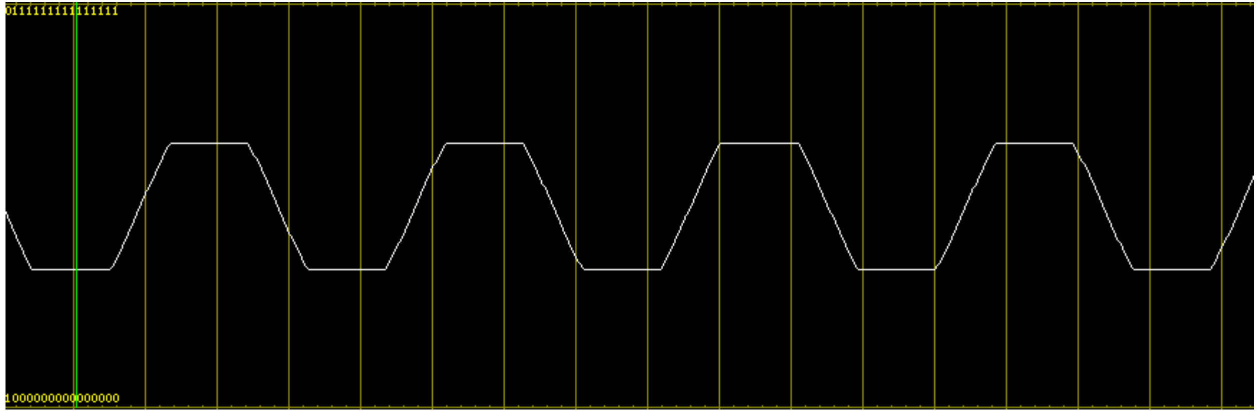

Figure 8. Sine wave with top clipped

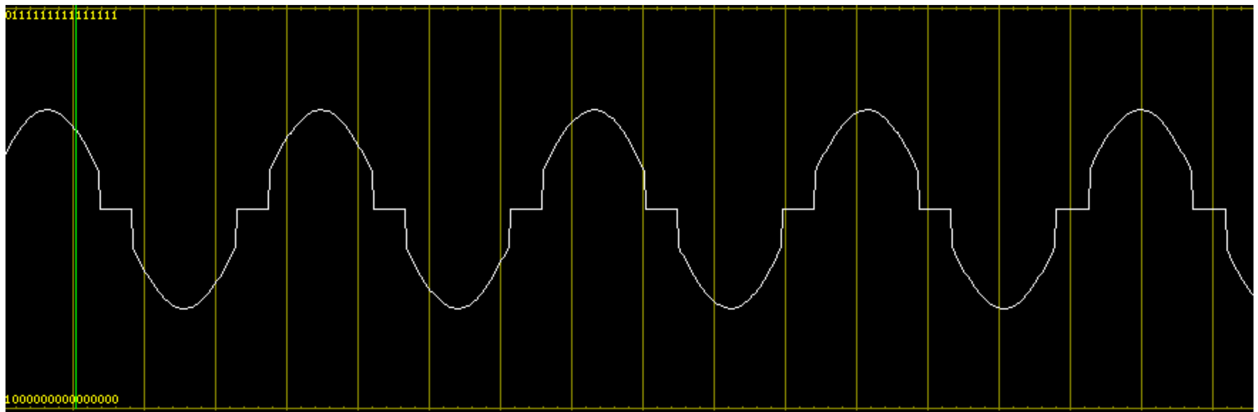

Figure 9. Sine wave with bottom threshold

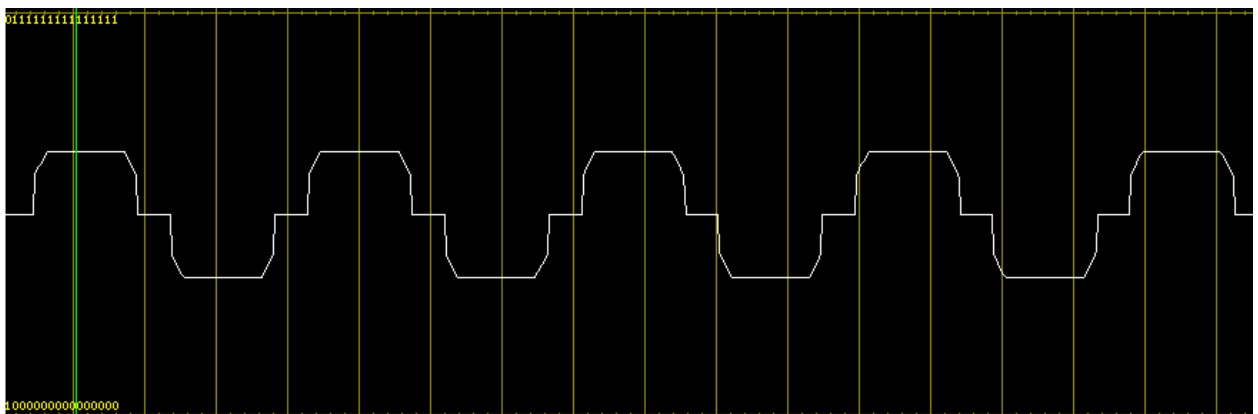

Figure 10. Sine wave with both top clipping and bottom threshold

Testing and Debugging

The apparently simple task of communicating with the AC97 codec was perhaps the most frustrating aspect of this project. Initially, a modified version of the Labkit audio tutorial code was used as the basis for the AC97 module; the code seemed to work just for a simple square wave that was created by using a counter and having it send alternating high and low values to the digital input of the AC97, and the labkit produced sound. However, for some reason, when the output of the sine wave ROM (or a large case statement producing a sine wave, which is effectively the same thing but was later not used due to long compile time), the AC97 ceased to function. Everything simulated properly in ModelSim (using artificial 27 MHz and ac97_bit_clocks, although after this problem was figured out, all debugging was done on the logic analyzer), and an identically structured case statement to produce a square wave (albeit with much fewer lines) also worked. The signals from the sine wave ROM even showed up perfectly on the logic analyzer – until they were connected to the AC97 module. This bizarre problem was worked on for four days to no avail; finally, the AC97 module code was completely rewritten based upon a previous 6.111 lab that provided a basic structure for an audio subsystem, and everything worked perfectly. While it was frustrating to wrestle with a problem for so long and not get a satisfying solution, the problem most likely had to do with the tutorial AC97 code not providing the same amount of time delay before starting the AC97 bit clock upon reprogramming.

Beyond this problem, testing and debugging went relatively smoothly; each new module layer (mixing, sound effects) worked as desired, and the outputs to the logic analyzer matched expectations. Screenshots of various waveforms of output are shown in the figures used in this section; they were clocked on the "ready" pulse in order to create smooth-looking waveforms.

Top Level Instantiation

Two top-level instantiations were used: one is the “demo” code, which maps the button inputs to a primitive C-major scale keyboard when the sine wave output is selected, and maps the available switches to control sound effects. When the “Gus voice” output is selected, the buttons control the sound effects and shift the key up and down, and the switches control the number of notes to play, the type of chord to build, and the number of note inversions. This demo version allows demonstration of the different music synthesis abilities of the system without having to wait for them to occur randomly.

The final instantiation of the code incorporated the control unit as the source for the root note input to the sound system rather than direct input from switches. In this compilation, switch 7 still shifts from sine wave to “Gus voice”, and the remaining switches still cause the same distortion sound effects, but the buttons are now mapped to changing volume and shifting key, and the knobs allow users to alter randomness in musical parameters as described above.

Both the “demo code” and the final code writups are included in the appendix, along with all MATLAB code used.

Results

