# 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.

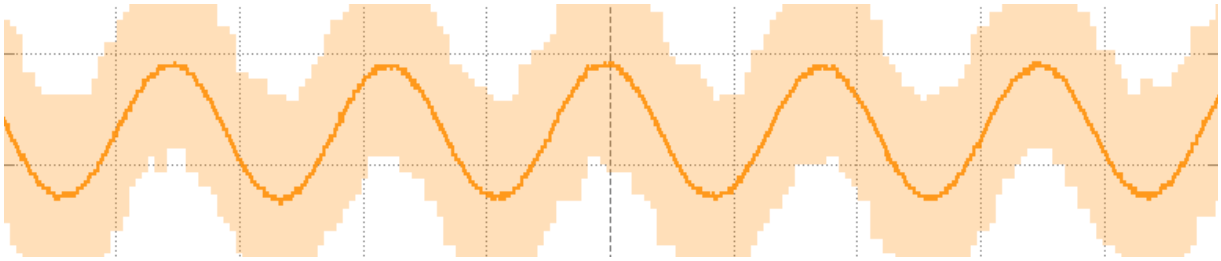
In this cookbook, I'll discuss the high level ideas behind audio synthesis, as well as how they can be implemented with FPGA and SoC devices in mind. By the end, you should be able to generate your own waveforms, free of aliasing, clipping, and with runtime-variable playback frequencies.

# ANALOG SYNTHESIS *- Grandma's Recipe*

*>> What is audio anyway?*

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.

*>> Our first Ingredient - Sine Wave*



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:

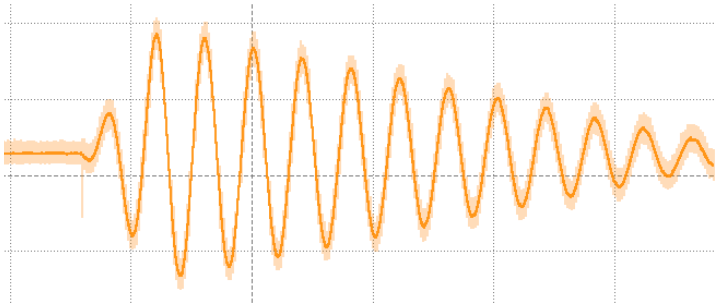
1. any complex waveform can be modelled by a collection of sine waves with 2 parameters, magnitude and frequency[1]; ie **a frequency distribution.**
2. Any complex waveform can be modelled by a **series of discrete samples** with 2 parameters, magnitude and time.
3. All sound can be thought of as a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

