

Digital Supersaw Synthesizer

Jacob Brown, Isabelle Liu

6.111 Fall 2018

Introduction

In decades past, electronic music was generated in the analog realm. Audio synthesizers were expensive, heavy, and prone to the same problems as in other integrated circuits. With the increase in transistor density and refinement of the processor, more and more complicated tasks are being performed in the digital realm. There are many software programs meant to perform the same functions as their analog predecessors, but few all-digital synthesizers have been created that satisfy the electronic music creators. Indeed, musicians are hard to please.

We have designed a multi-voice, configurable digital synthesizer, allowing a user to play notes/melodies on a USB MIDI keyboard, adjust for desired sound features on a separate control panel, and hear the synthesized 96 kHz, 16-bit stereo result through a standard audio interface. In addition, we display the output waveform and control settings in real-time over 12-bit VGA to provide visual feedback. The design is based off the commercially-available JP6K VST synthesizer plugin, which emulates the analog "Supersaw" waveform known in the EDM industry as the de-facto standard for trance leads. The Supersaw algorithm originated on the Roland JP8000 analog modeling synthesizer and consists of seven detuned and phase-shifted sawtooth waves, where the inter-set mix level is adjustable and the intra-set phase-shift amounts are random [1]. The number of voices (simultaneous notes) is set to 8 as in the original Roland; for two channels, a total of up to 112 concurrent free-running oscillators are active.

Summary

The project was divided into two sections: audio and display. Jacob implemented audio, while Isabelle implemented display. These were merged together after we were each convinced of the stand-alone functionality of our respective sections. The overall utilization can be seen in Figure 1.

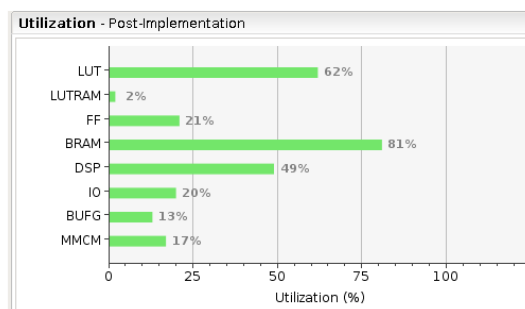


Figure 1: Utilization summary

The design process for the audio portion of the project began with a review of an analysis of the Roland JP8000 [1] [2] [3]. The sawtooth wave is very versatile and so our focus was to reproduce the Supersaw sound and not focus on sine, square, triangle, or mixed-mode waveform generation. Fortunately, the literature clearly quantified the sonic qualities of the Supersaw such that writing its implementation in a hardware description language (Verilog) was straight-forward.

Displaying a real-time waveform and control panel on a monitor involved a lot of experimenting with timing, debugging glitches, and evaluating memory requirements. We displayed the control panel using ROMs and had a screen buffer for waveform display as elaborated in later sections.

