FPGA Digital Audio Workstation

Nader Jemel
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, U.S.A.
naderj@mit.edu

Charalampos ”Charis” Georgiou
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, U.S.A.
cgeo@mit.edu

***Abstract*—We present a hardware design for a Digital Audio Workstation implemented on an FPGA, utilizing powerful caching techniques and parallelized real-time audio manipulation pipelines. This workstation can record, manipulate and mix sounds from any audio source, such as musical instruments or music players. The user has the ability to play stored tracks, store mixed and manipulated sounds and reuse them. We implement this design using a Nexys 4 DDR FPGA and evaluate its performance by testing it with a keyboard's inputs.**

***Index Terms*—Digital Systems, Field Programmable Gate Arrays, Audio Processing**

I. Goals and Challenges

Our implementation allows the user to record audio inputs and store them in slots called **channels**. The audio is represented following the I2S protocol, using sequences of 8 bits, which we call **words**. The user has the ability to apply **effects** such as delay, echo, distortion... Each channel's modified audio track would be combined into a single output through a process called **mixing**. While mixing tracks, the user can adjust the volume of each channel separately and choose which will be muted.

A. Commitment

Our commitment was to design and implement a system which is able to record an instrument, store it into the SD card, and retrieve its sound. We also promised that our design would be able to apply basic audio effects to the sound, mix multiple tracks before outputting them, and provide a basic graphical user interface. As described in later sections, our implementation satisfies all the above requirements.

B. Goal

Our goal was to have a workstation that would be able to record and playback at a sampling frequency of 44.1KHz, while providing continuous playback and the capability of storing mixed tracks into the system to be reused as audio sources. We aimed to do all these while minimizing the latency of retrieving the audio tracks from the SD cards. All the above goals were met. The only checkpoint that we did not manage to meet was recording and playing audio at different beats per minutes (BPM), although this can be implicitly achieved by the user by utilizing the effects pipeline.

C. Memory Challenges

Storing audio tracks in internal memory would not be feasible since RAM storage on the FPGA is limited and not retrievable after reboot. Instead, we need to use an SD card, which introduces issues with latency since reading from it is much slower than reading from a RAM.

SD cards are split into **sectors**, which are contiguous blocks of 512 bytes. All read and write operations are performed on exactly one whole sector.

Furthermore, extreme attention to detail was necessary when it came to synchronization. Each word must be stored and retrieved exactly once from the SD card, and while crossing clock domains this is something one has to be cautious about.

The most crucial aspect of the memory module design was retrieving multiple channels' audio information for mixing. This meant that we would need to be able to access information from multiple sectors at virtually the same time. All of these had to be done with low latency and while keeping the area used at a minimum.

D. Audio Challenges

The main challenge with audio was dealing with overflow, since we're limited by 8 bits, summing data for effects or the mixer always causes overflow which eventually causes clipping.

II. IMPLEMENTATION DETAILS

The workstation is composed of a Nexys 4 DDR FPGA from Digilent, which has attached an I2S2 Periferal Module Interface (PMod). A 2GB SD Card is plugged into the FPGA. Audio source and output devices will be connected to the PMod using 3.5mm stereo audio jacks.

Our design uses a clock of frequency 100MHz for its main operations, one of frequency 65MHz for VGA output and one of 22.579MHz for audio input and output as we will see below.

We are using **CALCULATE NUMBER HERE** Block RAMs and our worst negative slack is **CALCULATE NUMBER HERE**.

III. Audio input and output

Charis Georgiou

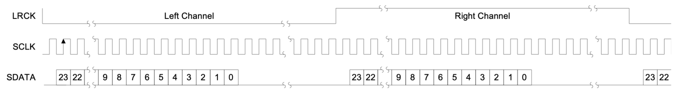
The I2S protocol is used to receive and transmit audio signals from and to the secondary interface. Using a main clock we generate a serial clock and a word clock which are necessary for synchronizing the process.

