# ECE 385

Spring 2019

Final Project Report

MIDI-Compatible Synthesizer with Parameterizable Effects

Matthew Bremer, Justin Meyer

ABC - Tuesday 11:30 AM

Zhenhong Liu, Gene Shiue

## Introduction

For our final project, we designed a MIDI controlled synthesizer that can play multiple different notes in four unique waveforms and each waveform can be combined with another.

There are two different play modes, Mono and Poly, in the Mono mode one note will play at a time while in Poly multiple will play at once if multiple are pressed. The Glide option can be enabled when in mono mode which lets a note flow into another note instead of the note taking over, this option has an adjustable glide speed.

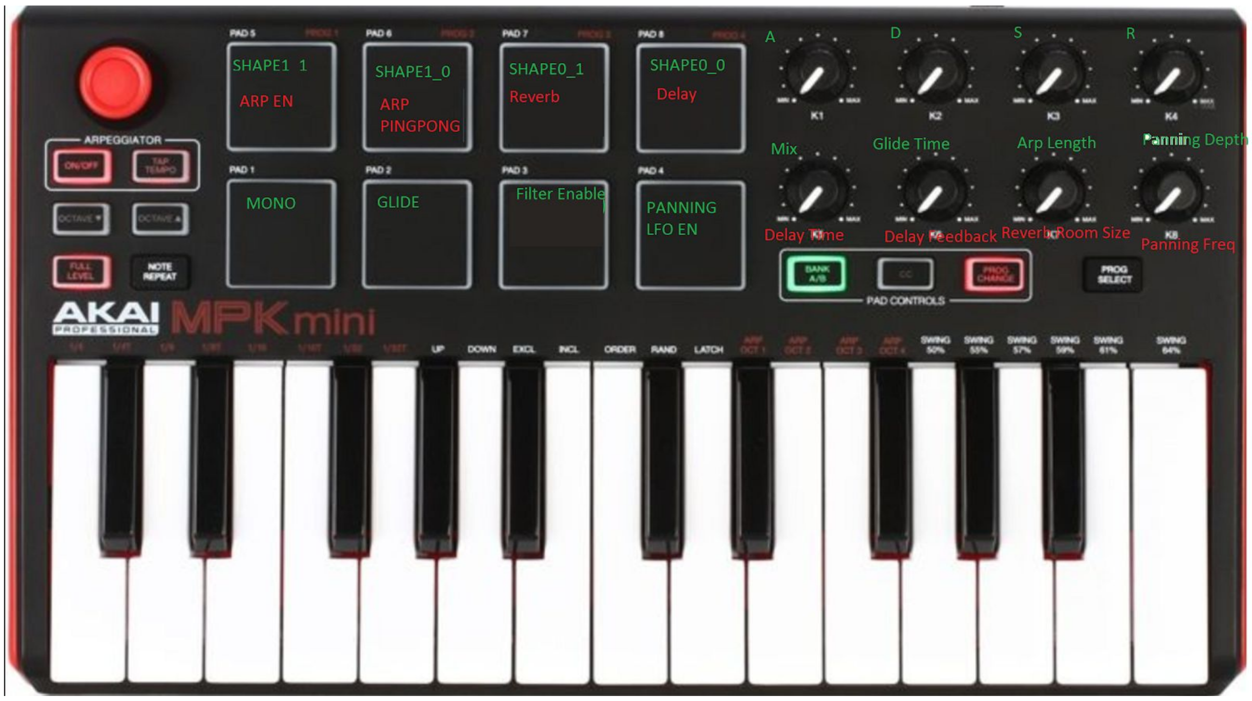
For pre-generation we added an Arpeggiator that while in Poly mode, plays all notes pressed one after another instead of at the same time, the order they play in depends on the order of the keys pressed the rate at which they are played is also adjustable. The arpeggiator also has a ping pong option where instead of just playing first to last and starting over it would go first to last and continue last to first over and over until deactivated.

For note control we have an adjustable ADSR that controls the rising, sustaining, and decreasing parts when pushing, holding, and releasing a note respectively, this will be explained further in the body.

For post-generation effects we added Panning, Reverb, and Delay. The Panning is an effect that switches the note from the left speaker to the right at a adjustable frequency, there is also an adjustable depth to control if the switching goes all the way left to right or stops some time before that. We also have the adjustable Reverb which makes a note sound as if coming from a larger room and the "room size" is adjustable. The next effect is Delay which plays a note multiple times after the initial press at an adjustable rate but it decays each time, which the amount of decay for each instance is also adjustable. All of these are mapped to different control buttons or turnable knobs on a Akai MPK Mini II controller.

## Operation of the Machine

The controller used for the whole project is the Akai MPK Mini II board pictured below. The notes are played on the keyboard keys and the control enables are on the center square pads where the green indicators are for Bank A, and the red indicators are for Bank B letting us map two controls to each button. The turnable knobs adjust the values for certain effects and the green indicators are for program 1 and the red are for program 2. The way to switch between the Banks is the Bank A/B button under "Pad Controls" and the way to switch the programs is by hitting the Prog Change button under "Pad Controls" and then hitting the square pad that corresponds to the same number. A button map is included in the picture directly below.



