Lucas Elroy

6/2/2023

Audio Effects and Visualization

Design Procedure:

The idea of this project was to create some distortion effects on input audio then have some waveform of this sound show on the video display. This simple idea became complex very fast. Originally the effects were thought to be the challenging portion of the project, but it was quickly discovered that the video interfacing would be the most challenging.

Overall System:

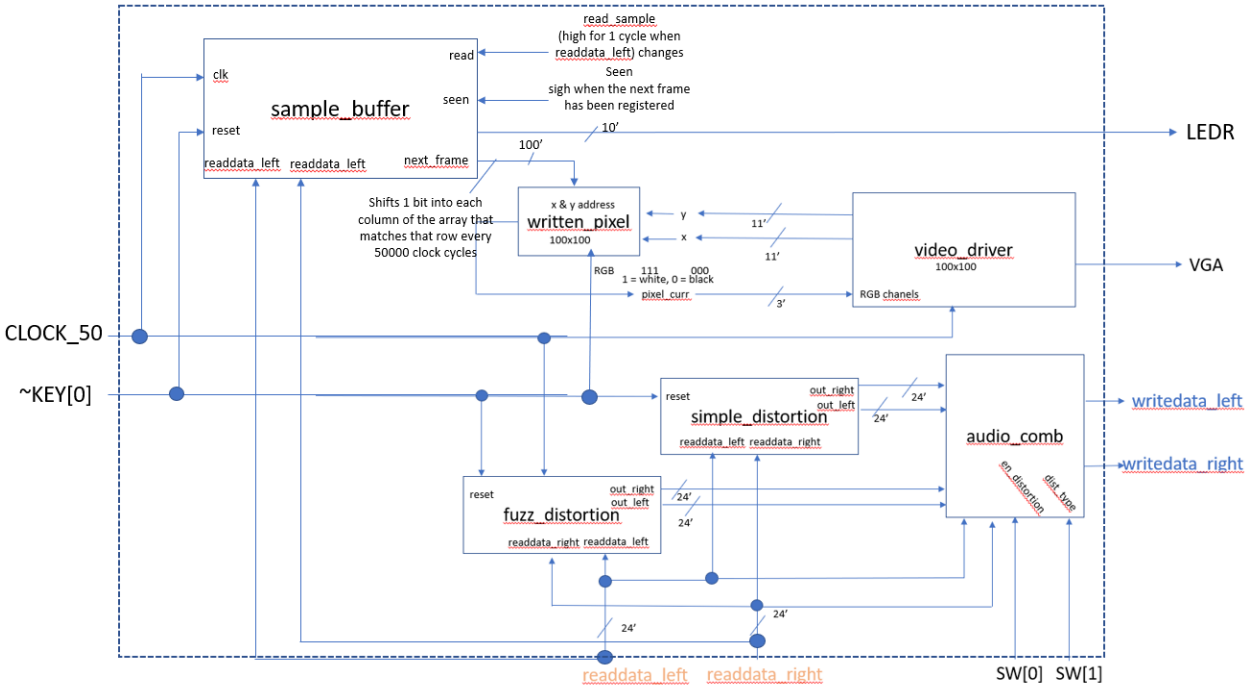

Figure 1: block diagram

It is important to first mention that this diagram does not cover most of the intricacy that is seen in the System Verilog code. The main thing that this diagram does not do a good job of covering is the system's reliance on clock counting. Basically, modules use the idea of counting a certain number of clock cycles before something happens. The first example of this is how the screen is shifted over every 500000 clock cycles. This number essentially equates to this action happening 10 times a second. These seemingly arbitrary numbers are seen in many places but basically all have the same idea. For more information about this check the comments in the Verilog code.

There is an auxiliary asynchronous RAM structure in place to keep track of the current state of the VGA screen. Every pixel is stored in a 100x100 array RAM. This was done to keep

the amount of bit storage as low as possible. With the standard SystemVerilog RAM there would be an excess of bits, so to reduce memory the 2D array structure was used. More about this in the sample buffer module section.

VGA:

The first part of this lab was designing the interaction between the audio and the video. The idea was to create a waveform which sampled once every 10^{th} of a second and averaged all the audio samples heard in that time. To do that a module was created called sample_buffer. This would essentially take in audio samples and average them, then calculate how large the wave line should be then sends it to the VGA. To begin to solve the rest of the interactions 100x100 array was introduced to store the color of every pixel. The idea with this was to shift in a 100-bit signal from left to right that would represent this waveform sample. This 100-bit signal could be scaled to have a higher number of 1s if the audio sample is loud and a lower number of 1s if quiet. There was quite a bit of auxiliary problems that came with implementing this.

The next step was to find out what values to use for scaling. This process probably took longer than any other step. To start this process the white noise level in the room of the DE1_SoC had to be measured to understand what level of sound should be registered. Then the level of the input audio had to be scaled so that all the different ranges of sounds are seen at different volumes. This alone took close to 100 iterations of values and systems.