On each rising edge of the word clock (also called Left-Right clock) a new word is passed between the main and secondary modules. This means that the frequency of the word clock should be equal to the sampling rate.

On each rising edge of the serial clock a new bit is passed between the main and secondary modules. The frequency of the serial clock is l times the frequency of the word clock, where l can be any number greater than the word length.

$$f_{sclk} = l \cdot f_{wclk} = l \cdot f_{sample}$$

In our implementation we are sampling at 44.1KHz using 8-bit words, with $l = 64$. Therefore, we have $f_{wclk} = 44.1KHz$ and $f_{sclk} = 2.82MHz$. To derive these, we are using a main clock with $f_{mclk} = 22.58MHz$, which is 8 times the frequency of the serial clock.



Example waveform of the different clocks [1]

Essentially, our I2S modules will always output the 3 clocks (main, serial and word-select), and either receive or transmit bit-by-bit a word output. For the specific timing and logic configurations we followed the I2S2 Pmod reference [2].

A sensitive issue that came up during implementation was the necessity of transferring information between clock domains, specifically between the 100MHz and 22.58MHz areas. After experimentation and consulting the teaching assistants, we decided to address this by using dual-port Block RAMs, in which each port would be driven by a different clock. That design choice allowed for infallible and synchronized clock domain crossing.

IV. Audio Effects

Nader Jemel

A. Delay

This effect plays back delayed samples of the same audio to create an echo effect. This effect can be reduced to a special case of a finite impulse response (FIR) filter [3]: $Y[n] = X[n] + \alpha X[n - m]$ The user will have the ability to set the amount of delay and the coefficient. To implement this, we save samples in the BRAM, read the delayed sample, and sum it with the input. For this project, we are using 16-bit audio recordings with a samling rate of 44.1 kHz. To save 500ms of samples, we'd need storage of 44100 * 16 / 2 = 352.8 kilobits. Thus, to save enough samples for the delay, we will need 10 BRAMs.

B. Chorus

Chorus is an effect where multiple sounds, in the same tune, are played together approximately at the same time. To create a chorus-like sound, we will use a bunch of delays with different and small latencies. This will provide sounds that are slightly similar, but still in tune and not exactly the same. These samples will be played at the same time to create the warm chorus effect.

C. Distortion

Distortion creates a fuzzy sound by clipping the high and low amplitudes to a certain limit specified by the user. To implement this, we just need to check if the sample is higher or lower than the set limit and change it to the limit if that is the case.

D. Echo

Echo's implementation is very similar to that of the delay. The only difference is that instead of saving the dry input to the BRAM, we save the delayed version. That way, we get an effect that's similar to delay that propagates more over time

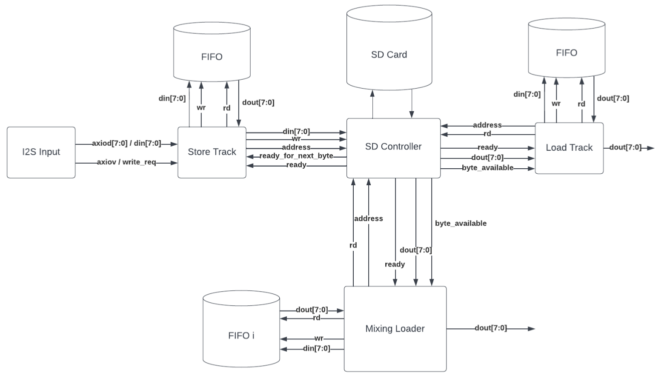
E. Tremolo

The tremolo effect is very similar to flanger it's just that the multiplication by the wave is applied to the current sample, not a delayed one.

V. Audio Track Storage and Playback

Charis Georgiou

In order to record, modify and replay high quality audio we need great amounts of storage; far more than it is reasonable to have on an FPGA. To address this challenge, we are storing all recorded tracks on an SD card.