**Software Component**

Since the synthesizer was to be controlled by MIDI over USB, we adapted the given Lab 8 keyboard driver to instead communicate with our MIDI controller. First, the GET_CONFIGURATION messages were changed in order to allow us to read interfaces on the controller. Once the MIDI message endpoint was found, we simply had to change the OTG Transfer Descriptor to send a Bulk Transfer Request of size 0x20 on Endpoint 2. (The original sent Interrupt Transfer Requests of size 0x10 on Endpoint 0).

At this point our program can receive all MIDI messages from the controller. We utilize four types: NOTE_ON, NOTE_OFF, CONTROL_CHANGE, and PROGRAM_CHANGE. Program change messages are utilizes simply to change what internal controls are changed by which MIDI controls, since there are more of the former than the latter, so demultiplexing is necessary. All of the other messages, at their core, make changes to the static controls that lead from the NIOS II to the rest of the hardware. Since there are many static controls (27 Global Controls and 32 Per-Voice Controls), it was necessary to use a memory-mapped interface like the one used for AES Decryption. The interface uses a 64x32-bit register file to read and write from the NIOS II, and to provide static outputs to the rest of the hardware.

To change the Global Controls is a relatively easy task: On retrieval of a CONTROL_CHANGE message, update the control based on its new value (0x00-0x7f, given as part of the MIDI message), the current "program", and whatever mapping we decided was appropriate for the given control value.

To update the values controlling the voices themselves was a slightly more complicated task since part of our proposal was having polyphonic and monophonic playing modes, and so our synthesizer required "voice-stealing", the act of pre-emptying a note playing on one of the eight voices with a note that was just pressed, in the situation that all voices are occupied. In essence, our program keeps track of which keys are currently pressed, which oscillators are currently playing a key (in order of how long ago they were pressed), and which keys have been preempted and are waiting for an available voices. When a key is pressed, it looks for an available (not playing) voice. If it gets one, it activates it and is done. If not, it steals the voice of the key that's been pressed for longest, and forces that key on a wait queue. When a key is unpressed, if it is waiting, it simply removes itself from the wait queue. If not, it adds its voice back to the pool of available voices, and "wakes up" the most recently waiting key.

With this algorithm in place, all that is left is to add switching between polyphonic and monophonic modes. In the former, every voice is initially marked as available. In the latter, only one voice is marked as available, and every other pressed key besides the most recently pressed must wait for it.

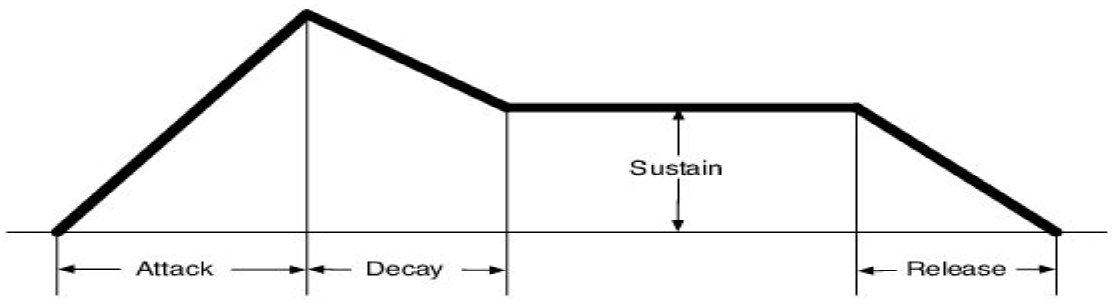
## Hardware Component

The static NIOS II controls lead into the following components: The eight voices each have unique controls for frequency (as MIDI note number), two amplitudes (one for each wave output), and *key_on*, a boolean representing whether the voice should be playing. They also have a number of shared controls: Two shape controls (for the two output shapes), attack, decay, sustain, and release (all parameters to the ADSR module), glide enable, and glide rate. In each voice besides the first, glide enable and rate are set to 0, since in glide mode, only the first voice is used. The outputs of these eight voices are summed, then fed through the following modules: Reverb, Delay and Panning. Each of these has an enable control as well as unique parameters fed through the NIOS, and are described below.