Block Diagram

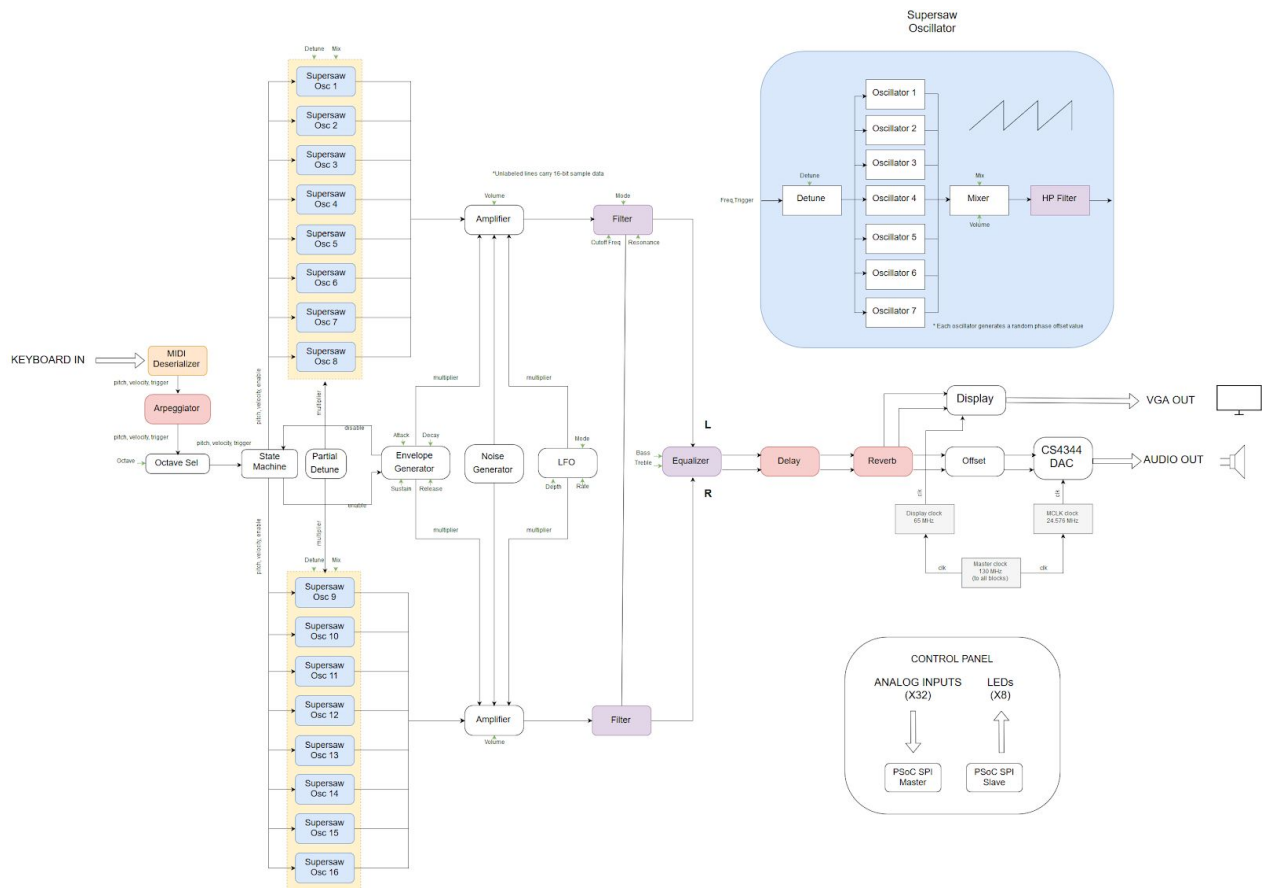


Figure 2: Block diagram

Goals

The project was divided into three achievement levels: commitment, primary, and stretch. The description of these levels are as follows. The goals that we have achieved in each category are underlined.

Commitment

The commitment goal includes everything needed to hear the basic Supersaw waveform in stereo and monitor it with the display. The following modules apply to this level: **PSoC 5LP ADC, ADC Host, Deserializer, State Machine, Oscillator, Supersaw** (to be further subdivided), **Amplifier, Offset, CS4344 DAC, and Control Panel Display**.

Primary

The primary goal includes the commitment modules plus the **Octave Select, Equalizer, Resonant Filter, Envelope Generator, LFO** and **Waveform Display** modules. This represents what we believe to be the most useful addons to the core of the project.

Stretch

The stretch goal includes all of the above plus the **Arpeggiator, Delay, Reverb, Noise Generator**, and **Global Detune** modules. These are features that have the potential to create the richest, most pleasing musical palette but which have an uncertain requirement on time investment.

MIDI Communication

This section describes the data flow and conversion from MIDI keyboard input to the Nexys4 FPGA. A USB MIDI keyboard is connected to a Windows PC, which performs the necessary handshaking and converts the USB signals into serial using the open source software Hairless MIDI, which is output to a USB-to-Serial adapter as used in Lab 2b. This adapter connects to a Digilent Pmod RS232 adapter that converts serial data to Pmod inputs which is finally fed into the Nexys4 Pmod input ports. A possible alternative is using an embedded USB Host microcontroller such as the CY7C63310, which would require additional programming but would eliminate the PC entirely. See Future Work for details.

Audio (Jacob)

A description of the audio (and supporting) modules, along with an explanation of their means of communicating to other modules, follows. It may be helpful to examine Figure 2 simultaneously to understand the interconnectivity and keep track of the signal flow.