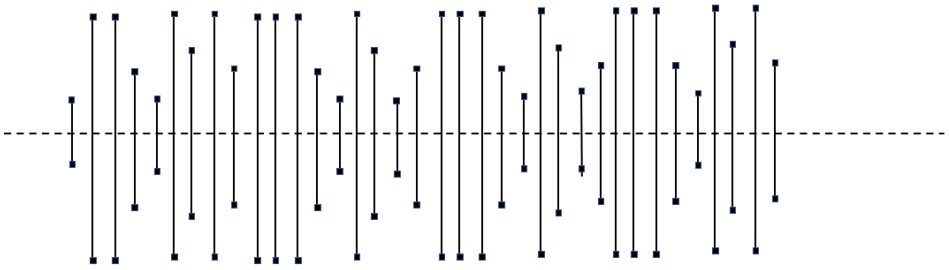

Figure 2: Waveform Draft

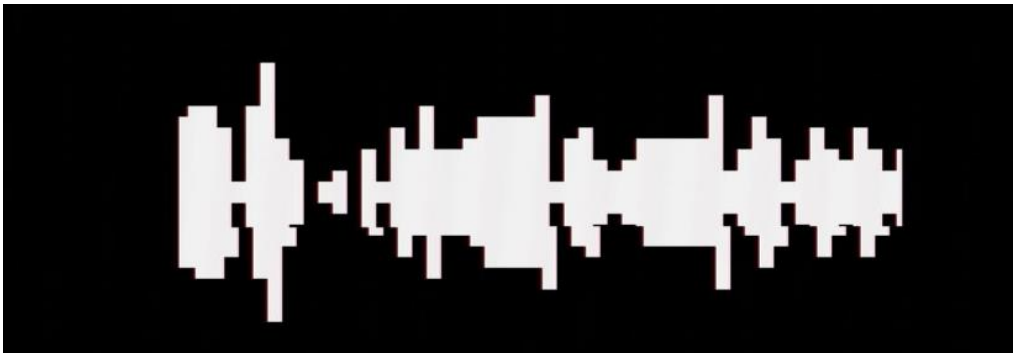

Figure 3: Final DE1_SoC Waveform

Simple Distortion:

Implementing this was quite simple. This distortion is defined by an idea of flattening the upper and lower portions of the audio. This is done by setting everything above a threshold to that threshold. The number that was chosen for this was 600000. Once again, arriving at this number took a significant amount of tinkering. This type of distortion is intentionally overwhelming and loud.

Fuzz Distortion:

This type of distortion uses the previously defined type of distortion with an extra level of complexity. Instead of just cutting anything above a number, it will now scale depending on some factors. The factors are based on how many samples in a row are above the previous sample. So, if we have a sample sequence like 5,7,10,14, and 17. There are 5 consecutive samples of increasing max. If all of these are above our threshold, then 5 would be scaled down only by 1 (first above the threshold). Then 7 would be scaled down 2, 10 would be scaled down 3 and so on. The ending result of doing this is shown in the image below.

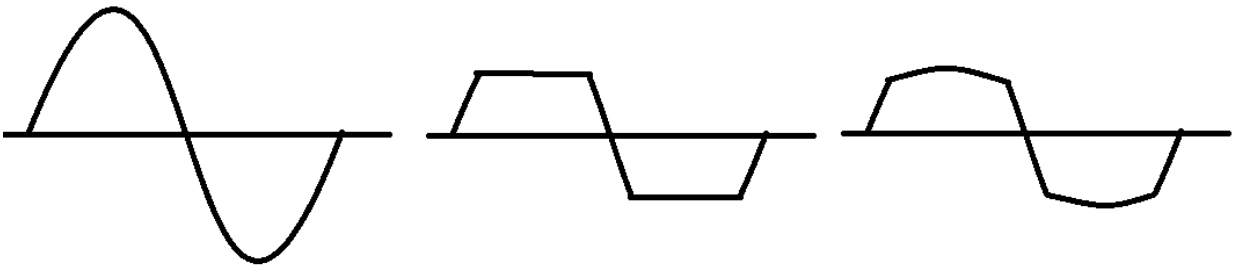

Figure 4: Left shows no cut, middle shows hard cut, then fuzz cut.

Basically, this type of distortion does not kill the harmonic nature of the higher regions of the signal. So, it ends up sounding like a much more musical version of the original distortion. The next step in creating an even better type of distortion would be introducing noise, but that would've been too much for this project.

Results:

To show the proper functionality of the system, it was important to test each component. Testbenches and simulations were created for both distortion effects and the sample buffer.

sample_buffer:

