cooking beats from scratch

A Cookbook for Digital Audio Synthesis on the Zybo Z7-20

Dylan Sandall

03/17/2025
CPE-542

ABSTRACT

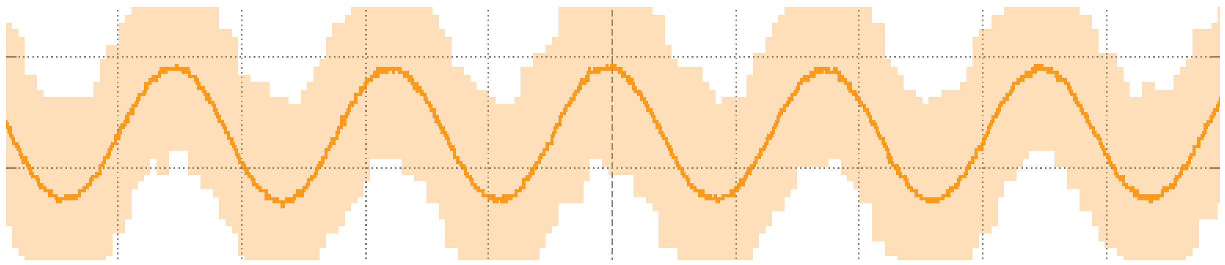
While it's easy to take audio for granted, real time calculation of audio effects can be a computationally intensive process. Something like a low-pass filter requires many multiplication operations per sample, and when working on projects with high sample rate or many audio sources, quickly add up. Luckily, these operations are often static and lend themselves well to digital (or analog!) hardware. For these reasons, hardware acceleration of audio creation and other DSP is a great tool to have under your belt.

In this cookbook, I'll discuss the high level ideas behind audio synthesis, as well as how they can be implemented with FPGA and SoC devices in mind. By the end, you should be able to generate your own waveforms, free of aliasing, clipping, and with runtime-variable playback frequencies.

ANALOG SYNTHESIS *- Grandma's Recipe*

>> Our first Ingredient - Sine Wave

Audio, as you and I hear it, is just pressure waves in the air wiggling the fleshy bits in your inner ear. This allows you to perceive sound frequencies all the way up to 20 kHz - meaning that the pressure rises and falls 20,000 times in a second. The rate at which the air is moving is perceived as a *pitch*, and the total amount the air pressure changes is the *volume*. By precisely controlling the pressure of the air around your ears, with something like a speaker, the perception of sound can be created.



A sine wave is the fundamental type of wave, and it cleanly represents a single frequency (read: Fourier Transform, time-frequency duality). Waves are great, but we often like to think of things in seconds, or countable increments. For this reason, we also like to break our waves into discrete chunks. While this is not doing the subject justice, you can take three things from this:

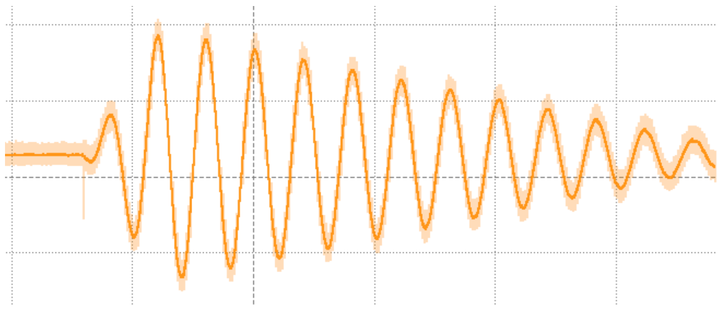
1. any complex waveform can be modelled by a collection of sine waves with 2 parameters, magnitude and frequency[1]; ie **a frequency distribution.**
2. Any complex waveform can be modelled by a **series of discrete samples** with 2 parameters, magnitude and time.
3. All sound can be thought of as a complex waveform, or either of the above

>> I'm sorry, I thought we were cooking beats?

We are, but a critical part of cooking is planting the seeds and watering the crop. Assuming you've done your homework on waves, let's relate this to something a little more… fresh.

[1] Phase is also important, and can make a big impact on audio.

