Lab 5: Signal Synthesizer
A.k.a. How to talk to codec's 101

Adam Shanta
Nemo Thompson
Ocean Hurd
CMPE 125/L

06/07/19

1 Project

The primary goal of the design is a multidisciplinary pursuit in three fields: signal processing/analysis, time division multiplexing (TDM), and technical document specifications. On a high level discussion, this project has us communicate with the Audio Code 97 Controller, where we send signals–sine, square, sawtooth, and other sound waves–to communicate to an auxiliary headphone jack to produce sounds. In a more fundamental method, this project was an iterative networking challenge of re-/configuring a codec based on user input, and tested in simulation as well as in hardware by each component and unto the larger design as a whole.

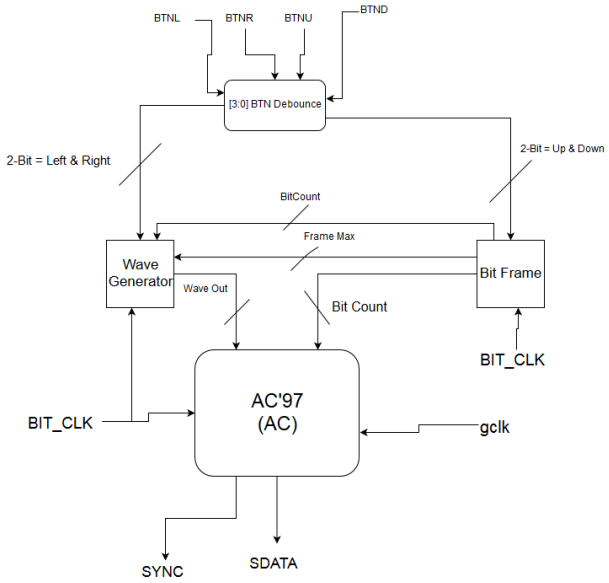

Figure 1: Top Level Diagram

2 Methods

The project was built iteratively and had gone through multiple editions until it became these three major components that contribute to the communication and development of the sound design.

2.1 Bit Frame Count

The bit frame count has two functions that deal with the TDM and frequency modulation for the design. We use a total of four counters synchronization of the program, and an extra two for the extra credit for the song. The bitCount handles synchronization and SYNC timing to communicate and validate the frame. Frame count and freqCount held a reciprocal relationship that dictated the frequency of the program based on button left and right presses. FreqCount would set the constant for a LUT holding frameMax constants that define the frequency of the program, and would change the amount of frames transpired–with the bitCount flag–to adjust frequency. Lastly, for the extra credit, we designed a state that would simply assign a specific frequency–aka a note–in it's own LUT and would play that melody constantly.

2.2 Wave Data

Wave data is a wave generator that is trivial in its conception given one particular task of the generation of the trig function sine wave. Effectively, the waves are generated every time a frame has transpired to the codec, it would propagate a wave by one unit of change each trigger. All frames are propagated at the same time and there's a case switch statement that chooses a particular wave based on the switch combination. The other stipulation of the design was effectively developing a method of convolution to the waveforms for any two waves being active at the same time. For us to perform a true convolution of two frequencies, we would have needed to multiply the signals requiring a 36-bit bus that would've consumed more memory. Rather instead, we simply took the average of two to three waves together and used that as an output.

- *Sawtooth* - This counts up one fraction of 18'h3FFFF over the frameMax depending on what frequency we want. If we want a higher frequency we incremented over a larger step size.
- *Sine* - Stored in memory an amount of discretized values that are indexed in the look up table that complete a sine iteration per wave. This is simply indexed and stored into a new wave.
- *Square* - Effectively a whole bus of the clock that changes with it.
- *Triangle* - Similar to saw, but reaching the peak, begins descending by fractions of the frame max.

2.3 Audio Controller (AC)

Our AC controller wrote information to the Codec using through SDATA OUT and SYNC. This SDATA OUT was made up of a constant stream of frames. Each frame is made up of 19 slots. Each slot is 20 bits (with one exception). Each of these slot's bits represents certain instructions or data necessary to the Codec. These slot's bits were hardcoded in our ac controller and set in if else statements in an always block to alternate between two types of frames appropriately. The first slot sets the valid frame, which Codec to write to (primary or secondary) and the tag bit for slot 1 and 2. We set slot 1 to a hexadecimal address to specify that we are either writing to the Aux or PCM volume. Here the Aux is the general selection to write to the headphones, and PCM volume controls changing the volume. Slot 2 set the data to be read, for example, all zeros in slot 2 with a slot one that specified the PCM volume meant that the volume would be on the highest possible level since it is active low. Slot 2's contents do not matter if the frame is being read instead of writing. Slot 3 and 4's contents both contained the information for the waveforms. Slot 3 just sends this waveform to the left headphone, whereas slot 4 sends it to the right headphone. Our slot

bit values were handled in if/else statements because we needed to first wait for a delay before we start the frame flow so the Codec could warm up, then set the first frame to write for aux, then the next frame to write for PCM volume. Finally, we had an if statement that then set a frame to write after reading for many frames. Doing this made it possible to set the volume, since volume can be set at any point, it needs to alternative between reading and writing to give the user a chance to write new volume data in the first place. In this module we also set our SYNC to go high at our bit count of 255 instead of 256 because we needed it to go high a bit early so the slot 0 information could all be read. Our AC controller ran off of the BITCLK.