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

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.*

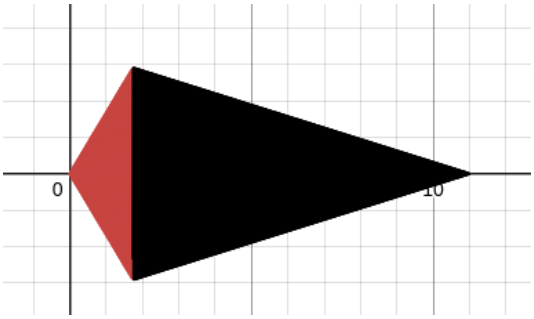


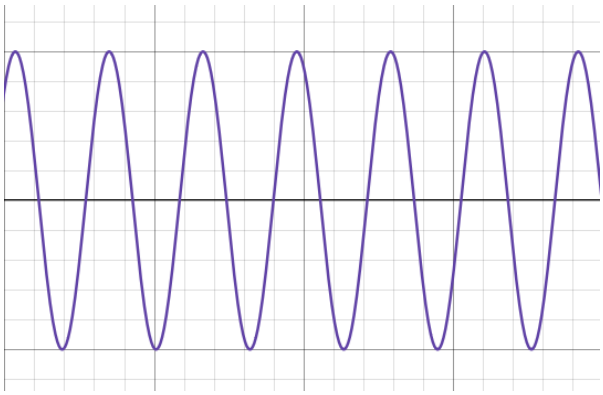
*>> Our first dish: The 808 Kick Drum*

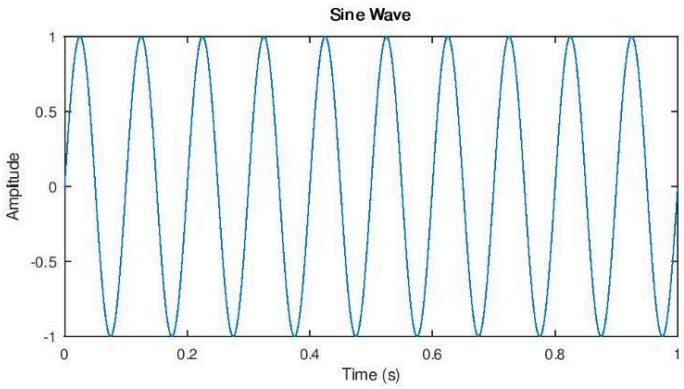
Do you recognize that sound? Of course! It's an 808 kick drum! Popularized by the Roland TR-808, this is a common kick drum for electronic and hip-hop music, and has thousands of variations. Let me talk a little about how we got this starting from a sine wave.

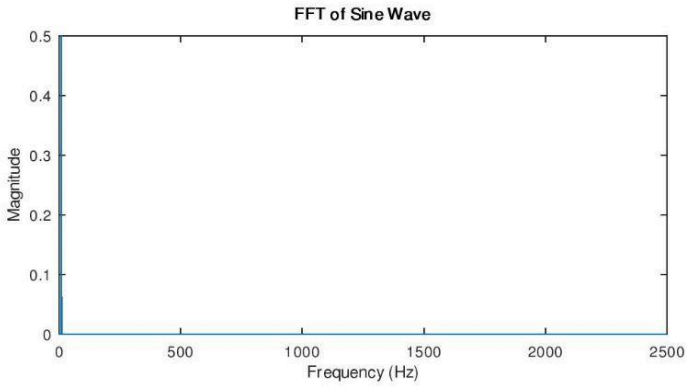
We start with an ***oscillator***, which produces a humble sine wave at a low frequency, somewhere in the range of 40-120 Hz. This gives us a nice and hearty flavor, as this is near the lowest tones we humans can hear. But the percussive nature of a drum means that the sound takes a moment to build and 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).

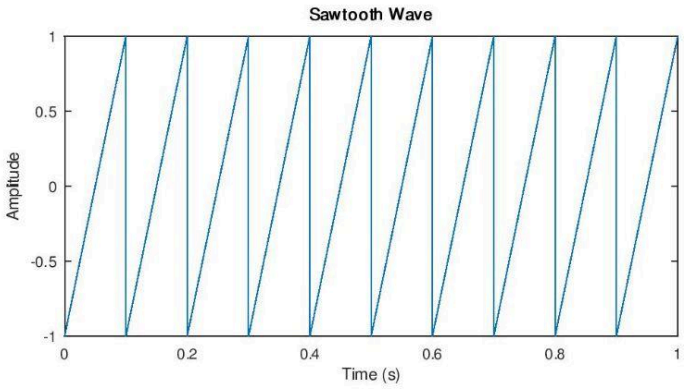
Envelopes are incredibly versatile, and can be used to control frequency, volume, distortion, filtering, or any other parameter. You could even use a sine wave as an envelope, if you wanted.