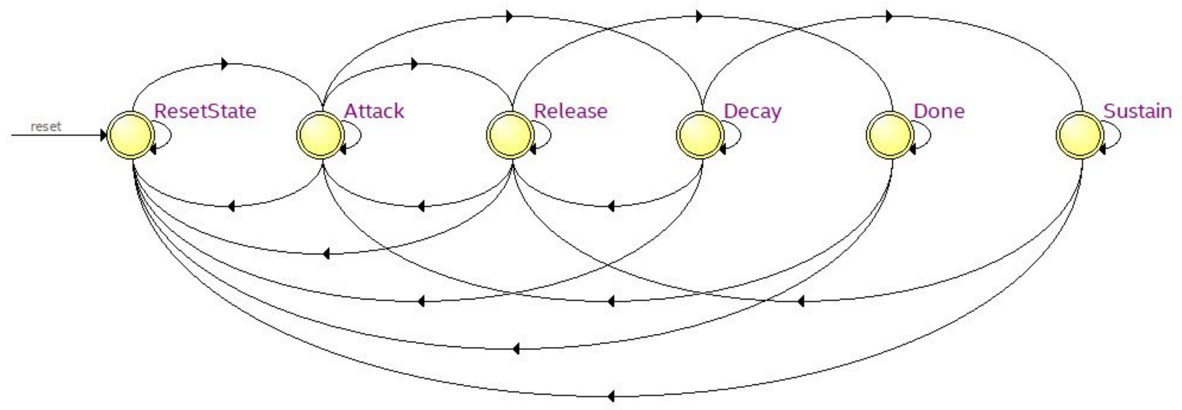
Each voice contains a Dual Oscillator and and ADSR, where the output is the product of the two, since the ADSR is an envelope generator. In addition, there is a MIDI note frequency lookup table, where the input is a MIDI note number *nn* and the output is the associated frequency: $(440/SR) * 2^{(69-nn)/12}$ Hz, where SR is the sampling rate of the hardware (48 kHz). There are also several glide modules repurposed as amplitude smoothers, that ensure that none of the amplitude envelopes (ADSR, Amplitude 1, Amplitude 0) change at a rate of more than 0x88 levels per sample. This was because a popping noise was initially observed when an amplitude value travelled its entire range (0x0000 to 0x7FFF) in less than 5 ms. To combat this, we choose a maximum rate of 0x7FFF levels/ (48000 samples/s * 0.005s) ≈ 0x88 levels/sample.

The dual oscillator itself has two parts: a digital integrator (also called a numerically controlled oscillator), and a double shape selector. The digital integrator simply takes a 24-bit frequency value and adds it to its phase over and over, wrapping around at the maximum value. This means that the digital integrator simply serves to go from its minimum to maximum value, at the specified frequency. Then, the top 12 bits of the phase are connected to the dual shape selector. Why does the integrator use 24 bits if it only needs 12? This is in order to maintain frequency accuracy - If only 12 bits were used in the phase, the quantization error in the frequency calculation would make the integrator noticeably out of tune. Finally, the shape selector works by feeding the phase into four different 4096x16 ROMs, each one of which is a wavetable, containing 1 cycle of a different waveshape generated in Python: Sine, Square, Sawtooth, or Triangle. Then, depending on the two selected shapes, two 4-to-1 multiplexers decide which table to collect a sample from. These are multiplied by their respective amplitudes and outputted into the ADSR module.

The ADSR module is a noise envelope that allows for control over the amplitude of the note played and make them audibly distinct from one another. ADSR stands for Attack, Decay, Sustain, and Release and the purpose of each section can be visualized by the graph below. The attack and decay allow for a release together makes it that the sustains of two different notes are audibly unique and separated.

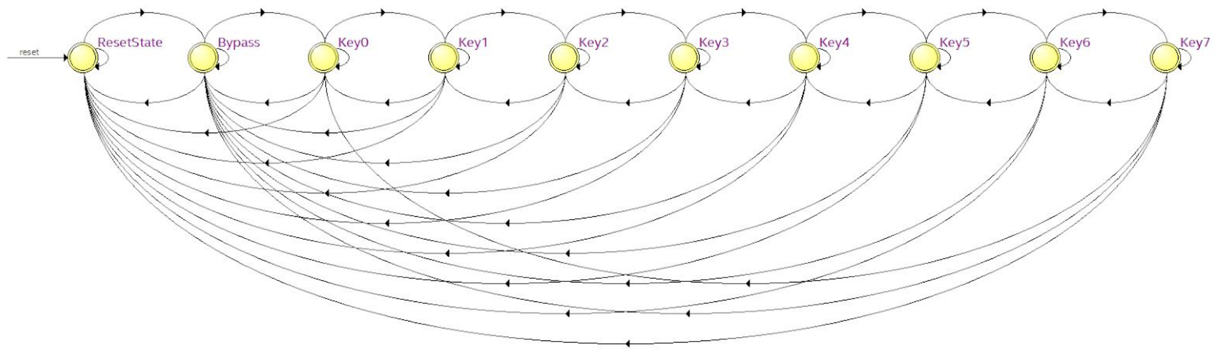


The way these are controlled are by a state machine that outputs an amplitude slope increases for each sample of a wave. The A, D, S, and R are the values that control the slope of each section through the inputs. The attack adds A to an input until the amplitude reaches the maximum value of 0x7FFF and D subtracts from the amplitude until it reaches the sustain value which is held until the key is released which is when it goes to the release state. The release state can be reached from any of the states as every state goes to release when a key off message is sent. At any point reset can also be activated to go to the resetstate and stay there until a new key is pressed.