MISE EN PLACE *- Chop, prep, and marinate. Let's get the building blocks in place.*

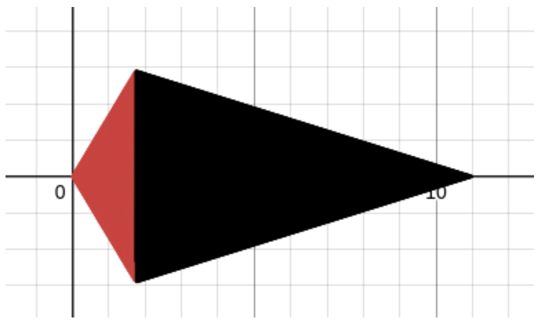


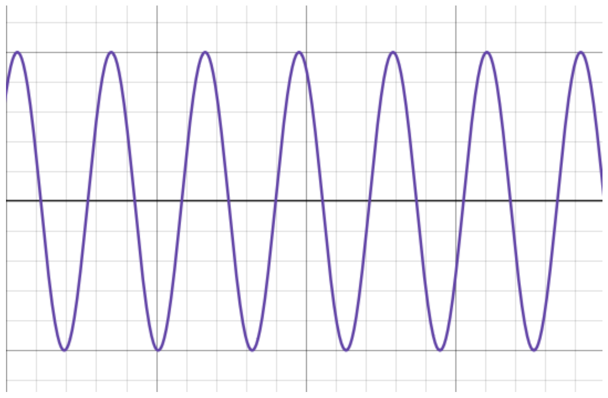
>> Our first dish: The 808 Kick Drum

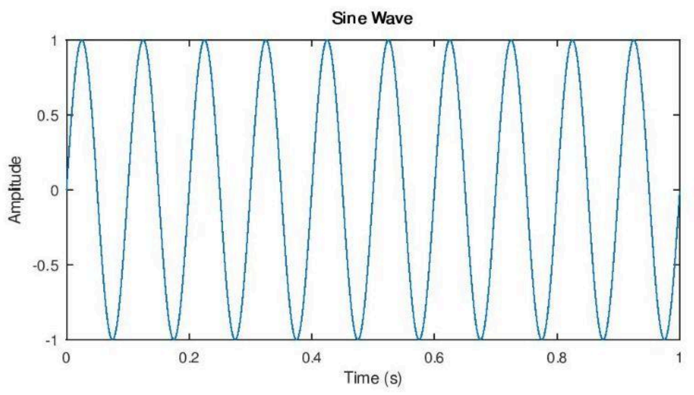
Do you recognize that sound? Of course! It's an 808 kick drum! Popularized by the Roland TR-808, this is a common kick drum for electronic and hip-hop music, and has thousands of variations. Let me talk a little about how we got this starting from a sine wave.

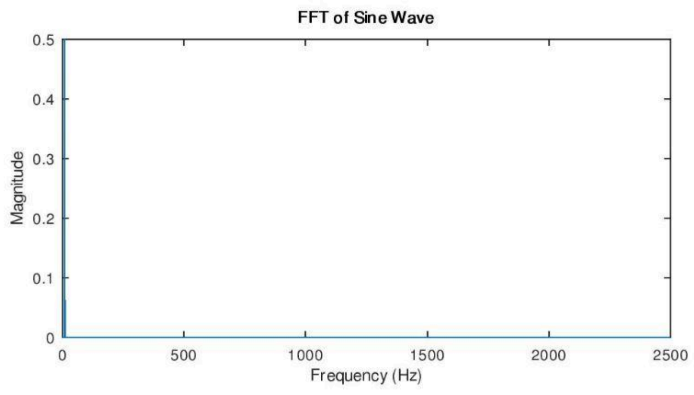
We start with an ***oscillator***, which produces a humble sine wave at a low frequency, somewhere in the range of 40-120 Hz. This gives us a nice and hearty flavor, as this is near the lowest tones we humans can hear. But the percussive nature of a drum means that the sound takes a moment to build and even longer to ring out and fall silent again. For this, we use ***envelopes***. Think of an envelope as a thing that controls a parameter for another device. The amplitude envelope shown here controls the volume of the sine wave over time. The amplitude envelope and oscillator for our 808 looks like this - a quick ***rise*** or ***attack*** (shown in red), and a slow ***fall*** or ***decay*** (shown in black).

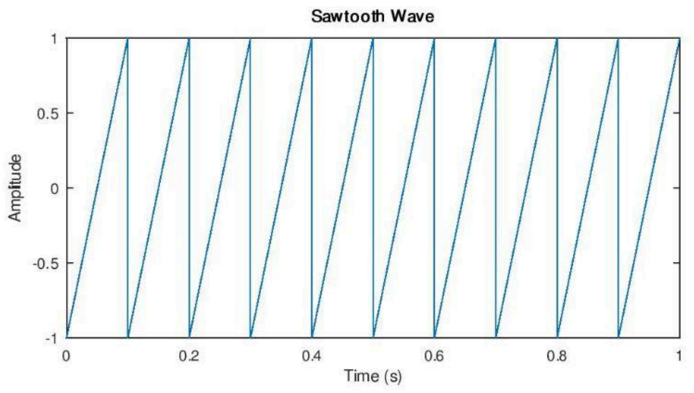
Envelopes are incredibly versatile, and can be used to control frequency, volume, distortion, filtering, or any other parameter. You could even use a sine wave as an envelope, if you wanted.