PSoC 5LP ADC: Converts the analog signals from the various passive control elements (potentiometers, switches, and sliders) into 8-bit digital values to be interpreted through the PMOD inputs. Two Cypress PSoC 5LP modules were utilized: an SPI Master device which sends digital conversion results to the FPGA, and an SPI slave which drives appropriate LEDs on the control panel.

Deserializer: Converts raw serial data (sequences of 3-byte packets) from the Nexys4 PMOD port into (pitch, velocity) data to be sent to the **Octave Select** module.

Arpeggiator: Staggers all active notes to produce an arpeggio. Takes input from the **Deserializer** module and **ADC Host** and outputs raw note data (pitch, velocity) to the **Octave Select** module according to an internal sequencing algorithm.

Octave Select: Scales the pitch information up or down by one or two octaves (multiples of 12 semitones). Takes input from the **Deserializer** module (or the **Arpeggiator**, if present) and **ADC Host** and outputs raw note data (pitch, velocity) to the **State Machine** module.

State Machine: Keeps track of how many notes are active based on incoming (pitch, velocity) data and feedback from an ADSR envelope generator. Sends (pitch, velocity) and enable signals to the **Supersaw** modules; enable signals to the **Envelope** module.

Supersaw: Produces a set of seven detuned and phase-shifted sawtooth waves corresponding to a single fundamental frequency. Takes (pitch, velocity) and enable inputs from the **State Machine** module and parameter inputs from the **ADC Host**. Converts MIDI pitch values into oscillator increment values, and instantiates seven sawtooth **Oscillator** modules with (increment, enable) inputs. Mixes the outputs of each **Oscillator** module in relation to the base frequency. Scales the amplitude according to velocity data and parameter input, and finally applies a high-pass filter at the base frequency with an FIR/IIR algorithm, which requires a 127-column memory lookup. Outputs a new 16-bit sample on each sample clock to one of two **Amplifier** modules.

Oscillator: Generates a single 16-bit sawtooth wave. Takes (increment, enable) inputs from a **Supersaw** module and performs integer addition at a fixed time interval. Outputs a valid 16-bit sample on every clock cycle, but only updates the value after 9358 clock cycles (this value was chosen as a tradeoff between bit depth and detune accuracy).

Noise Generator: Generates white noise and sends it to the **Amplifier** modules, where it is mixed into the output of both **Supersaw** modules. Takes parameter input from the **ADC Host**.

Global Detune: Detunes the two **Supersaw** modules by up to 10% with respect to each other. Takes parameter input from the **ADC Host** and passes a multiplier value to each **Supersaw** module.

Envelope Generator: Tracks the rise and fall of each note and modifies it according to an ADSR curve. Generates an amplitude envelope, applied at the trigger event of each note. Takes a set of enable signal inputs from the **State Machine** module and ADSR (Attack, Decay, Sustain, Release) inputs from the **ADC Host**. Outputs 1-8 disable signals to the **State Machine** module, along with 1-8 multiplier values to both **Amplifier** modules, corresponding to the output of each **Supersaw** module.

LFO: Produces an amplitude envelope that varies in time and which is applied to all oscillators simultaneously. Takes input from the **ADC Host** and, preferably using IP cores, generates multiple envelope waveforms including sine, triangle, square, and sawtooth types; these waves oscillate from 0-20 Hz. Outputs a single multiplier value to both **Amplifier** modules.

Amplifier: Mixes the output of up to eight **Supersaw** modules into a single waveform. Takes input from the **Supersaw** modules, parameter inputs from the **ADC Host**, white noise from the **Noise Generator** module (if present), and scaling information from the **Envelope** and **LFO** modules. Outputs a new 16-bit sample on each sample clock to one of two **Filter** modules.

Filter: Applies an FIR/IIR filter (either high-pass or low-pass) to the signal. The filter order is fixed, but inputs from the **ADC Host** determine the cutoff frequency and resonance of the filter. Samples arrive from one of two **Amplifier** modules and are sent to one of two **Equalizer** modules. The same filter coefficients are used in both instances of this module. Requires a memory lookup from up to $128 * 128$ columns of coefficients.