The glide effect lets a note transition smoothly into another note in mono mode. The state module just has a reset state and a glide state where in glide the state checks if the target frequency is higher or lower than the current frequency and adds or subtracts the gliding rate from the frequency until the frequency is reached. The enable of this mode is external in the voice.sv module in a ternary operator. This behaves more closely to a register than a state machine.

The Arpeggiator is a state machine that can be used while in Poly mode and takes all the notes that would usually be played together and plays them one after the other in the order that the keys have been pressed. Once it reaches the last note it wraps around back to the first note and starts the cycle over. At any point keys can be released and it will play the keys that are left in their order. The Arpeggiator also has a Pingpong enable mode where once the last note pressed is reached, instead of starting from the beginning, will play them in reverse order until the first note is reached at which point it will be in the chronological order once more. The Arpeggiator has a bypass state in which it will sit, passing through every key on signal until the enable is pressed and will run through the states until a key is pressed. When Pingpong is disabled the state machine runs through all 8 note states checking for a key on signal and continuing if there is none. If there is a key on signal, it will stay in the state for a amount of time set by the user through the counter boundary that is attached to one of the control knobs. If Pingpong is enabled each state will check if the note being played is either the first or last note in the order by checking all the other key on signals above or below the state depending on the direction the order is traveling. If it is the first or last state, then the state will reverse the direction and immediately transition in that direction.  In each respective state, only the relevant key on signal is passed through the state machine. If while in any state, reset is pressed or the enable turns off, the state will immediately go to the resetstate or bypass state.



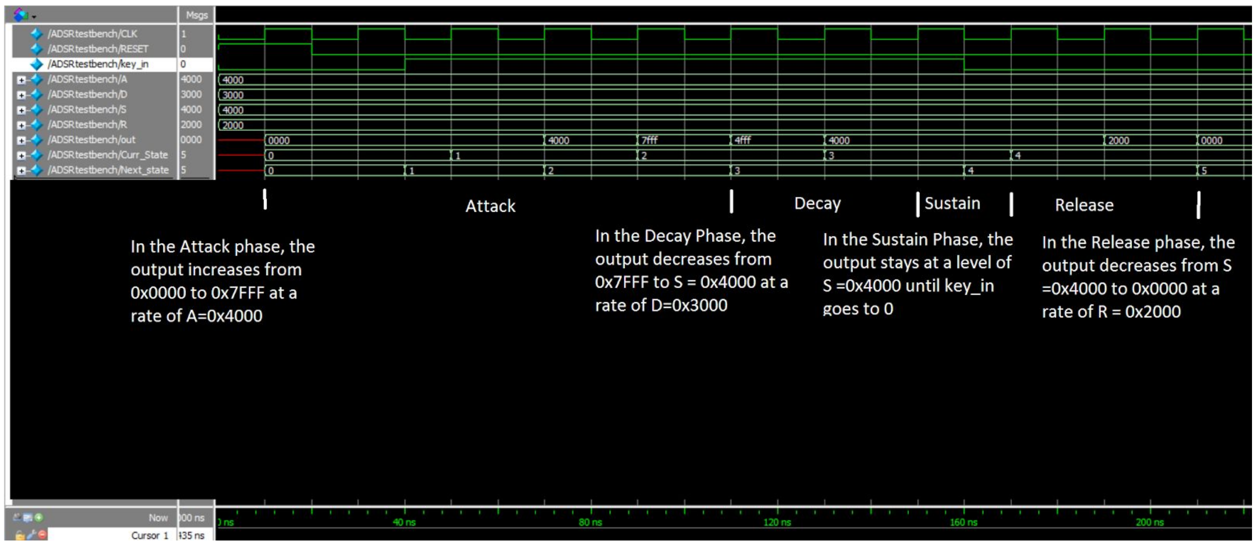
For the Panning module, the left output is run through the module that generates a very low frequency sine wave to multiply the output by. The right output is then multiplied by the max value 0x7FFF minus the height of the slow sine wave. While the enable is off, the exact middle value 0x4000 is used to rebalance the two outputs to be equal. While on, the sound will fluctuate between the left and right ear at the frequency of the sine wave which is adjustable. The depth of the variation can be modulated too to narrow the fluctuation, this can be controlled the Panning Depth knob on the controller.

The Delay and Reverb module work in very similar ways, both instantiate as RAMs that store the current value of the note pressed and play it back at a certain interval by using its own feedback to store the current state of the note. The difference is in Delay, the decay of the note can be controlled while it is set in Reverb. Another difference is that while both have controllable feedback times, the Reverb time has much smaller values in its range and a smaller range than the Delay module. And because the time is much shorter on Reverb, the note is replayed while it is still playing like a slight echo and thus does not need to be as accurate so only 960 samples per wave are stored rather than the 48K samples in delay.