Block diagram of our design for memory management

By using peripheral storage we encounter the issue of high latency reads and writes. To minimize latency and provide an accurate playback, we use dual-port Block RAMs (BRAMs) on the FPGA, modeling them as caches that will load each channel from the sd card and provide outputs in parallel.

The important tradeoff between space and time is particularly important for this design, therefore we opted for a solution that allows for multiple sound samples to be stored while simultaneously not sacrificing latency.

A. Assumptions

In our design and implementation we assume that all tracks have the same duration. In other words, we work under the assumption that all tracks will be recorded and replayed following the same Beats Per Minute (BPM). That allows us to fix the number of bytes (and sectors) to be retrieved from the SD card.

This is crucial to be able to mix multiple channels in a synchronized manner, since it will allow us to collect the same number of words from each channel, because each time interval will correspond to the same number of words irregardless of the channel.

B. SD Card Details

To interface with the SD card, we make use of an SD Controller module [4] which when requested, will provide or store a sector of 512 bytes, by sending one byte at a time. Because this module uses sensitive timing loops, we need to use a clock at 25MHz, which we can derive by dividing the system's 100MHz clock.

An audio track is obviously composed of multiple bytes, which will be retrieved or given to the SD card when it is ready for new pieces of information. This brings up the necessity for an intermediate module which will load partial words into a buffer before providing them for manipulation and transmission.

C. First In - First Out Caches

Interacting with the SD card is relatively fast when compared with the sampling frequency. For that reason, we will use Block RAMs to store sectors of the SD Card, which will only request to read or store new sectors when needed. We model the BRAMs as first-in first-out (FIFO) caches, which will serve and store words every time they are requested.

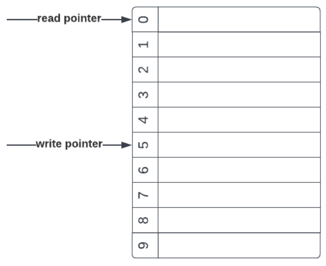


Diagram of first-in first-out BRAM

On request, move the corresponding pointer and loop over once we reach the depth

The buffers will work in a cyclical fashion. As soon as the address they are reading or storing reaches the maximum depth, they will restart from the beginning. This is equivalent to a least-recently-used (LRU) eviction protocol, since each sector is loaded and used in sequential order.

Our design stores left and right channel information in alternating order. When recording, on every edge of the word-select clock (either positive or negative), a new word is written into the buffer. On every edge of the clock when playing sound, a new word is read from it. This implies that left-channel words will be stored in odd positions and right-channel words in even positions of the FIFO (or vice-versa).

D. Storing

Storing is the process where an external module (either the I2S receiver, or the final mixer) will provide input to be written into the SD Card.

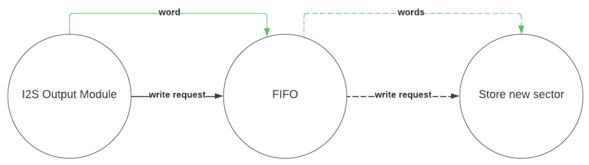


Diagram of storing logic

Every 512 write requests to the FIFO, we make a write request to the SD card

The arrival of a new word is signaled by a write request to the FIFO cache. The words are stored into an intermediate buffer, with capacity about 8 times the sector size of the SD Card (4096 bytes). As soon as a batch of 512 words have been received, a write request is issued to the SD Card, which will accept the new sector of words.

Just as we aimed for, little space is used since the buffer only needs to store a constant amount of SD sectors at all times (we are actually overshooting by allowing it to store 8 times the sector size).

The latency of storing into the SD Card does not affect the overall latency of the module greatly, for two reasons. Firstly, the SD Card operations are driven by a 25MHz clock, which is significantly faster than the 44.1KHz sampling rate. Secondly, by only requesting writes when we have a full sector prepared, we are allowing some "slack time" for the SD card to store the current sector, while we are writing into the buffer the next one.