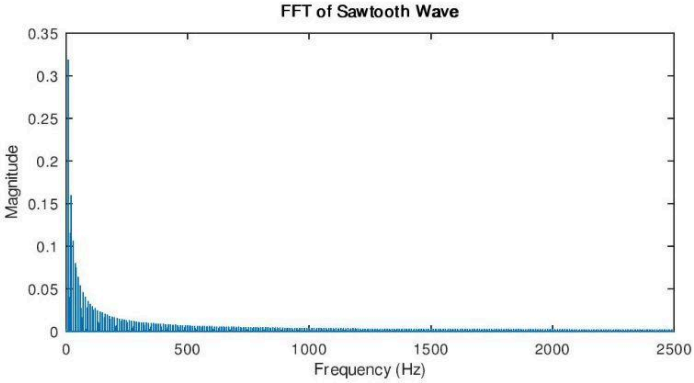


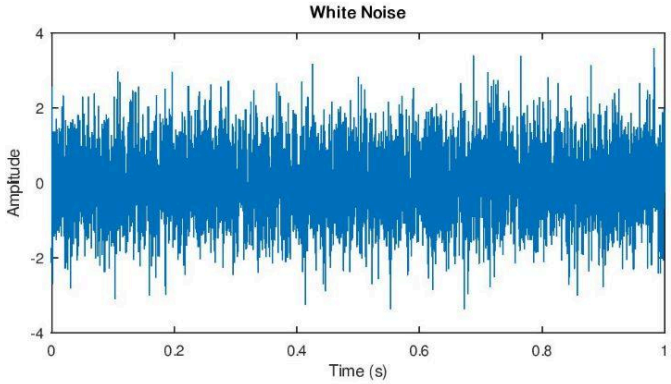



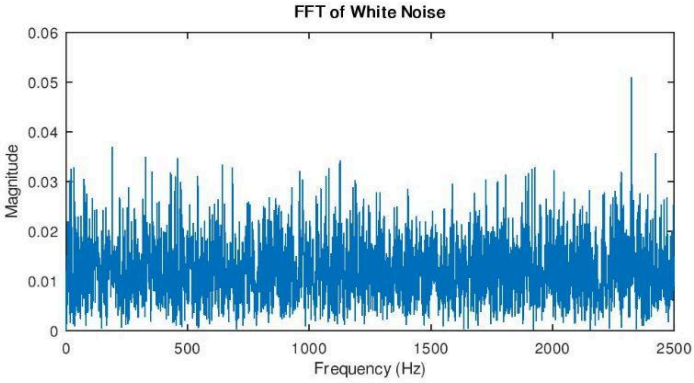



*>> A Little Spice - Harmonics and Alternative Waveforms*

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 very 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
- 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**
            - Audio Combinator, Master Buffer and I2S Controller
            - Audio Sources & Oneshots
                - Combinations of the following:
                - debouncers,
                - envelope controllers,
                - oscillators,
                - filters,
                - and more
            - *Included as: Vitis SoC Block Diagram, All SystemVerilog and Python Files*

# “Global” memory and Caches

*> Why Caches?*

As any digital designer worth their salt knows, memory is slow, and for this reason caches are a crucial optimization point for processor design. Layering caches increases the area usage of the design, but can allow for much faster data accesses when nearby, a principle known as *spatial locality.* Because the memory accesses are decoupled from the cache accesses, there are a number of cache control strategies which attempt to increase the chance that the requested data is already present in the most local cache, including direct and set associative.

*> Why are Caches problematic for GPUs? (and multicore CPUs)*

For a single core, it is trivial to manage data synchronization, as you only have a single writer/reader of the data[2]. Parallelization through increasing the core count of a processor adds increased complexity to the cache, as now you must handle transactions with some number of consumers, and some number of data providers. For a 4 core processor, with single channel memory, you have 4 potential sources of data, and no guarantee that they will agree. This is the problem known as *cache coherency*, and is a huge problem for modern digital design. Modern high performance processors typically include 3 layers of cache, and rarely 4 or more, due to the additional area, power, and

To further complicate the issue, and illustrate how there is no clear solution to the problem, there are multiple hardware configurations which can be used. Transferring data from N number of memory channels to M lower level caches can be done using a simple shared bus, crossbar busses, or a dedicated memory transaction arbiter (Yousif et al.).

*> Great minds think in parallel*

This problem of shared memory access is fundamental, and there are numerous ASICs, algorithms, and design patterns which exist to exploit parallelism and avoid the need for shared memory in the first place.

[2] As long as the memory itself is not shared with other hardware units, like interrupts or DMA channels.

# “It’s probably fine… just ship it”

*> What is most important: Power Consumption, Clock Speed, or Man Hours?*

As one might expect, designing a processor is a difficult task. If you disregard the actual design, fabrication, and funding, you still have to go through the process of testing it out before selling to a customer. Validation, which accounts for a huge amount of the effort put into modern chips, is a problem that has only grown in scope as our digital design skills have increased. As chips scale down, core counts and cache layers rise, the need for an efficient software stack and comprehensive testing environment becomes evident.

## HIGH LEVEL DESIGN

*> An introduction to tiny-gpu*

> *“A minimal GPU implementation in Verilog optimized for learning about how GPUs work from the ground up. Built with <15 files of fully documented Verilog, complete documentation on architecture & ISA, working matrix addition/multiplication kernels, and full support for kernel simulation & execution traces”* [https://github.com/adam-maj/tiny-gpu](https://github.com/adam-maj/tiny-gpu)

The tiny-gpu, as provided on github, has 2 compute cores,[3] with 4 threads each. It makes use of the Harvard Architecture for memory, and uses memory controllers to interface the fetchers and load store units with the program and data memory. The data memory has 4 8-bit channels, and the program memory has 1 16-bit channel. There is 1 fetcher and 4 load/store units per core.