**Simulations**

Many parts of the Synthesizer are useless to simulate, just because the output isn't human readable. Here are simulations for major modules utilizing state machines, with relatively human-readable outputs:

ADSR Simulation:



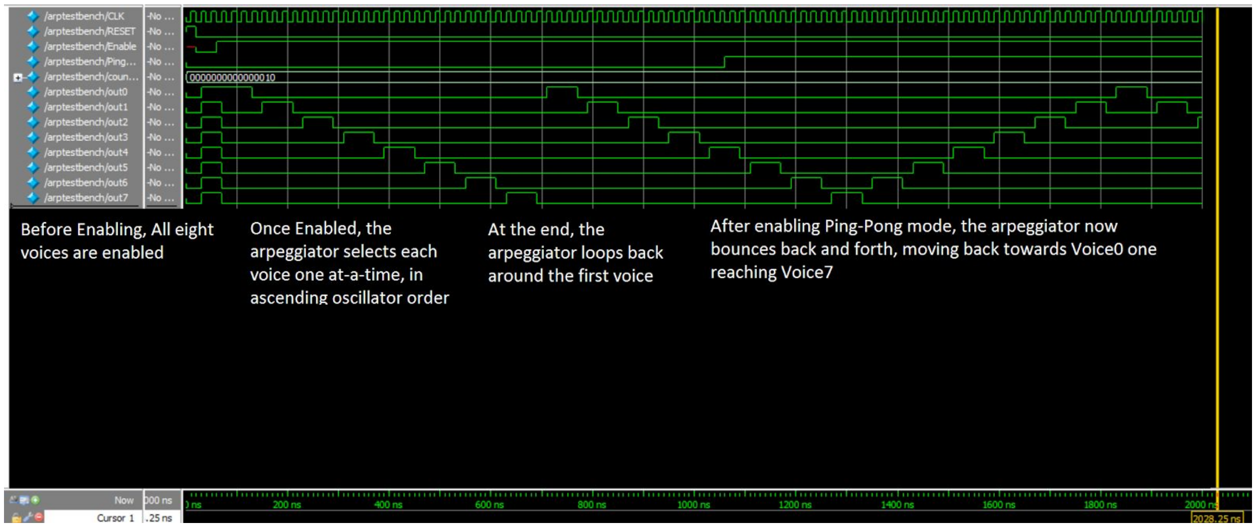
In the Attack phase, the output increases from 0x0000 to 0x7FFF at a rate of A=0x4000

In the Decay Phase, the output decreases from 0x7FFF to S = 0x4000 at a rate of D=0x3000

In the Sustain Phase, the output stays at a level of S =0x4000 until key_in goes to 0

In the Release phase, the output decreases from S =0x4000 to 0x0000 at a rate of R = 0x2000

Arpeggiator Simulation:



Before Enabling, All eight voices are enabled

Once Enabled, the arpeggiator selects each voice one at-a-time, in ascending oscillator order

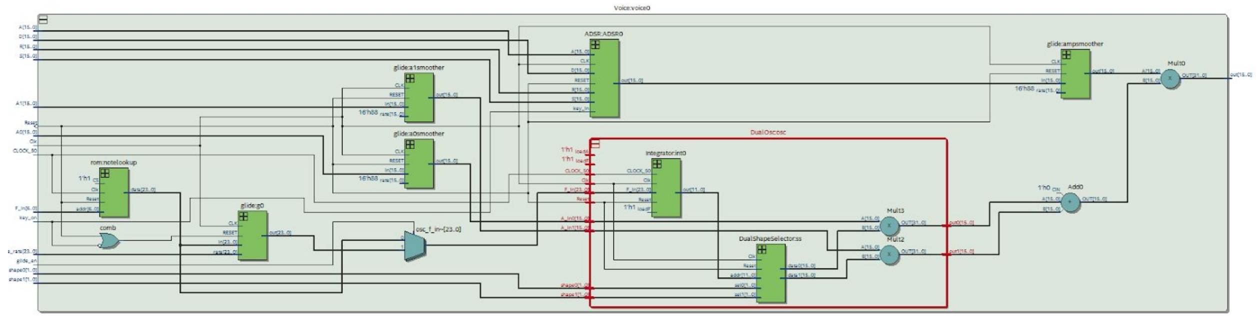
At the end, the arpeggiator loops back around the first voice

After enabling Ping-Pong mode, the arpeggiator now bounces back and forth, moving back towards Voice0 one reaching Voice7