Equalizer: Applies two FIR/IIR shelving filters in series to the output of one of two **Filter** modules. Takes sample input from the **Filter** module and parameter input from the **ADC Host** to approximate bass response and treble response. The coefficients are also shared between both instances of this module. Requires a memory lookup of up to $128 + 128$ coefficients. Outputs sample data to the **Offset** and **Display** modules (or **Delay** module, if present).

Delay: Produces an echo effect by delaying the mixed, filtered, and equalized oscillator outputs. Takes sample input from an instance of the **Equalizer** module along with parameter inputs from the **ADC Host** and outputs sample data to the **Offset** and **Display** modules (or **Reverb** module, if present). Stores up to two minutes (~44MiB for two instances) of the incoming sample data to memory and recalls, attenuates, and mixes together the original waveform with its delayed components.

Reverb: Produces a reverberation effect by applying an FIR/IIR model to the mixed, filtered, and equalized oscillator outputs. Takes sample input from an instance of the **Equalizer** module (or **Delay** module, if present) along with parameter inputs from the **ADC Host** and outputs sample data to the **Offset** and **Display** modules. Stores up to 30 seconds (~11MiB for two instances) of the incoming sample data to memory to be convolved by up to 128 different impulse response coefficients, corresponding to the varying Decay parameter. The Predelay parameter determines how many samples to delay the incoming data by and introduces another memory requirement of up to 375KiB for one second of predelay. Finally, the Mix parameter sets the relative mix level of processed to unprocessed sound.

Offset: Removes the DC offset of the output waveforms to allow for proper DAC operation. Takes input from an instance of the **Equalizer** module (or **Delay/Reverb**, if present) and produces a signed sample output centered at zero. This is simply a subtraction of 32,768 from the unsigned integer value.

CS4344 DAC: Converts the digital signals from the two channels into analog signals at an appropriate voltage to drive a line-level stereo audio output jack. This is a PMOD accessory available for purchase from Digilent, and is necessary for stereo audio, as the Nexys4's audio jack is mono only. A separate clock domain was created for this module (24.576 MHz) using an IP Core.

Display (Isabelle)

My work focused on producing video output to be displayed on an attached VGA monitor running at 1024x768 pixels x 60 Hz refresh rate. The top half of the screen shows the left supersaw output and takes inputs from all **ADC Host** channels and sample data to draw in real-time the waveform corresponding to one channel. The bottom half is the control panel of the synthesizer to provide visual feedback in the style of the Roland JP-8000 analog modeling synthesizer to show the current state of the parameters.



Figure 3: Picture of screen at idle

Control Panel

We designed the control panel in Adobe Illustrator based off the JP6K GUI. The panel had originally 8 bit color depth but the Nexys4 VGA only has 12 bit color, i.e. 4 bit color depth for red, green and blue respectively. To accommodate this limitation, We wrote our own MATLAB script to create image ROMs and colour ROMs based on the script used in lab 3. The script is included in Appendix A. Many implementation challenges were encountered as listed below.