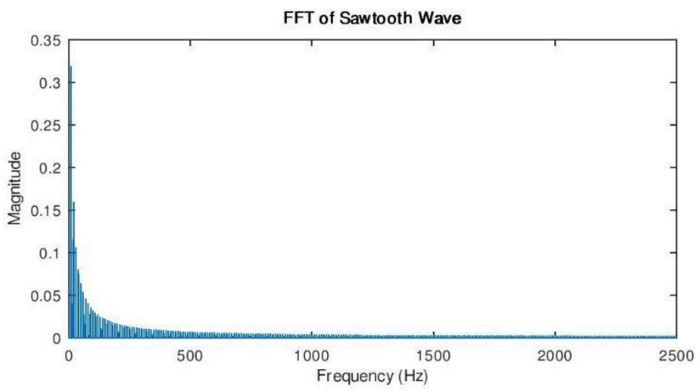


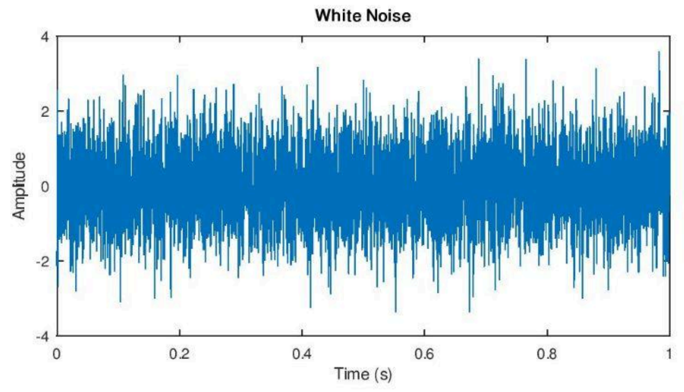


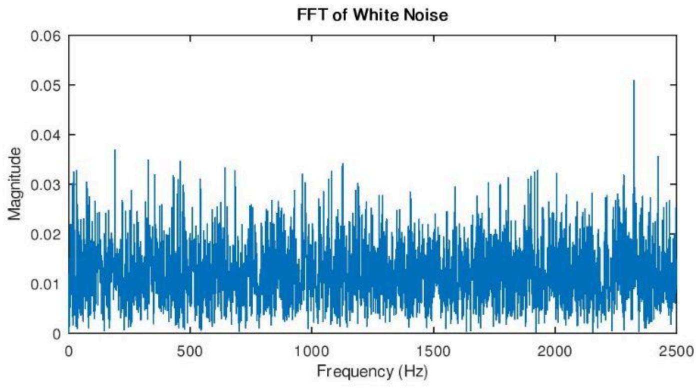












>> *A Little Spice - Harmonics and Waveshaping*

Sine waves are great - as mentioned, you can technically create any periodic waveform from scratch with only variations on sine. But some types of sounds are much easier to achieve with a different base. Much like chicken stock or bread, I much prefer to go to the grocery store. For this recipe, we're gonna need 3 total ingredients - sine, sawtooth, and noise. Above you can see the 3 waves in the time and frequency domain.

A sine wave sounds clean, and allows you the most control over your frequency distribution. It only creates a spike at your chosen frequency. A sawtooth wave is simple enough in the time domain, but is technically composed of a series of sine frequencies called ***harmonics.*** Harmonics are integer multiples of your base frequency, but the key thing to know is that they are always higher than your chosen frequency. To keep this short and sweet, a sawtooth wave will introduce some brightness to your dish - the high frequencies of the sawtooth wave will ring out in the higher notes and remain in tune with the lower base frequencies, without any unwanted dissonance you may get by sprinkling in higher notes by hand.

The final ingredient is **white noise** - known for its soothing and smothering effects, white noise is defined as “noise which has an equal distribution across the frequency spectrum” - what this means in practice is that white noise is not characterized by a single frequency, but all of them equally. White noise drowns out low and high frequencies with equal effectiveness, and is a good way to put some percussive, or rattling texture into your sound. Think of the sound of a maraca, or the sound of pouring dry cereal.

>> Packaging your Dish - Digitization

One of the beauties of audio is that it is completely non perishable - provided that you store it properly. If you want your canned digital audio to sound as fresh as the analog equivalent, there are a number of concerns - let me try to bring you up to speed.

