XILINX™ Shuffle

Abate, Alex
Brain and Cognitive Sciences Department
Massachusetts Institute of Technology
Cambridge, MA
aabate@mit.edu

Kravtiz, Samuel
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
skravitz@mit.edu

***Abstract*—We present the Xilinx Shuffle, a purely hardware-based MP3 audio playback device implemented on an FPGA. With a 2GB SD card used to store MP3 files, it can hold ~600 songs in memory, and playback the songs at high-definition 44.1 kHz. Our algorithm is built to support all varieties of MPEG-1 Layer III encoded data at a fixed sampling rate of 44.1 kHz. Furthermore, the XILINX Shuffle user interface consists of a play, pause, and song skip button. Though we were not able to complete the full product, we implemented in SystemVerilog up to the penultimate step of conversion from an mp3 bitstream to 8-bit PCM values. This includes frame parsing, Huffman decoding, requantization, antialiasing, stereo conversion (supporting dual channel, joint stereo, mixed stereo, and some combination of the three), an inverse modified DCT, and frequency inversion. Additionally, we implemented the interface to read mp3 bits directly from an SD card.**

Keywords—MP3, MPEG-1, audio playback, lossy compression

I. Introduction

MP3, a file format for audio, is perhaps the most ubiquitous codec in use today. First proposed by the Moving Pictures Entertainment Group (MPEG) in the 1990s, it was standardized by the International Organization for Standardization (ISO) in 1993. The MP3 codec employs a combination of lossy and lossless compression to reduce file sizes typically by a factor of 10 while maintaining discernible audio quality. Additionally, MP3 files are structured in mostly independent chunks which allow for the streaming of data. These qualities have made MP3 the first choice for many digital applications that store and play audio.

For example, Apple's first generation IPOD Shuffle functioned as a dedicated MP3 playback device. Future generations over the next decade held most of the audio playback device market share. However, these devices employed a system on chip (SOC) with a single processor executing software commands. To this end, we introduce the Xilinx Shuffle, an MP3 playback device capable of storing and playing nearly 600 songs on board. Leveraging the streaming decodability format of MP3, our playback device is designed purely in hardware, implementing the entire algorithm without the use of a central processor. As such, our design is optimized, and more efficient than previous generation IPODs for MP3 playback.

II. Components

A. 2 GB SD Card Writing and Reading

As part of the project, there is a python file that allows a user to convert an mp3 file into a .mem file. The user can then copy the data from the .mem file onto the SD card using HxD, while keeping track of the initial offset and the number of frames the mp3 file contains. These can then be inputted into the top level as parameters, and users can use the switches to change which song they would like to play. Once they have selected a song, they can use the up button on the fpga to begin playback. If we had more time, another additional user interface would be to add a switch to act as a play/pause button, which would stop the decoding process after the current frame. We could also have added fast-forward/reverse capabilities by using the left/right buttons to increment or increment which frame will be read next by the SD controller.

The button is part of an finite state machine (FSM) for reading off the SD card, moving from the IDLE to PLAY states. In the play state, the board communicates with the SD card itself using the SPI controller with a 25 Mhz clock. It will continue to read off the SD card until the number of frames read is equal to the number of frames inputted by the user as a parameter. Although we were unable to get the decoding pipeline working on the board itself, the SD state machine would likely also need a flag from one of the later modules to ensure that header and side information data would not get overwritten mid-frame.

From the SD controller, each byte was written into two modules: one which stores the most recent 4 bytes and goes high when a valid header is detected, and a multiplexer that takes in that valid header signal and either passes the data to a module that parses the side information, or into a FIFO buffer to be stored for Huffman decoding.

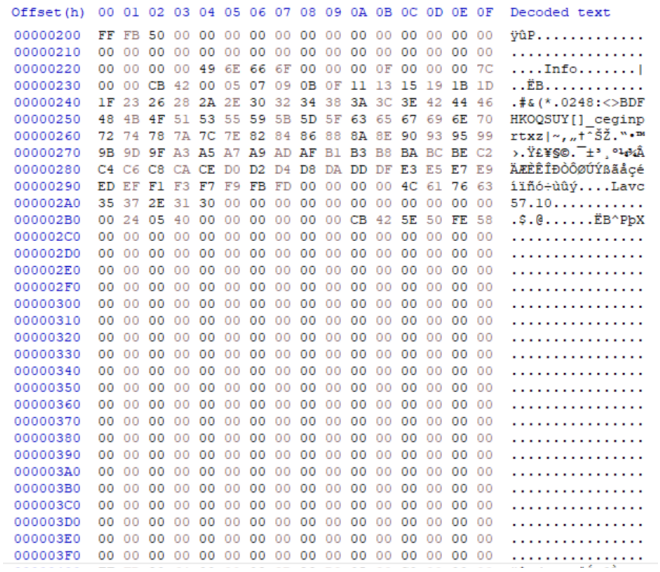

Fig 1. SD card writing mechanism: Example of frame data in SD card sector

B. *Nexys 4 DDR FPGA*

The FPGA will implement all steps of the decoding process and audio reconstruction algorithm. First, it will stream out each MP3 frame from the SD card. Then, it will run the full decoding process on the frame. Finally, it will reconstruct the audio waveform, create a PCM signal, and send the signal to an onboard mono-channel audio jack.