E. Loading

Loading is the process where an external module (either the I2S transmitter, or the audio effects pipeline) requests to read a track from the SD card.

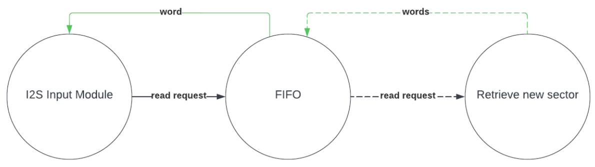


Diagram of loading logic

Every 512 read requests to the FIFO, we make a read request to the SD card to refill

The request of a new word is signaled by a read request to the FIFO cache. As soon as a batch of 512 words has been given to the external module, a read request is issued to the SD Card, which will fill the cache with a new sector of words. The intermediate buffer has capacity about 8 times the sector size of the SD Card (4096 bytes)

For reasons similar to those of the storing module, this way of loading a single audio track is both time and space efficient. Since we are using a block RAM, each read request will be fulfilled only after 2 clock cycles of our main 100MHz clock!

F. Retrieving all tracks simultaneously

This is the greatest challenge of designing the SD card interaction modules. In order to mix tracks, we need to retrieve them in parallel. That is, for each time step we need to receive the next word for each channel.

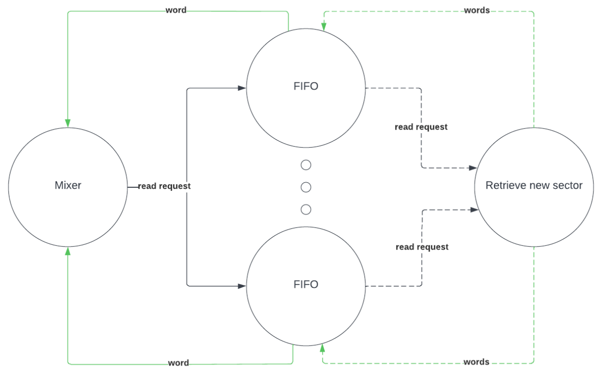


Diagram of logic to load all tracks

Every 512 read requests to the FIFOs, they all make read requests to the SD card to refill

We chose to follow a design that uses one buffer for every channel. Each time the mixer requests for new words from all channels, each FIFO will return the next word for its own channel. Every time a batch of 512 bytes have been delivered to the mixer, each channel's cache will take turns requesting a new sector from the SD card.

To create the looping effect, after a certain amount of sectors we will start from the beginning of each track. That is, we will read sectors $[x, x+k]$ for each track's starting address x (since the SD is evenly split between all channels, x is not hard to find). For 16 second loops, we calculated k to be about 2^{10} based on our sampling frequency.

The increased amount of requests to the external memory poses the danger of delay accumulating, which can lead to the FIFOs not having words loaded into them in time. To address this issue, we pre-loaded the caches with a few sectors as soon as the mixing starts. This allowed for some additional delay to accumulate without breaking our buffers' timing.

This structure allows for low space usage, since each buffer will need to store only a small, constant number of sectors (512 bytes) at any time.

Most importantly, using this design we managed to reduce the latency required to read words from all channels simultaneously, achieving an amortized delay of only 2 cycles of our 100MHz clock!

VI. Graphical User Interface

Nader Jemel

We built an interactive GUI where the user can control all the channels and the mixer. It consists of a table with 5 columns for the 4 channels and the mixer, and 10 rows. The user controls the cursor between the table entries using the buttons on the board. The user can start recording, add and remove effects as needed through the GUI.

A. VGA Details

We'll be using a divider to get a 65 MHz clock from the system's 100 MHz clocks which we'll use to create a GUI with a resolution of 1024 x 768 and a 60 MHz refresh rate. For now, we just have the background be a specific color, with different blocks added on top.