The first is **aliasing** - I will not be explaining the arcane causes of this, but I highly encourage you to read up on the subject. The recipe for avoiding aliasing is as follows: Low pass your signal to remove anything higher than half your sample frequency. The next is **clipping** - this happens when your ideal waveform cannot be represented in the format chosen - it’s like blowing out your speakers and has the same fix, turning the volume down. A similar-sounding issue can happen when an audio source is immediately turned on or off, leading to **clicks and pops** from the sudden transition. This is what leads to the “tapping on my eardrum” effect when you unplug a live set of speakers.

We’ll talk more about this in our implementation details, but making use of envelope controllers and filtering prior to downsampling handle these common digitization issues.

INGREDIENTS *- This is just what I had on hand. Feel free to spice it up!*

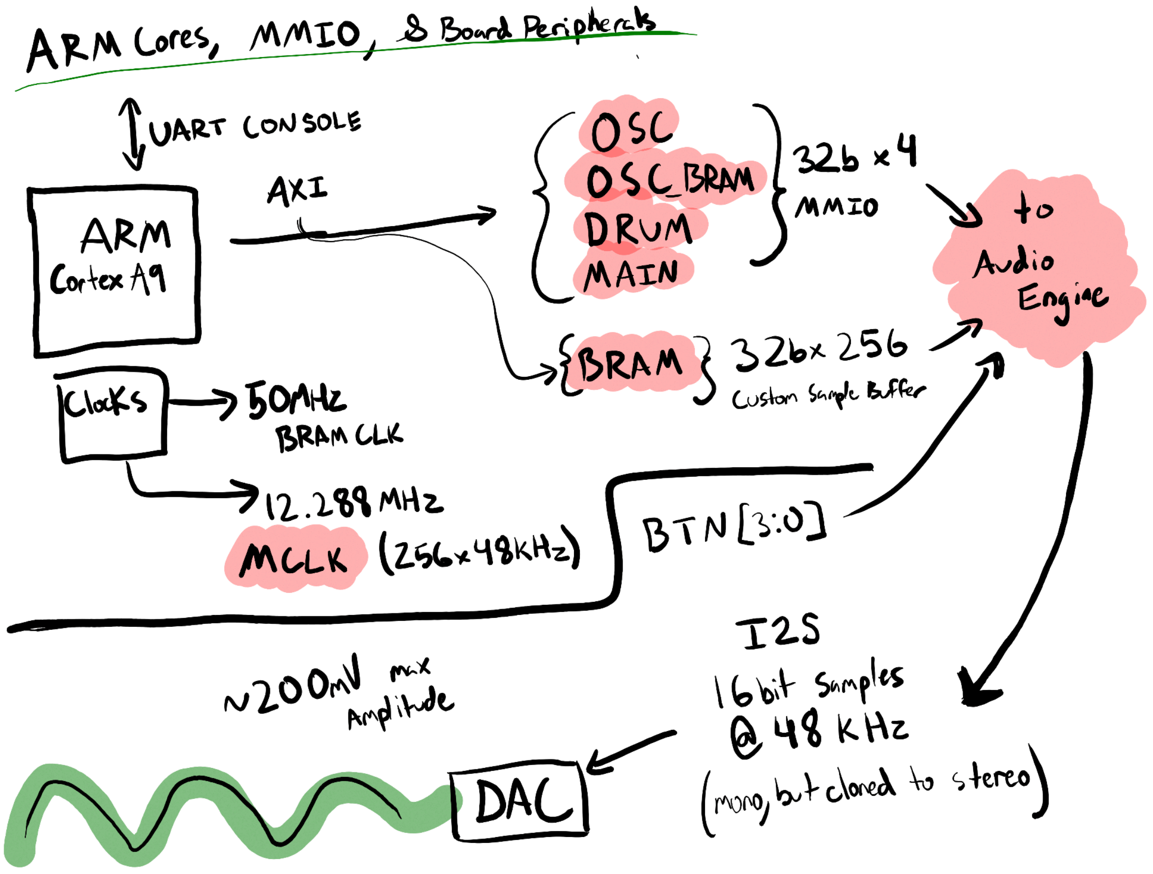
- UART Host PC
- I2S DAC - PCM5102 Breakout Board from Amazon –>

- Zybo Board - Zynq FPGA/SoC
    - PS
        - Arm cores handle UART messages, and control PL using MMIO
        - *Included as: helloworld.c, Vitis Libraries*
    - PL
        - Parameters are received from MMIO, and audio is synthesized with the **Audio Engine,** using digital waveshaping.
            - Audio Combinator and I2S Controller
            - Audio Sources & Oneshots
                - Combinations of the following:
                - debouncers,
                - envelope controllers,
                - oscillators,
                - filters,
                - overdrive
            - *Included as: Vitis SoC Block Diagram, All SystemVerilog and Python Files*