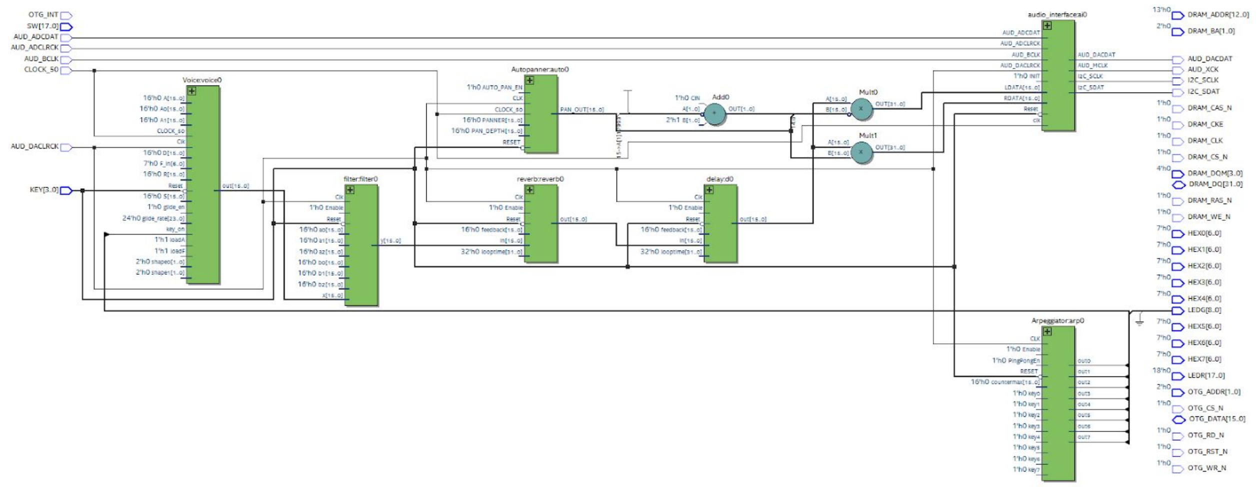
**Block Diagrams**

Due to the largely parallel nature of the connections between the NIOS and hardware, the top level by itself is not very useful. Here we begin at a basic level and add more until reaching the full top-level diagram.

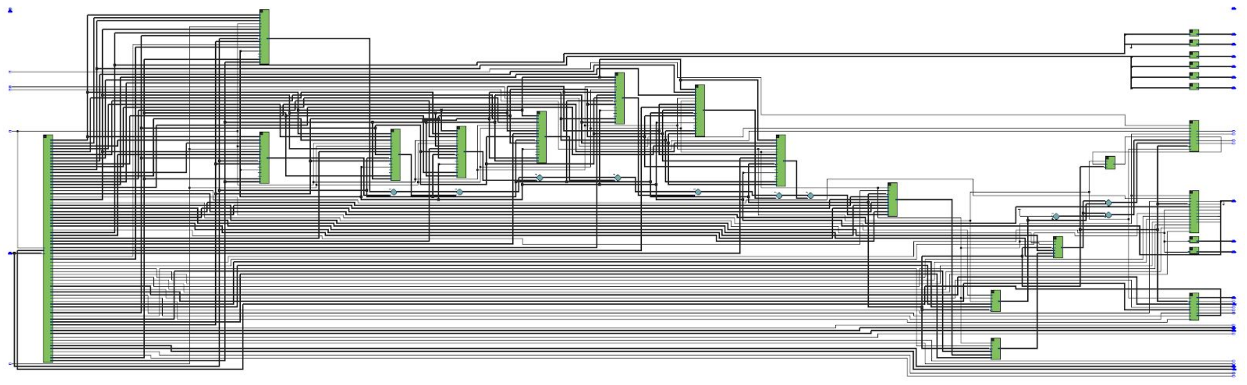
Block Diagram for a single voice:



Top Level with a single voice and no NIOS or audio driver:



Complete Top-Level with 8 Voices:



## Conclusion

The functions for our final projects sounded smooth and worked as intended during our presentation and we were able to implement all the function that were part of the proposal. The sound quality could be improved by adjusting ranges and multiplication coefficients but finding small differences for speakers that already were not perfect was unnecessary. The timeline we set in our proposal was followed in order but some of the more complicated parts such as the USB interface took extra time compared to some of the sound effects implemented. Memory is also a slight problem with this project since we reached after 85% capacity of virtual memory used after optimizing since our design at previous points did not fit on the board. This is due to the large look-up tables required to generate waveforms in Quartus.

## Design Resources and Statistics

| | |
|---|---|
| LUT | 12532 |
| DSP | 175 |
| Memory (BRAM) | 3,465,216 |
| Flip-Flop | 5605 |
| Frequency | 63.99 MHz |
| Static Power | 103.26 mW |
| Dynamic Power | 81.29 mW |
| Total Power | 263.05 mW |

## Appendix

**Module:** filter.sv
**Inputs:**Clk, Reset, Enabl, [15:0] (x, a0, a1, a2, b0, b1, b2)
**Outputs:** [15:0] y
**In-outs:** NA
**Description:** This is a set of postive edge triggered 16-bit registers that have an asynchronous Reset and Enable which when not enabled just pass through the x input.
**Purpose:** This module is used to create low pass filter although it was never enabled in our project as its behavior was not working correctly.