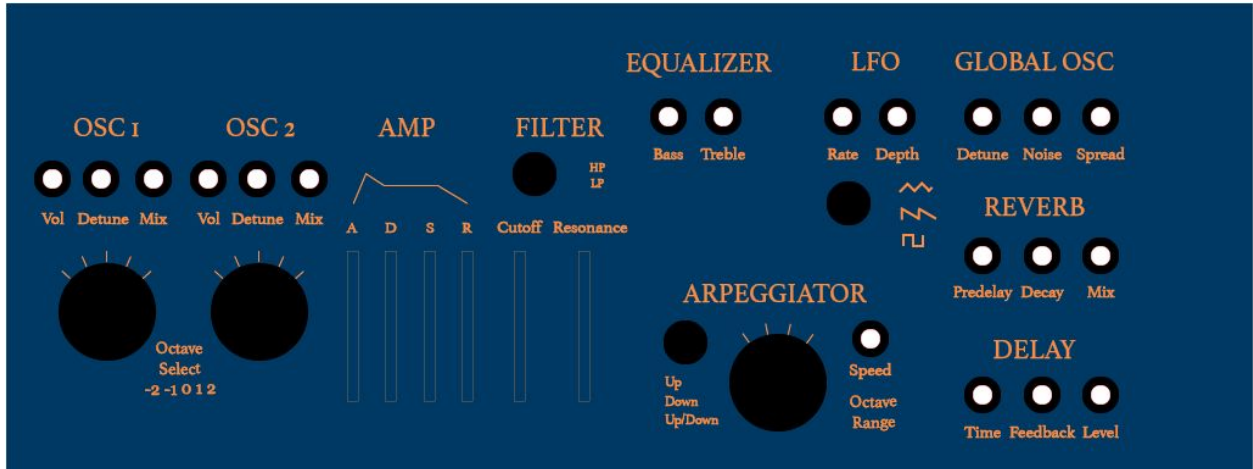


Figure 4: Control panel designed in Illustrator and used to create ROM

Display Timing

First of all, we realized that directly outputting a RGB pixel from image ROM and colour ROM required at least 4 clock delays, so hsync, vsync and hblank had to be delayed accordingly to avoid glitching. Furthermore, when displaying other objects that didn't have delays in generating pixel values (e.g. a blob object from Lab 3), the hcount bounds had to be adjusted so these objects would display at the same time as the ROM objects. For example, the red slider controls were assigned wires, so they had no clock delay in generating pixel values. To make these sliders display in sync with the control panel pixels, the sliders would display when $hcount \geq (x + delay)$ and $hcount \leq (x + slider_width + delay)$. Other calculations that generated pixel values faster had to be pipelined(shift registered) such that the pixel value would be delayed to output at the same time as the the ROM pixels.

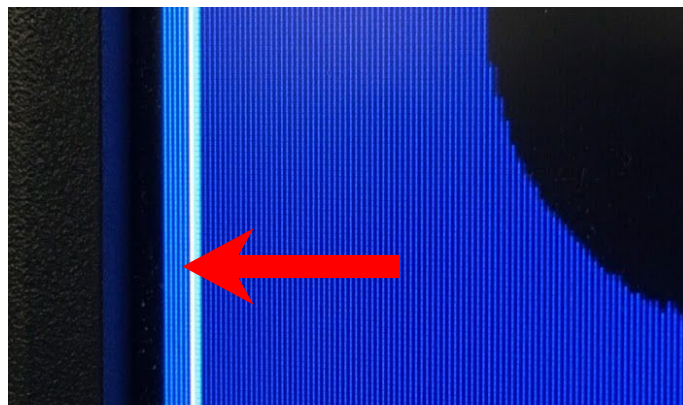


Figure 5: Example of glitching

Knob Lines

To display the potentiometer values on the screen in realtime, several methods were considered including calculating the angle in realtime and displaying a line with that angle, rotating the whole knob picture, having several ROMs each with a different angle and swapping them out, and comparing pixels with binary maps to determine what value the pixel would take. We decided to compare pixel values with binary maps because it required less memory than storing huge trig tables or ROMs and had the least delay in generating pixel values.

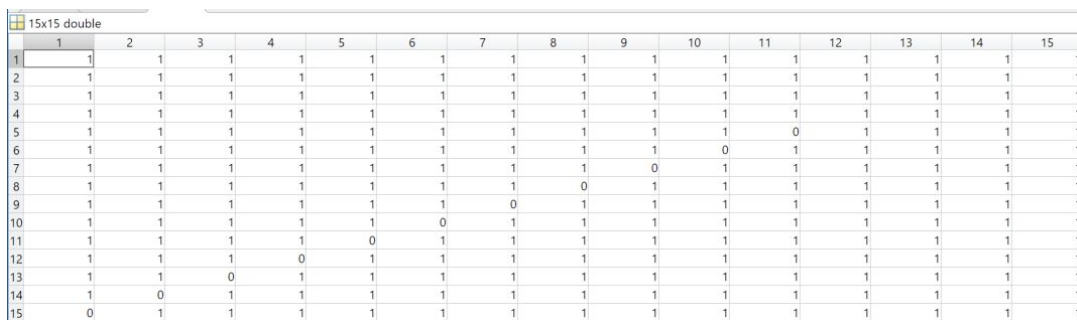
As hcount and vcount crosses the potentiometer region, the pixel value will be decided by a map. The map would consist of 1s and 0s, with 0 making the pixel black and 1 making the pixel white. If the pixel is white, it will be the same as the background; if it is black the pixel will become part of the “knob line” that displays what value the potentiometer has. In addition, there would be one map for each angle.