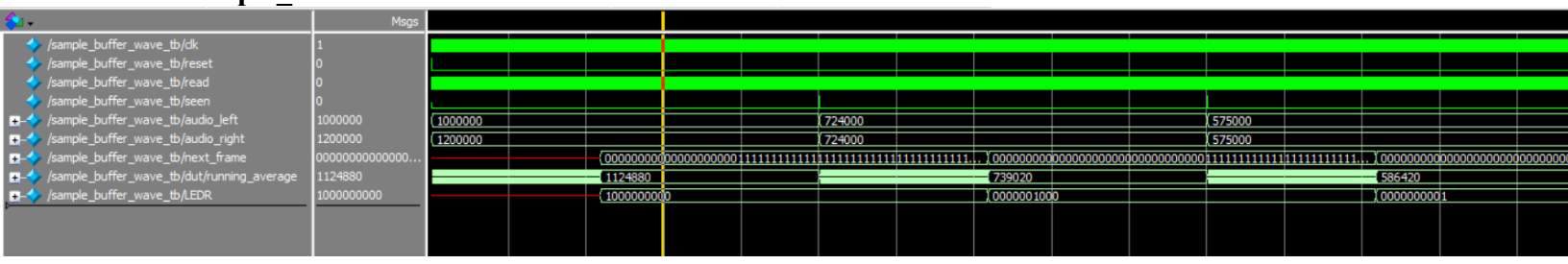

Figure 5: Sample Buffer Testbench waveform

Only 3 tests were written for this module, but each of them contains proof that all the other parts of this module are functional. The first test is the overload test. So, it tests how the system will react when it receives all extremely large audio samples. As this shows LEDR9 lights up after the running average is calculated. LEDR9 represents the largest possible wave. Then the waveform shows the running average held until the seen signal is high then it begins to calculate the next running average. The next running average is the middle case. It is a middle range of sound for the DE1_SoC and as is seen in the waveform. After a certain number of clock cycles it once again calculates the running average and outputs LEDR3. LEDR3 signals it is a middle range audio sample. Afterwards it waits for the seen signal and begins calculating the next average. This one has lower audio samples. Just like before the average is calculated and LEDR0 is lit showing it is a low volume.

simple_distortion:

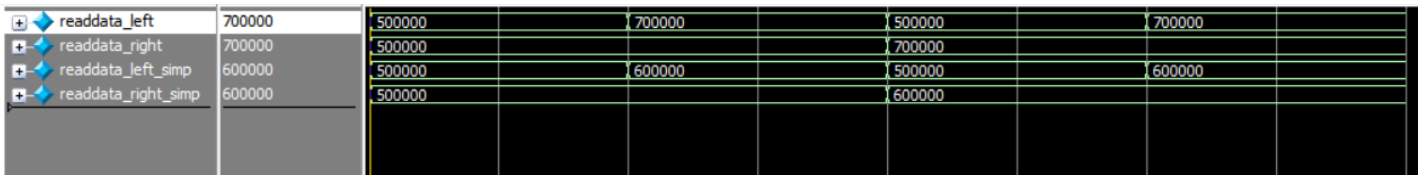

Figure 6: simple_distortion waveform

This wave shows functionality for both left and right channels. The first test shows that data below the 600000 cutoff will match for both input and output channels. The next test shows what happens when left audio data is above the cutoff and right data is not. As is seen in the waveform, 700000 is cutoff to 600000 and the right data 500000 is not changed. The next test shows what happens when right audio data is above the cutoff and left data is not. As is seen in the waveform, 700000 is cutoff to 600000 and the left data 500000 is not changed. The last test is when both signals are above the cutoff. As is seen both channels are originally 700000 and are reduced to 600000.

fuzz_distortion:

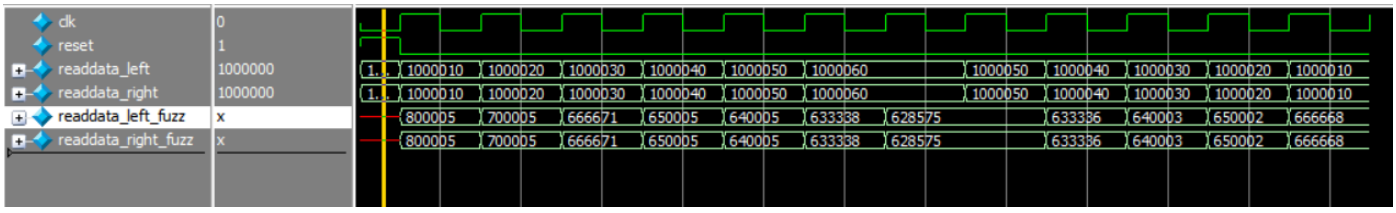

Figure 7: fuzz_distortion testbench waveform

The first half of this test bench shows proper functionality of increasing cutoff strength. If the maximum audio signal keeps increasing on subsequent reads, the cutoff strength increases. As seen in the waveform it starts off by only cutting 200000 then goes to 3000000. It basically makes the output closer to the cutoff but does it gradually. Then after this it shows that when there are two subsequent identical audio values it will not change the cutoff strength. Then it shows that if the audio value decreases it will also decrease the cutoff strength. This behavior is coupled with the ability to recognize when to read the next value based on when the data is changing. This is so the calculations are only made when the audio channel moves to its next output.

DE1_SoC:

It is worth noting that the top-level module is close to impossible to simulate with all the different audio channels and VGA interactions. The previous simulations of auxiliary modules should be enough to show that this system works as intended. For more proof of this, visit the provided video.

Flow Summary:

Flow Summary

<<Filter>>

Flow Status	Successful - Fri Jun 02 10:44:45 2023
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	10653
Total pins	105
Total virtual pins	0
Total block memory bits	12,288
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	2
Total DLLs	0

Figure 8: Flow Summary