**Module:** integrator.sv
**Inputs:**Clk, CLOCK_50, Reset, loadF, [23:0] F_in
**Outputs:** [11:0] out
**In-outs:** NA
**Description:** This is a set of postive edge triggered 24-bit registers with an asynchronous reset and load enable bit.
**Purpose:** This module is used to create an address to send into a look-up table depending on input frequency and internal phase to create our oscillating waveforms.

**Module:** arptestbench.sv
**Inputs:**NA
**Outputs:** NA
**In-outs:** NA
**Description:** This is a testbench that simulates certain signals for Arpeggiator.sv
**Purpose:** This module is used to simulate and debug the Arpeggiator state machine.

**Module:** control_interface.sv
**Inputs:**CLK, RESET, [5:0] AVL_ADDR, [3:0] AVL_BYTE_EN, AVL_READ, AVL_WRITE, AVL_CS, [31:0] AVL_WRITEDATA
**Outputs:** [15:0] (ATTACK, RLEASE, SUSTAIN, DECAY), [1:0] (SHAPE1, SHAPE0), ARP_EN, GLIDE_EN, [15:0] (ARP_TIME, PANNING, PAN_DEPTH, REVERB_FEEDBACK), [24:0] GLIDE_RATE, PINGPONGEN, AUTO_PAN_EN, FREQ0, FREQ1, FREQ2, FREQ3, [15:0] (AMP1_0, AMP0_0, AMP1_1, AMP0_1, AMP1_2, AMP0_2, AMP1_3, AMP0_3), KEY3, KEY2, KEY1, KEY0, [6:0] (FREQ4, FREQ5, FREQ6, FREQ7), [15:0] (AMP1_4, AMP0_4, AMP1_5, AMP0_5, AMP1_6, AMP0_6, AMP1_7, AMP0_7), KEY4, KEY5, KEY6, KEY7, FILTER_EN, [15:0] (FILTER_A0, FILTER_A1, FILTER_A2, FILTER_B0, FILTER_B1, FILTER_B2), DELAY_EN, REVERB_EN,
[15:0] DELAY_FEEDBACK, [31:0] (DELAY_TIME, REVERB_TIME), [31:0] AVL_READDATA
**In-outs:** NA
**Description:** This is a 64x16-bit register array with a positive edge trigger and asynchronous reset.
**Purpose:** This module is used to store and output control signal values and enables from the NIOS II and make them usable by our modules.

**Module:** Shapeselector.sv
**Inputs:**Clk, Reset, [11:0] addr, [1:0] (sel0, sel1)
**Outputs:** [15:0] data1, data0
**In-outs:** NA

**Description:** This is a set of RAMs that use the input address to output either a square, sawtooth, triangle, or sine wave. It outputs two waves so that each oscillator in a voice gets one shape that is selected by two different 4:1 Muxes controlled by the select inputs.
**Purpose:** This module is used to feed waveform data to our oscillators.

**Module:** Muxes.sv
**Inputs:** [width-1:0] (A, B, C, D, E, F, G, H, ALU, MDR, PC, MARMUX, In), [3:0] sel, [2:0] sel, [1:0] sel, sel
**Outputs:** [width-1:0] (Out, A, B, C, D, E, F, G, H)
**Description:** This is a collection of different sized muxes, 2:1, 4:1, 8:1, and 16:1, plus a 1:8 DMux. Each one has a parameterizable data bit width and an appropriate amount of selector bits depending on the size of the mux
**Purpose:** This file is used to create the 4:1 Muxes in our shapeselector.sv

**Module:** synth.sv
**Inputs:** AUD_BCLK, AUD_ADCDAT, AUD_DACLRCK, AUD_ADCLRCK, CLOCK_50, [3:0] KEY, [17:0] SW,
**Outputs:** AUD_XCK, AUD_DACDAT, I2C_SDAT, I2C_SCLK, [8:0] LEDG, [17:0] LEDR, [6:0] (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7), [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK
**In-Outs**: [15:0] OTG_DATA, [31:0] DRAM_DQ
**Description:** This is a the module that instantiates our arpeggiator, voices, panner, delay, reverb, and filter modules, the HPI interface, our NIOS II, the audio driver interface, and hexdrivers.
**Purpose:** This file is our top level, used to instantiate all our circuit functions and output devices like the speakers and Hex displays.

**Module:** rom.sv
**Inputs:** Clk, Reset, CS, [ADDRWIDTH-1:0] addr,
**Outputs:** [WIDTH-1:0] data
**In-outs:** NA
**Description:** This is a positive edge triggered RAM with asynchronous reset with parameterizable size and bit width that reads one of our generated waveform files and outputs the information one data point at a time
**Purpose:** This module is used to create the different waveforms by giving out the shape data generated in python into a .mem file.