Figure 6: Example of potentiometers displayed with the black line (knob line) indicating the current value

Each pot is 30x30 pixels wide. However, to save memory we decided the maps only needed to be 15x15 pixel wide and could be indexed into differently depending on the quadrant, creating the effect of mirroring the knob lines around the x and y axis.

We proceeded to create these 15x15 pixel “binary maps” in MATLAB as shown in Figure 7. Next, we experimented with maps of several different angles, to see what would be the optimal angle difference between each knob line. Having 22.5 deg angle between every line was optimal because it had enough resolution (four different angles per quadrant) without taking up too much memory. We created a map each for 22.5 deg, 45 deg, 67.5 deg and 90 deg. These maps were stored as 225 bit wires. Considering the physical angle limits on the pots were around 300 deg, 13 different possible angles were displayed as drawn in Figure 8. The potentiometer input to my modules were 7 bit wide (128 bit) so roughly each line corresponded to increments of 10, as coded in panel_input.v. Other challenges include debugging glitches caused by timing issues. Hcount bounds had to be modified to avoid glitching.

A screenshot of a MATLAB window showing a 15x15 double matrix. The matrix contains binary values (0 and 1) arranged in a pattern that represents a 45-degree knob line. The values are 1 for most pixels, with 0s forming a diagonal line from the top-left to the bottom-right.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
6	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1
7	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
8	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
9	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
10	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
11	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
12	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
13	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
14	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
15	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 7: MATLAB table of map for 45 degree knob line

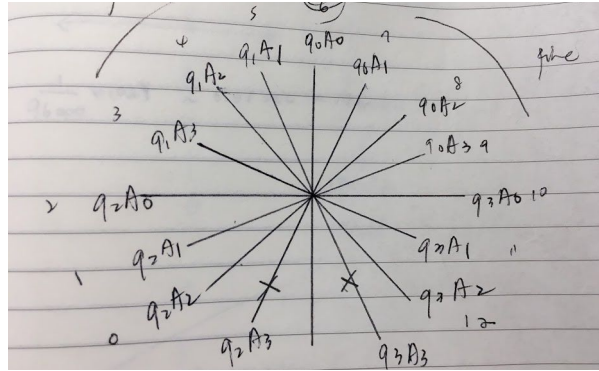


Figure 8: Drawing of the angles/lines that were displayed

Rotary Switches

The approach to displaying the rotary switch values were similar to the displaying the knob line of pots. The map size was 40x80 px to fit the size of the switches as displayed.

Waveform

The top half of the screen displayed real-time waveform of the left channel audio output. The waveform was captured with a temporary screen buffer of 1024 bits to fit the screen width of 1024 pixels. Once the temporary buffer had collected 1024 new audio samples, the whole temporary buffer was transferred to the actual screen buffer displayed on screen.

The audio output was between 20 - 20 kHz, so to avoid aliasing we chose the sampling rate to be $65 \text{ MHz} / 1024 = 63.47 \text{ kHz}$. In addition, the raw audio output was 16 bits wide whereas the top half of the screen was 348 pixels tall, i.e could only display values up to 348. So, to show the amplitude more clearly, each sample was sized to 8 bits, minus by an offset of 64, multiplied by 2.

Physical Control Panel (Isabelle, Jacob)

Isabelle designed the first laser cut file in Illustrator based on the Control Panel in Display. Together we laser cut $\frac{1}{8}$ " acrylic panels in EDS. Several revisions in Adobe Illustrator were later needed to optimize the layout. Jacob soldered the hardware/PSoCs to the panel.