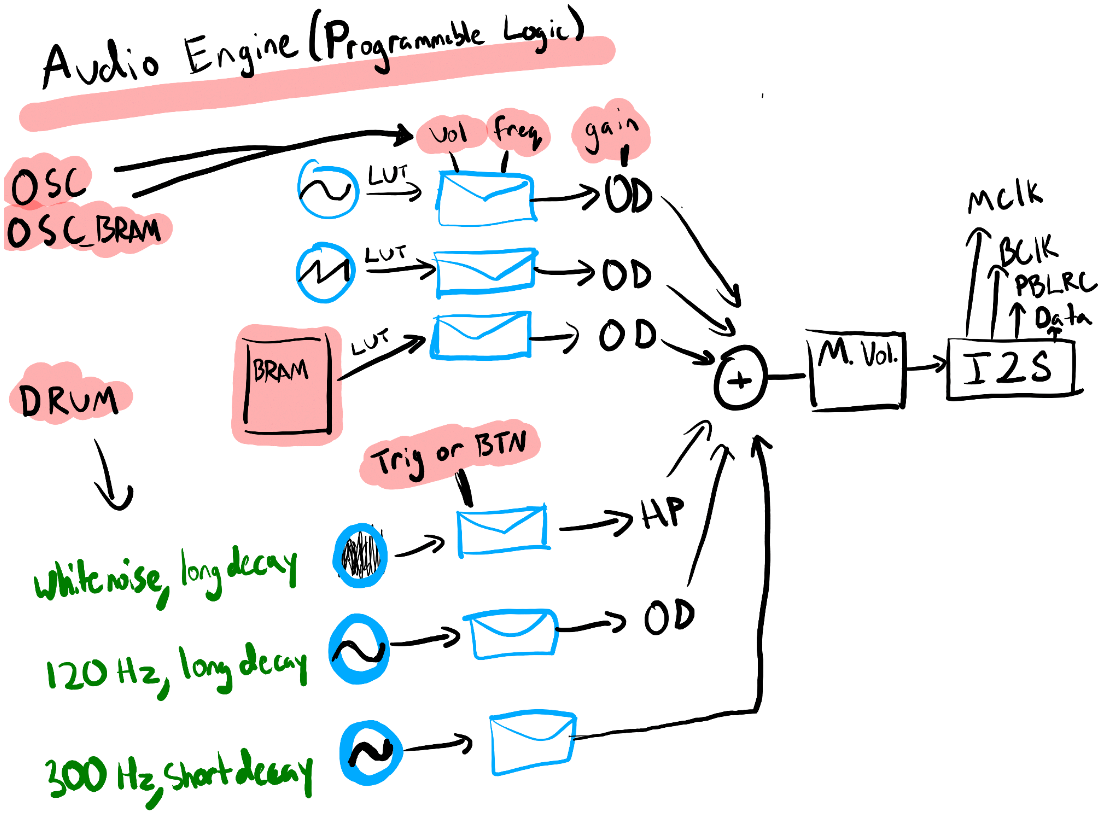
THE RECIPE

Let's start with the physical components. The Zybo Board includes a set of buttons, an I2S DAC, and a Zynq SoC. I elected to use an external DAC on one of the PMOD ports, but by changing the pins, and I2S DAC should work with the 48kHz 16b format. The DAC comes with a headphone jack and amplifier for audio output.

The real special sauce here is the Zynq Soc, consisting of a dual core ARM processing system (PS) and a smattering of flip-flops, LUTs, and DSP blocks connected by a configurable interconnect fabric known as the programming logic (PL). This combination is as good as sweet and salty, as the highly parallel PL can accelerate tasks for the sequential and conditional code executed by the PS. The interface between the PS and PL is done via MMIO registers, supported by the AXI Bus. For our hardware synthesizer, we synthesize the audio in the PL and handle higher level configuration and control in the PS. The PL is configured as an Audio Engine and provides various instruments. The PS plays them according to the user's code, or UART commands. The PL Audio Engine can also be accessed directly, using the buttons on board, and the result is directly output on the I2S DAC, and played through speakers or headphones.



The PL Audio Engine is populated primarily by Envelope Players - these are provided a short sample loop (like sine), appropriate clocking and configuration signals, and in return provide a clean, unaliased, and volume adjusted signal with the waveform frequency of your choice. The envelope players control rise and fall times, as well as whether or not the waveform can be held (oscillator vs one-shot). Envelope players are the only audio sources, and are summed before a final volume adjustment. The final signal is output via I2S, a fairly simple serialization protocol for streams of data. The audio engine internally runs two clocks - one at the I2S sample rate[2], and one that runs 256 times faster.[3] Having a faster clock domain means you can squeeze more calculation iterations into a single sample iteration, and enables sharper filters with many more coefficients. A single hardware multiplication unit can be time-division multiplexed for 256 times as many calculations per sample.