3 Testing

We initially started testing using the simulation from the ISE. This was effective in seeing how our signals were working in comparison to our other signals, but did not show the post synthesis BITCLK which we manually input into our simulation. This would work great in theory, but we had no idea what was actually happening on the physical hardware because of this.

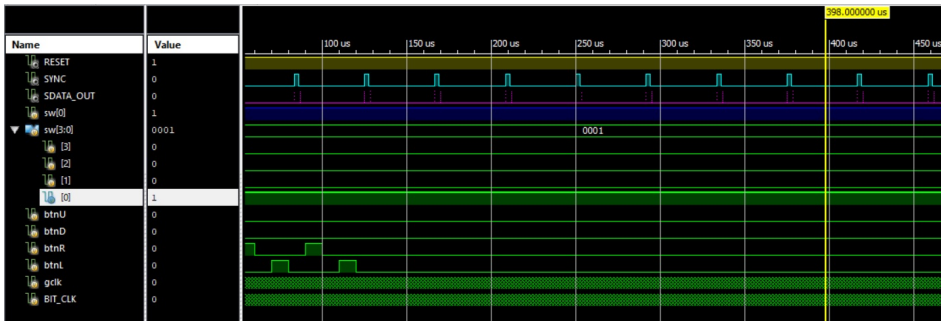

Figure 2: Top Level simulation of the design

A solution to this was to assign the JA ports to BITCLK, SYNC, and SDATAOUT so we could see the signals on the oscilloscope. We found out that BITCLK wasn't even running in the first place. We fixed this by setting the codec RESET output manually at the beginning of our program to run active low for an allotted period of time before setting it to high the rest of the time. Since we got sound to work relatively late in this lab process, we were very careful when testing to maximize our time efficiency so we could turn it in on time. Therefore every new abstraction and addition to our code was checked to see if it still produced sound on the board once we got the sound to work. Working incrementally we got far in our design process with the time we had, and although there could be further optimizations in the way we coded things, we were happy to see it work in the first place.

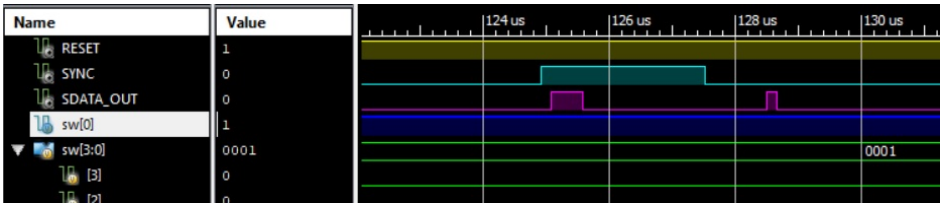

Figure 3: Here's a simulation snippet showing a frame of our data flow. You can see our SYNC goes high at 255, before slot 0 of SDATAOUT's configuration goes high. This snippet excludes slot 3 and 4 as they are low and just out of frame to the right. This image just demonstrates what SDATAOUT and SYNC look like up close, since they are so small and hard to read in the simulation following this one.

4 Challenges

Though we started this lab relatively early, we had most of our difficulty getting any sound in the first place. Many groups who started their lab after us got sound before us. We finally were able to get sound, and found out our struggle was due to not correctly managing when to read and write to the Codec from the AC97 Controller. This lab left an impression on us about how important it is to thoroughly read datasheets and manuals. Because even the misalignment of 1-bit could accidentally mute the whole system.

More importantly though is that to really diagnose an issue it is all about repeatedly testing over and over until it is more clear what is going on internally. We fixed our main problem with sound through testing, and were never actually able to find the corresponding information in the datasheet. Another conceptual hurdle we had to overcome was understanding that the write frame only needed to be configured once and could be read to the rest of the time in slot 1, yet in slot 0 certain valid bits always needed to be configured in each frame. The reason for this confusion was misunderstanding "you need to configure each time" to mean that you need to configure both slot 0 and slot 1 each time, but in reality this only referred to certain bits in slot 0, and slot 1 determined reading or writing. Another relevant challenge was hearing static and getting the sound information from slots 3 and 4 to work properly. The fix to this was setting the two most significant bits in both slots to be zero. Overall, this lab made us more proficient in the valuable and painstaking skill of parsing through technical documents to find the bits of information relevant to the work we are doing.

5 Final Design & Contributions

We were all working in parallel to take different approaches to implement the design for what was most effective and robust to build upon. We all spent roughly the same amount of time in lab for the two weeks we worked on it. We ultimately ended up using Adam's hard coded test case because it was the first to produce sound and worked from there. Nemo helped come up with different approaches to the waveform generation while Ocean worked on volume manipulation and the volume extra credit portion of the lab. Adam scaled the frequencies logarithmic ally for frequency modulation to have a smooth sound, Ocean developed a mapping of the LED average, and Nemo worked on the extra credit song that we mapped to SW[4].

6 Supplemented Materials

How many on-chip resources did you use (LUTs, block RAMs, etc.): Our program used 146 flip flops, 147 slice registers, 3,248 LUTs and block RAMs. We used 2 domains in our final version, consisting of a BITCLK and a system clock, “gclk”.