Despite the 16 bit instructions, the tiny-gpu operates in an 8-bit address space, and it’s operations are the bare minimum. Despite the limitations, it is highly parameterized, and could be adapted to be more suited for testing highly parallel cache designs.

[3] Although the diagram on the readme illustrates 4 cores, which is not the default configuration

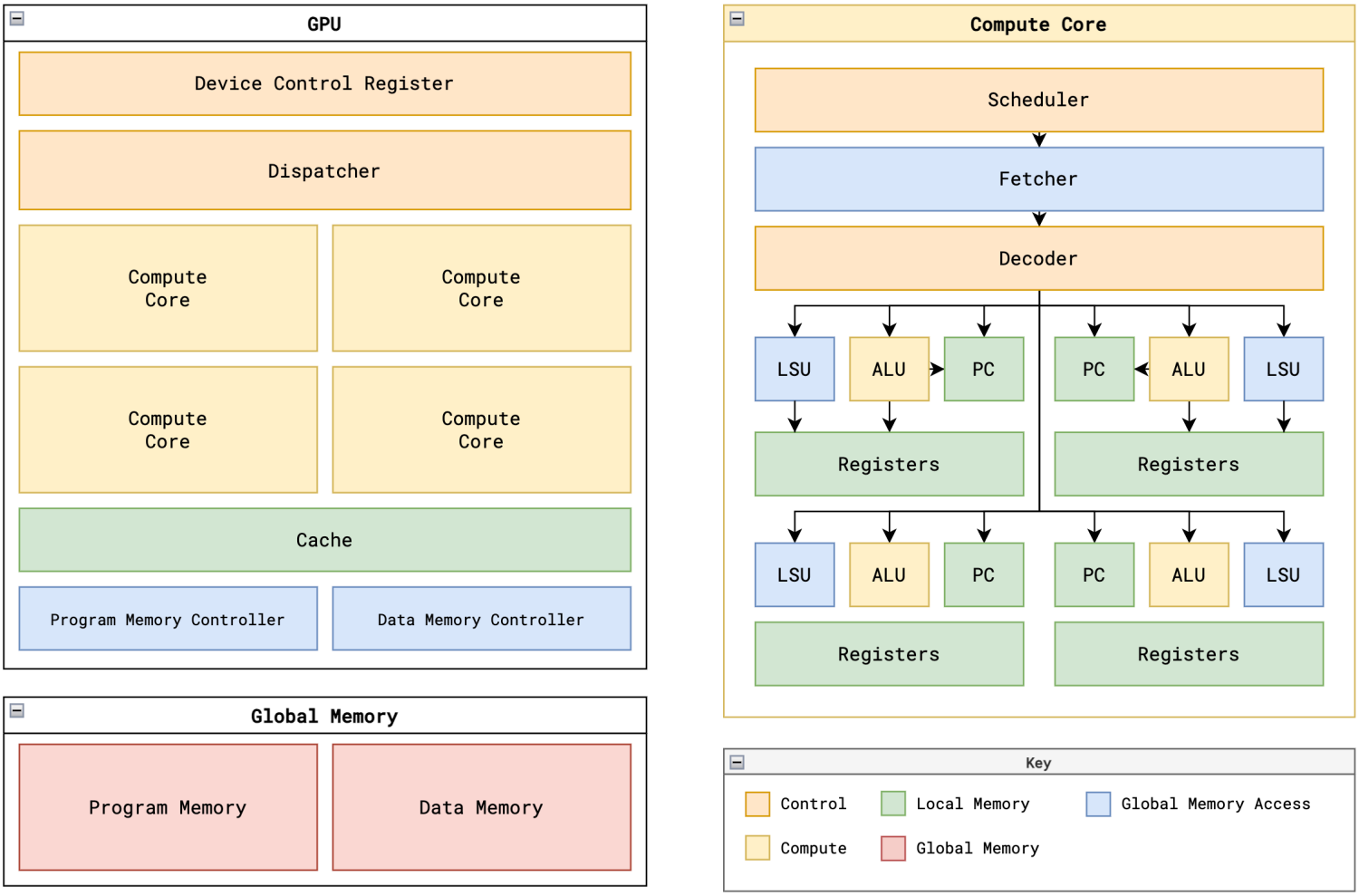



*Figure 1: tiny-gpu reference design, configured with 4 cores and 4 threads per core*