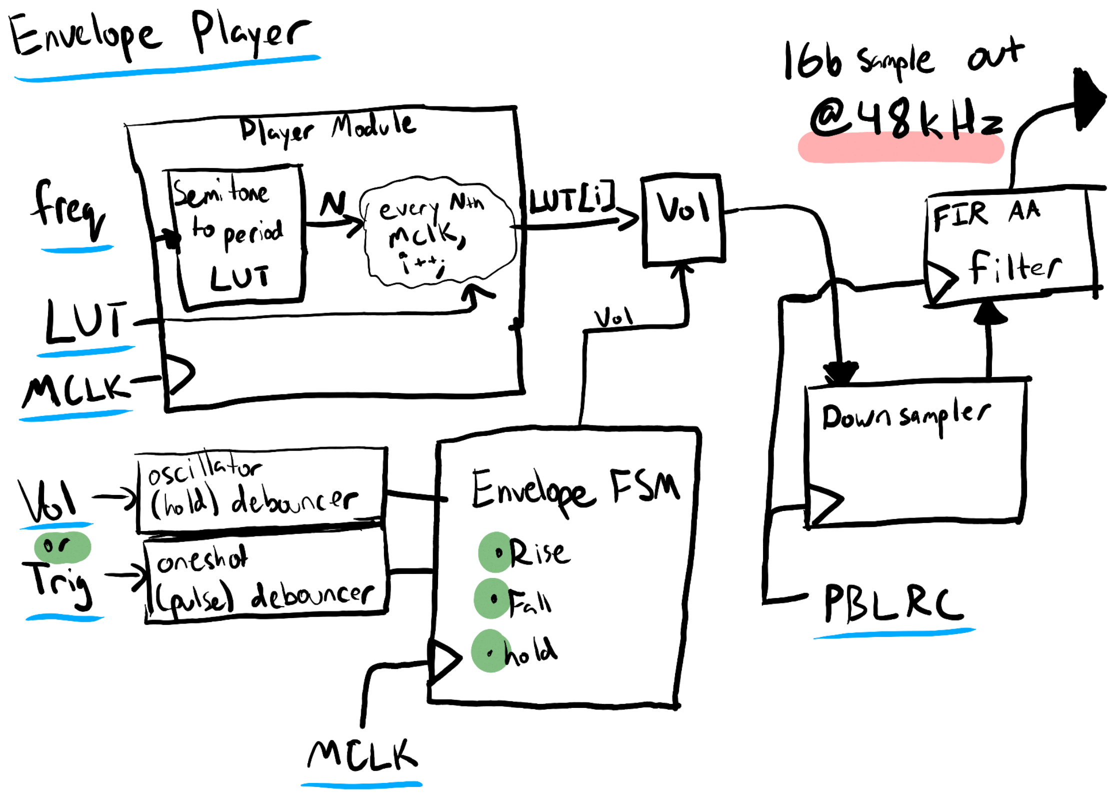


The other major concern of the envelope player is properly resampling the signal, such that a single LUT with a single fast clocking source can be replayed at a wide and precise range of frequencies. There are many solutions and there are tradeoffs to each design choice. The included code uses a small LUT to convert integer semitones to a clock prescaler, allowing for varied frequencies with integer math. This clock divider selects

[2] Playback Left/Right Clock (PBLRC)
[3] Main Clock (MCLK)

the index of a sample LUT, which is scaled according to our rise/fall envelopes. Both of these are updated on the faster clock, and the output is downsampled to the I2S playback frequency and processed with a lowpass filter to suppress aliased noises. Aliasing is caused by high frequency inputs corrupting the the output of a sampling system, and can be avoided by suppressing frequencies above the*Nyquist rate* prior to sampling. The Nyquist rate is half of your sampling rate, in this case 48kHz / 2.