Figure 9: Photo of control panel, Nexys4 FPGA, and PMODs

Implementation Hints

There were many challenging implementation details associated with our project, but here we summarize the key points to consider (for the audio-related modules). The first is that a sawtooth wave is very easy to generate in hardware; it consists simply of an add operation performed at a fixed time interval (see Figure 10). In Verilog fixed-point arithmetic, unsigned integers wrap around automatically, so all that is needed is to compute carefully the increments associated with each frequency (and its detune spectrum [1]) and shift the waveform down to a signed representation.

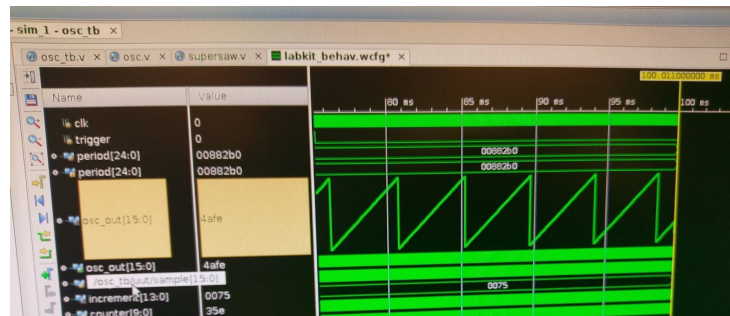


Figure 10: Screen capture of oscillator simulation in Vivado 2016

The mixing together of multiple waveforms (as in each Supersaw module, and each Amplifier module) was a challenging task, because it is desirable to maximize headroom/dynamic range and at the same time transitions between number of summed waveforms should not produce a jump in perceived volume. Multiple approaches were taken to solve this problem, but the solution we settled on was to perform a divide by the square root of the number of active waveforms (the perceived volume of summed non-coherent waveforms does not increase linearly) [4]. There may be more elegant solutions.

The filters can be created easily with the FIR Compiler IP Core (part of the Vivado suite) but MATLAB is needed to produce the coefficient sets [5]. We used the Control System Toolbox to generate frequency response vectors and the Signal Processing Toolbox to generate the FIR coefficients.

For implementation details on the display modules, see the section above.

Future Work

While we are very pleased with our accomplishment of reproducing the fabled Supersaw sound, we concede that much work remains to be done on this project. The remaining modules include the **Equalizer, LFO, Resonant Filter, Arpeggiator, Delay, Reverb, Noise Generator, Global**

Spread, and **Global Detune** modules. We would like to challenge any EDM fans or music aficionados to attempt them.

As we plan to finish these modules after 6.111 (IAP 2019), some thought has already been given to how they should be implemented. Specifically, the utilization of BRAM for the XC7A100T is close to its maximum (Figure 1), so we plan to implement an SD Card module to read in the necessary coefficients for the Resonant Filter module [5]. Code for this module is provided on the 6.111 class web site under Tools.

Another detail we would like to improve on is the response time of the oscillator module. Currently, a new sample is produced after 9358 clock cycles, which means that our choice of 96 kHz as the output sampling rate is a waste of resources for the audio DAC as well as the FIR filters. This is necessary given our choice to generate a 16-bit waveform directly, but an immense improvement could be seen by increasing the bit depth to 32-bit. The increment values per clock cycle would then be much more easily represented with integers, and accuracy for detuning small amounts would still be achieved.

Further, the Envelope Generator module could potentially be improved by adding logarithmic tapers to the attack, decay, and release curves [6]. While it was simple to code a linear increase or decrease in volume over time, extra ROMs may be needed for this.

Finally, it would be desirable to eliminate the laptop as the USB MIDI device passthrough. This should be a simple add-on to the project, since a dedicated Serial USB Host can be purchased pre-programmed to function with USB MIDI communication capability [7]. This would replace the PMOD RS232 adapter and allow for a more compact, all-in-one design. Minimal modifications would be required of the Deserializer module.

Possible additional features for the display may include sections of the screen to display the total number of active oscillators, memory usage, and/or pitch/frequency values.