B. Text Generation

Similar to pong game, we used sprited for text, where the output is just two colors.

VII. Mixer

Nader Jemel

The user will be able to set a volume for each channel. The mixer's role is to play all the unmuted and programmed channels at the same time. The volumes can be set at any value between 0 and 63, so we will need 6 bits to represent it. We will then multiply the volume by the output of each channel, and shift it right by 6 bits. Finally, we will sum up all the sample output from each channel and shift right depending on how many channels are active to make up for overflow.

VIII. Retrospective

Designing and implementing this project proved to be a challenging task. Nevertheless, we believe to have made appropriate choices when faced with dilemmas, leading to a satisfying final project. In this section, we describe how we would approach some aspects differently if we had to start again. We also propose some additional features that we could have added to the workstation, if time allowed.

A. Memory management

Our design uses a simplistic way of partitioning the SD card into a fixed number of channels. Looking back at the design with the experience of implementing it, we realize there are more efficient and even simpler designs that allow for more flexibility.

A different approach would be by storing an index table in the first 100 sectors of the SD card, where each sector would hold information about a track, such as its name, starting and ending addresses and the beats per minute that it was recorded. This would make our design more versatile and able to support additional features later on.

Furthermore, while we have a great latency for our multiple track retrieval, we are not able to increase the number of channels arbitrarily, due to the delay of SD retrievals accumulating. One possible way to go around this would be to request for new sectors from the SD card more often, determined by a function of the number of channels (for now, we only request new sectors every 512 reads). By determining a ratio of bytes needed for each new request, we should be able to increase the number of channels without failure.

Having said these, our current design as is can be easily modified to support multiple audio tracks to be ”loaded” by channels, instead of each track always corresponding to a channel, by just using registers to remember each channel's selected track and calculate their starting addresses in the SD card.

B. *Fragmentation*

This technique would be hard to implement but would provide our system with much greater flexibility while overwriting stored tracks. We would utilize an insignificant portion of each SD sector to allow for fragmentation of information. While retrieving an audio track from the SD card, each sector's last 4 bytes would indicate the address of the next sector to be read. To indicate the end of an audio track, we would use a terminal code (such as 32'b0 or an invalid address).

Following this design, each audio track's information could be fragmented across the SD card and not stored in contiguous sectors. This means we could record tracks without worrying about overwriting past information, since with the help of a simple "availability table" we could find the next available SD sector to write into.

C. *Naming Stored Tracks*

An additional feature that we would include would be the opportunity for the user to browse among a large collection of saved tracks and load the selected ones into the audio channels. For the selection to be meaningful, the tracks would need to be named. We could allow for the user to type words by using a PS/2 keyboard and translating its input by utilizing an ASCII lookup table, which could be implemented as a read-only block RAM.

IX. Contributions

We split our system's features in a way that we could work on them independently until the final integration. The research, design and implementation of the audio effects pipeline and mixer was Nader's. Charis had to design and implement the I2S input/output protocols and all memory management modules. As mentioned before, the SD Controller module, which was crucial for our project, was implemented by Jonathan Matthews.

The general block diagram was designed by Charis and we both worked on the presentation. The reports were written by both, each focusing on their own components and Charis writing the joint sections.

Finally, we are thankful for the advice and guidance of the teaching assistants Jay Lang and Fischer Moseley, as well as the professor Joe Steinmeyer, throughout the processes of researching, designing and implementing the project.

References

[1] https://d3uzseaevmutz1.cloudfront.net/pubs/proDatasheet/CS5343-44F5.pdf

[2] https://digilent.com/reference/_media/reference/pmod/pmodi2s2/pmodi2s2_sch.pdf

[3] https://wiki.analog.com/resources/tools-software/sharc-audio-module/baremetal/delay-effect-tutorial

[4] B. Gross, J. Matthews, N. Rodman "Live-Action RC Mario Kart", 2014