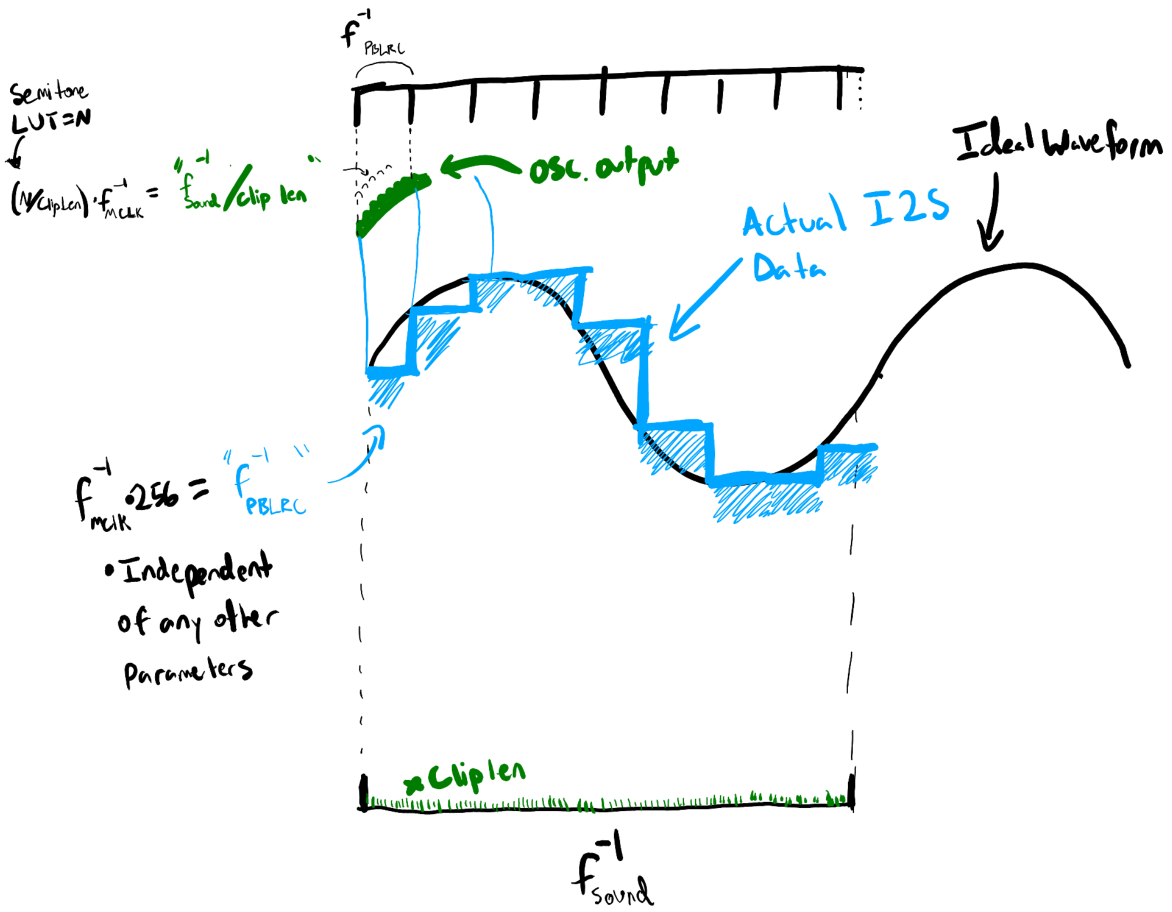
In the frequencies of the audible spectrum, for our chosen system parameters, a single sample value is held constant for multiple iterations of the I2S sample rate[4]. This is a technique known as *zero-order hold* downsampling, and does not actually solve aliasing, but the effects of aliasing are mostly drowned out by the overdrive distortion and filtering. An aggressive low pass filter (with a cutoff frequency of 10kHz) applied after downsampling also suppresses the very highest frequencies that are particularly unbearable.



The challenge here is that attenuating the high frequencies should be done prior to downsampling, as frequency content over the Nyquist rate prior to sampling is the sole cause of aliasing. An actual implementation of the discrete time filter would use a variable sample frequency produced by the semitone LUT. Consistently keeping the filter coefficients aligned to a cutoff frequency of around ~ 24kHz is rather tough. There are

[4] This is because the oscillator's sample update frequency is always much lower than 48kHz. Even with a sample buffer that is 1024 samples per period, you would have to play it at 12kHz before samples are not duplicated in a period of I2S playback.

strategies for this, but in this design, the filtering is just done after downsampling. It does lead to aliasing, but works well enough. The below diagram illustrates the audio signal at a few points in the signal chain, for a playback frequency that is very high (towards the upper end of the audible spectrum). It can be seen that the oscillator (green) is updating much more frequently than the I2S sample rate (blue). For lower frequencies, the oscillator's clock divider causes a single oscillator sample to be duplicated.


f^{-1}_{PBLRC}
Semitone LUT=N
$(N/\text{ClipLen}) \cdot f^{-1}_{MCLK}$ = "f^{-1}_{sound}/clip len"
osc. output
Ideal Waveform
Actual I2S Data
$f^{-1}_{MCLK} \cdot 256$ = "f^{-1}_{PBLRC}"
• Independent of any other Parameters
x Cliplen
f^{-1}_{sound}

An improved design would use *zero-insertion*, an alternative to zero-order hold, and a rational fraction upsampling/downsampling scheme. Zero-insertion does exactly as it sounds, separating the actual samples by loads of 0s to stretch it out. This is in contrast to zero-order hold, in which the sample is duplicated, leading to a staircase effect. Zero-insertion introduces its own distortion, as it duplicates a time-scaled copy of the original signal across the frequency spectrum, and lowers the amplitude of each copy. After the higher copies are filtered out, you can apply a variable gain to the extracted signal, and downsample once again to obtain what should be a much cleaner signal.