> *Modifications Introduced*

- Program Cache
  - 16 x 16 bit cache lines (yes, a whopping 256 bits of cache!)
  - 4 bit tag/index, no offset
  - Direct Mapped
- Data Cache / Controller
  - 16 x 16 bit cache lines
  - 3 bit tag, 4 bit index, 1 bit offset
  - Direct Mapped

*> Design Process*

Implementing a cache requires two related but separate tasks, storing the data and managing its correctness, and communicating with the source and consumers. The memory controller included with tiny-gpu handles the second task nicely, and naturally integrates with the cache design. The program memory cache as included with this document is separate from the program memory controller, whereas the data cache is integrated with the controller for better performance.

By first implementing a single input, single output cache for the program memory, I was able to get a basic cache design working. The program memory cache used in this design improves performance only when branches and loops are used, as no prefetching occurs.

Following that, a direct mapped cache was designed based on the program cache, including modifications to allow writes and integrating the memory controller with the cache storage.

*> Validation*

Beyond the intricacies of the cache design, a major hurdle was uncovered - validating the design. Aside from the provided test_matadd and test_matmul programs, there was no way to test code on the tiny-gpu, cache or no cache. This lead me to building a “quick and dirty” assembler for the tiny-gpu ISA. It took me way longer than expected, and was only the first part of validation.

Another issue arose from the memory simulation - The existing tiny-gpu project makes use of a CocoTB module for memory simulation rather than SystemVerilog, so it took a bit of work to make it behave as desired. The benefits of writing testbenches in Python is very attractive, but CocoTB’s simulation is very slow, and most of what it enables could be done with scripting. Overall, I would not recommend CocoTB for any serious project, aside from the rapid prototyping stage.

## VERIFICATION/THOUGHTS ON METRICS

Testing was done using CocoTB, icarus-verilog, sv2v. A variety of assembly programs were tested and verified that their resulting data memory was equal. Various memory read/write delay parameters were used to evaluate the performance in different memory configurations.

## RESULTS - Figure 1

| Test Duration (ns) | tiny-gpu (reference) | tiny-gpu + tiny-cache | speed (%) |
|---|---|---|---|
| Test_matadd<br>5 cycle memory delay | 7075 | 6775 | 96 % |
| Test_matmul<br>5 cycle memory delay | 17050 | 17675 | 104 % |
| Test_load<br>0 cycle memory delay | 15025 | 17025 | 113 % |
| Test_load<br>20 cycle memory delay | 30150 | 21825 | 138 % |

## ANALYSIS

There is considerable room for improvement, but the results are generally positive. An improvement on the design would have cache controller threads lock onto cache lines, not lock onto requesters. Future improvements could include set associativity, or splitting the cache levels even further. Some sort of synchronization primitive could be useful as well, but the most pressing improvement would be in the test and verification environment. While much has been done to set up a test environment, it is still lacking, and severely limits the testing for highly parallel systems, where layered caches become more important.

## RESOURCES

**Resource #1 - tiny-gpu files**

- Majorly changed files include:
  - tiny-cache/test/
  - tiny-cache/src/dmem_cache.sv
  - tiny-cache/src/pmem_cache.sv
  - tiny-cache/src/gpu.sv

**Resource #2 - tiny-assembler files**

- This was entirely from scratch, referencing  the tiny-gpu ISA and test programs.

## RELEVANT PAPERS

Indra Vir Singh, et al. “Cache Coherence for GPU Architectures.” *High-Performance Computer Architecture*, 23 Feb. 2013, https://doi.org/10.1109/hpca.2013.6522351. Accessed 1 May 2023.

Loh, Gabriel H. *Extending the Effectiveness of 3D-Stacked DRAM Caches with an Adaptive Multi-Queue Policy*. 12 Dec. 2009, https://doi.org/10.1145/1669112.1669139. Accessed 29 Aug. 2023.

Yousif, Mazin S, et al. “Cache Coherence in Multiprocessors: A Survey.” *Advances in Computers*, 1 Jan. 1995, pp. 127–179, https://doi.org/10.1016/s0065-2458(08)60546-x. Accessed 8 Dec. 2024.