References

1. Szabo, A. (2010) *How to Emulate the Super Saw*
https://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2010/rapporter10/szabo_adam_10_131.pdf
2. Roland Corp. (1996) *JP8000 Owner's Manual*
https://www.roland.com/us/support/by_product/jp-8000/owners_manuals/
3. KVR Audio, Inc. (2009-2018) *Roland Supersaw* (forum discussion)
<https://www.kvraudio.com/forum/viewtopic.php?t=258924>
4. DSP StackExchange (2012-2017) *Algorithm(s) to mix audio signals without clipping* (forum discussion)
<https://dsp.stackexchange.com/questions/3581/algorithms-to-mix-audio-signals-without-clipping>

5. Xilinx, Inc. (2013-2015) *Vivado FIR Compiler v.7.2 Product Guide*
https://www.xilinx.com/support/documentation/ip_documentation/fir_compiler/v7_2/pg149-fir-compiler.pdf
6. Electronics StackExchange (2017) *Formula for Logarithmic (audio taper) pot* (forum discussion)
<https://electronics.stackexchange.com/questions/304692/formula-for-logarithmic-audio-taper-pot>
7. HobbyTronics, Inc () *USB Host Controller Board V2.4*
<http://www.hobbytronics.co.uk/usb-host-board-v2?keyword=usb%20host>

Appendix A : MATLAB Script for creating ROMS for 12 bit VGA

Image ROM Script

```

clear all
[picture] = imread('roland_v4_4bit.bmp');
picture_size = size(picture);    %figure out how big the image is
num_rows = picture_size(1);
num_columns = picture_size(2);
pixel_columns = zeros(picture_size(1)*picture_size(2),1); %pre-allocate a space for a new
column vector

for r = 1:num_rows
    for c = 1:num_columns
        pixel_columns((r-1)*num_columns+c) = picture(r,c);    %pixel# = (y*numColumns)+x
    end
end

%so now pixel_columns is a column vector of the pixel values in the image
rounded_data = round(pixel_columns); %rounds them down
data = dec2bin(rounded_data,4);    %convert the binary data to 4 bit binary #s

%open a file
output_name = 'panel_image.coe';
file = fopen(output_name,'w');

%write the header info
fprintf(file,'memory_initialization_radix=2;\n');
fprintf(file,'memory_initialization_vector=\n');
fclose(file);

%put commas in the data

```

```

rowxcolumn = size(data);
rows = rowxcolumn(1);
columns = rowxcolumn(2);
output = data;
for i = 1:(rows-1)
    output(i,(columns+1)) = ',';
end
output(rows,(columns+1)) = ',';

```

```

%append the numeric values to the file
dlmwrite(output_name,output,'-append','delimiter','', 'newline', 'pc');

```

Color ROM

```

%Repeat once for each colour using corresponding section

```

```

clear all

```

```

close all

```

```

[picture color_table] = imread('roland_v4_4bit.bmp');

```

```

figure

```

```

image(picture)

```

```

colormap(color_table)

```

```

colorbar

```

```

title('4 bit bitmap displayed using color table')

```

```

% red = color_table(:,1);          %grabs the red part of the colortable

```

```

% scaled_data = red*15;           %scales the floats back to 0-15

```

```

% green = color_table(:,2);       %grabs the green part of the colortable

```

```

% scaled_data= green*15;          %scales the floats back to 0-15

```

```

blue = color_table(:,3);          %grabs the blue part of the colortable

```

```

scaled_data = blue*15;            %scales the floats back to 0-15

```

```

rounded_data = round(scaled_data); %rounds them down

```

```

data = dec2bin(rounded_data,4);    %convert the binary data to 4 bit binary #s

```

```

%output_name = 'panel_red.coe'; %open a file

```

```

%output_name = 'panel_green.coe'; %open a file

```

```

output_name = 'panel_blue.coe'; %open a file

```

```

file = fopen(output_name,'w');

```

```

%write the header info

```

```

fprintf(file,'memory_initialization_radix=2;\n');

```

```
fprintf(file,'memory_initialization_vector=\n');  
fclose(file);
```

```
%put commas in the data  
rowxcolumn = size(data);  
rows = rowxcolumn(1);  
columns = rowxcolumn(2);  
output = data;  
for i = 1:(rows-1)  
    output(i,(columns+1)) = ',';  
end  
output(rows,(columns+1)) = ',';
```

```
%append the numeric values to the file  
dlmwrite(output_name,output,'-append','delimiter',' ','newline','pc');
```