**Module:** register.sv
**Inputs:** Clk, Reset, Load, [WIDTH-1:0] D
**Outputs:** [WIDTH-1:0] Q
**In-outs:** NA
**Description:** This is a positive edge triggered register with a synchronous load and asynchronous reset that has parameterizable bit width
**Purpose:** This module is used to store and accumulate our waveform phase in integrator.sv

**Module:** NCO.sv
**Inputs:** Clk, CLOCK_50, Reset, loadF, loadA, key_on, [23:0] F_in, [15:0] A_in, [1:0] shape
**Outputs:** [15:0] out
**In-outs:** NA
**Description:** This is three register units from register.sv, two multipliers, and a rom.sv module.
**Purpose:** This module is used to create a single oscillator but was phased out and replaced by the DualOscillator.sv in the final version of the project.

**Module:** Initializer.sv
**Inputs:** INIT_FINISH, Clk, Reset
**Outputs:** INIT
**In-outs:** NA
**Description:** This is a positive edge triggered state machine with an asynchronous reset that outputs a single enable signal.
**Purpose:** This module is used to initialize the values in the audio_driver.sv

**Module:** hpi_io_intf.sv
**Inputs:** Clk, Reset, from_sw_r, from_sw_w, from_sw_cs, from_sw_reset, [1:0] from_sw_address, [15:0] from_sw_data_out
**Outputs:** [1:0] OTG_ADDR, OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N, [15:0] from_sw_data_out
**In-outs:** [15:0] OTG_DATA
**Description:** This is a positive edge triggered Latch with an asynchronous reset with a tristate buffer that controls OTG_DATA
**Purpose:** This module is used to connect the NIOS II and the EZ-OTG chip that lets the NIOS communicate with the USB devices.

**Module:** hexdriver.sv
**Inputs:** [3:0] In0

**Outputs:** [6:0] Out0
**Description:** This is the controller for the hex LEDs
**Purpose:** This module is used to display data on the Hex display.

**Module:** ADSR.sv
**Inputs:** CLK, RESET, key_in, [15:0] (A, D, S, R)
**Outputs:** [15:0] out
**Description:** This is the positive edge triggered state machine with an asynchronous reset
**Purpose:** This module is used to create the noise envelope for a note so it has an adjustable rise, fall, and sustained levels to be able to distinguish the notes easier.

**Module:** Glide.sv
**Inputs:** CLK, RESET, [WIDTH-1:0] in, rate
**Outputs:** [WIDTH-1:0] out
**Description:** This is the positive edge triggered state machine with an asynchronous reset
**Purpose:** This module is used to create a smooth noise transition between two notes while mono mode by increasing or decreasing towards the target frequency from the current frequency.

**Module:** Arpeggiator.sv
**Inputs:** key0, key1, key2, key3, key4, key5, key6, key7, CLK, RESET, Enable, PingPongEn, [15:0] countermax
**Outputs:** out0, out1, out2, out3, out4, out5, out6, out7
**Description:** This is the positive edge triggered state machine with an asynchronous reset and synchronous bypass disable bit.
**Purpose:** This module is used to play one note after another in the order the notes are pressed when enabled, it bounces back and forth when the Pingpong enable is turned on, and plays all notes at once in poly mode when disabled and thus bypassing the module.

**Module:** Dualosc.sv
**Inputs:** Clk, CLOCK_50, Reset, loadF, loadA, key_on, [23:0] F_in, [15:0] (A_in1, A_in0), [1:0] (shape1, shape0)
**Outputs:** [15:0] (out1, out0)
**Description:** This is two multipliers that multiplies waveform data by an input amplitude
**Purpose:** This module is used to create two waves of unique shape and amplitude but equal frequency.

**Module:** Autopanner.sv
**Inputs:** CLOCK_50, RESET, CLK, AUTO_PAN_EN, [15:0] (PANNER, PAN_DEPTH)
**Outputs:** [15:0] PAN_OUT

**Description:** This is a multiplier and an adder that use a input frequency to create a sine wave with an integrator and ROM that has an adjustable amplitude and frequency
**Purpose:** This module is used to create the panning effect by creating a mixing sound wave that throws the sound back and forth from the left to right speaker.

**Module:** delay.sv
**Inputs:** Clk, Reset, Enable, [15:0] in, [15:0] feedback, [31:0] looptime
**Outputs:** [15:0] out
**Description:** This is 16 register array that each hold 48k bits of data to save a complete waveform
**Purpose:** This module is used to create the delay effect that plays a note at an adjustable frequency and adjustable decay after the initial press.

**Module:** Reverb.sv
**Inputs:** Clk, Reset, Enable, [15:0] in, [15:0] feedback, [31:0] looptime
**Outputs:** [15:0] out
**Description:** This is 16 register array that each hold 960 bits of data to save a complete waveform
**Purpose:** This module is used to create the delay effect that plays a note as if it was being played in a larger room.