As mentioned previously, the key difference here is that the schema used in the final dish (no upsampling, no zero insertion) requires a low pass filter that can adapt to different sample frequencies, but the improved design has a constant sample frequency for the upsampled intermediate product. This means a single set of filter coefficients can be used to prevent aliasing for all playback frequencies, and is much easier to implement in hardware. To further improve the efficiency of the filter, a *polyphase filter* can be implemented, which takes advantage of the fact that many of the samples in the signal are zero, and thus do not need to be explicitly multiplied by a filter coefficient, as the product is known to be zero.

RECIPE CARD - *Steps for viewers at home*

1. Install Vivado and Vitis - I used version 2024.2, but the steps should be similar on newer versions.
2. Start by getting to know the Vivado Block Diagram system - this is the best supported way of configuring your Zynq PS system, and connecting PL elements to the PS. I started with this 3 part series of videos, which take you from a totally blank project to a good starting point. This includes:
    a. Clock and reset configuration
    b. MMIO setup over AXI for PS-PL intercommunication, allowing you to use Vitis C libraries to access registers via memory address.
    c. BRAM instantiation, including access from the PS and PL. In this project, this is used for custom runtime waveshapes for your oscillator units.
    d. And all of the steps that Vivado requires to make it happen.
3. By this point, you should be able to:
    a. Read buttons and switches from the PS and PL
    b. Use a UART terminal to send messages to the PS, and toggle registers in the PL (try an LED).
    c. Read and write to the BRAM from PS or PL.
    d. Add your own IP through the Block Diagram, or pass necessary control signals from the Block Diagram to the BD wrapper, written in SystemVerilog.
4. Implementing an I2S transmitter and audio source in PL. This can be done from top down or bottom up, but I like to prioritize incremental testing, and start from the parts of the design that are more fixed (like I2S or interfacing with the PS). Test each step, both in simulator and in hardware.
    a. Start by implementing the I2S spec as a transmitter - this isn't as bad as it sounds, as stream protocols are often easier than addressed protocols. There are tons of resources on this (including my I2S transmitter). The I2S transmitter actually creates the PBLRC, which later hardware depends on.
    b. Next, try to play a square wave, by playing either high or low with a clock divider. Ensure that you are using 16b signed samples, and that the square

wave updates in time with the same clock used by the transmitter. Sweep the frequencies and listen or check it out on a scope.

    c. Your square wave module should serve as the basis for your envelope player. Flesh it out with volume control, more accurate frequencies[5], and finally implement one of the resampling and anti-aliasing schemes discussed. I recommend you package this all into a single module, parameterizing things like the sample LUT, frequency, and volume - these are functions that nearly every output signal will need.
    d. Finally, implement enveloping. This can be as simple as a state machine that modulates volume over time - try out different parameters, at runtime if your hardware allows. I would also suggest you play with one of the online waveshapers, just to get a feel for how it sounds. You could also just verify everything on a scope, but cooking to taste is much more fun.

5. If you did not do this during the previous stage, now is the time to integrate the PS, as a high level controller for the Audio Engine.
    a. Attach MMIO registers to various parameters and control signals in the PL. I chose to give the PS access to only frequency and volume for the most part, with everything else being static, but the world is your oyster.
    b. Handle UART input with C code, and tweak the MMIO registers accordingly. You can even program short songs in C, as the timing controls of the PS are much less precise than PL, but still sufficient for orchestration.
6. Add more sources, sum them, and further parameterize. I chose to add 3 synth leads, and 3 unique drum sounds, but get crazy with it. Here are some things you might consider:
    a. Chording can be accomplished by playing other frequencies with the same waveshape - for my implementation, this is as simple as using a *generate* statement to duplicate an audio source, then adding a constant (in integer semitones) to the frequency.
    b. Filtering your output can do more than prevent aliasing - I used a high pass filter to make white noise sound more like a hi hat hit, for example. There are many sources for waveshaping techniques, try googling “how to make X in Serum” or just “waveshaping” to get more inspiration.

[5] For the 12-tone western music scale, every 12 semitones corresponds to an octave, which is a doubling of the frequency. This means that you only need an LUT of size 12 - 1, as shifting easily makes your clock divider change octaves.