The system is random in nature and design so the way in which the results of the system match up to what the control unit is “supposed” to produce is difficult to gauge and analyze precisely. The user inputs seem to consistently affect the system in real time in an expected manner. This suggests that the user interface works and is correctly integrated into the system, though it is difficult to notice the changes in “randomness” of the music. After several runs for extended periods of time, the system exhibits behavior that shows that the musical tendencies and rules incorporated in the algorithm are being followed. There are distinguishable instances of random behavior of the system, which suggests that the system is not overly rigid in enforcing musical structure. Sound effects and synthesis work correctly and can be demonstrated to do so separately from the algorithm control unit using manual inputs and switches. The sound effects and synthesis are also seen to work perfectly when wired to the control unit.

Overall, the system performs very well and meets our expectations. Even better behavior could be established by tweaking certain parameters and minor algorithm adjustments after listening to the system perform over an extended period of time.

Conclusions

The results exhibited by this system are very promising and meet most expectations. Music theory is not an exact science and there are many different ways to produce pleasing sounds; establishing exact patterns is impossible. Mapping music theory to a mathematical algorithm is very difficult even on software-based platforms and even more difficult on a hardware-based platform as in this implementation. A particular algorithm was difficult to settle on and constantly changing its implementation in hardware (Verilog) proved challenging and tedious to debug. In the future, other musical traditions besides the classical Western progressions used in this implementation could be used, and the qualities that are universally understood to be musically pleasing could be studied in an attempt to make an audio kaleidoscope not limited to the Western musical tradition.

The sound effects and synthesis modules proved difficult in the initial stages. The AC97 interface was especially bothersome to implement and debug as it sometimes exhibited seemingly arbitrary behavior. Despite initial setbacks, the sound modules progressed extremely quickly and by the end of the project, all the effects desired were implemented in addition to effects beyond initial expectations. Future implementations could include functionality to sum arbitrarily many sine waves of multiple frequencies, effectively allowing the user to synthesize the timbre of any instrument, rather than directly storing samples for each sound. Also, more effects such as emulating the attack/sustain/decay pattern of natural instruments or implementing echoes and reverberations could be added. The system is well integrated and has no major bugs. The user interface is somewhat limited because of time constraints but this can easily be remedied in future development. With feedback from extended system usage, information can be gathered to tweak the algorithm to perhaps get even better system performance – and speaking of feedback, another improvement in implementation would be to have the actual transition probabilities stored in memory change with user feedback, rather than having the user only have in influence on how random the system is. The kaleidoscope is also fun to use and could potentially be developed into a commercial electronic entertainment device or a tool for musicians to explore new ways of creating music. In conclusion, the system meets the design goals of the project as a working customizable audio kaleidoscope; it was also a very rewarding experience to design and create a working and entertaining device from a simple idea.