Additionally, the FPGA will have onboard play, pause, and song skip buttons.

III. MP3 Codec Parsing

The MP3 codec (based on MPEG-1 Layer III) was designed to employ two forms of lossy compression, one based on mathematical limits of audio playback, and one based on exploiting biological constraints of human audition. Additionally, the final MP3 codec bitstream is compressed losslessly using Huffman coding. These components contribute to our design decisions for parsing the bitstream of MP3 data coming from the SD card. We can outline 3 components in each MP3 frame.

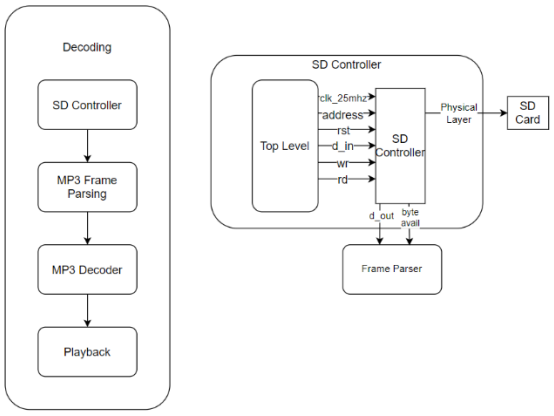

Fig 2: Full System Diagram (simplified)

A. *Frame Header*

The start of each frame consists of a 4-byte header. The header contains relevant information for parsing the side information and the Huffman codes throughout the rest of the frame. For example, our system can handle files with multiple audio channels and variable bitrates. Both of these quantities can be read out based on the header.

Some audio files are also protected by an MPEG-1 CRC checksum of the following form:

$$G(X) = X^{16} + X^{15} + X^2 + X^1$$

Our design incorporates this error checker in parallel and ignores audio frames containing broken data.

B. *Frame Side Information*

Following the 4-byte header information and possibly the 2-byte checksum, our parser then directs the incoming bitstream to two parallel side information decoders (one for single channel audio, one for dual/stereo/joint stereo audio). These occur in parallel because they require different numbers of bytes from the incoming bitstream. The side information also yields important fields from decoding the Huffman codes, including the number of bits allocated to a single granule and channel, the Huffman tables used to decode portions of the filterbank spectrum (based on a modified discrete cosine transform), and the location of the start of the audio data. Note that the information from a single frame may be encoded across several past frames according to a 'bit reservoir' technique to enhance quality in Layer III encoding.

C. *Frame Data*

The frame data consists of scaling factors (which determine how amplified certain frequency bands are) and the cosine transform coefficients for frequency bands. Ultimately, each audio channel is coded into two granules, which are further quantized into 576 different cosine frequencies. The MP3 codec emphasizes lower frequencies by coding them at a higher resolution than high frequencies. Our algorithm can handle multiple common choices for encoding sub-bands of the cosine transform.

To account for the bit reservoir framework employed by most MP3 encoding software, our bitstream parsing module maintains a FIFO buffer of 8000 bits (meant to encompass 3 frames of audio data and thus handle the maximum load and bitrate for MPEG-1 Layer III data). The FIFO buffer allows later components to read out the bitstream from the SD card which corresponds to scale factors and Huffman codes with a delay compared to when the headers and side information are read out. This accounts for the fact that audio data read in one frame may not be useful until the decoder has parsed the header and side information for the next three frames in line.

Our implementation also has error checking at this stage of bitstream parsing. The side information of each frame contains a negative byte offset indicating how many bits from past frame data are necessary for Huffman decoding. In the case where not enough bits are present (if for example, the system starting reading in frames from the middle of an mp3 file and does not

have a reservoir long enough yet) the FIFO buffer is maintained, but the future steps are not implemented and consequently no data is decoded for the PCM.

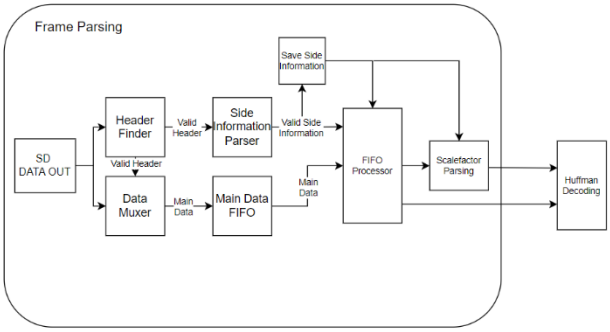

Fig 3: **Frame bitstream decoding Schematic:** The SD card controller streams out one MPEG-1 Layer III frame at a time. The above implementation accounts for multiple possibilities for metadata structure (channels, CRC checksums, and variable bitrates) while also maintaining the maximum size bit reservoir to enable downstream audio reconstruction processes (main data FIFO). Additionally, the side information parameters of the frame are saved when a valid frame is processed, and these parameters innervate almost all downstream modules.

IV. Audio Reconstruction

A. Huffman codes and scalefactors

The FIFO buffer containing the main audio data streams out the bits related to scale factors and Huffman codes sequentially. The bit-wise delay (the number of ancillary bits following the *last* frame) can be determined based on the value *main_data_begin* returned by the side information parsing module. The number of bits within the FIFO module dedicated to a particular channel and granule is also determined by the meta data of the frame (*part2_3_length*). The Huffman codes are decoded via a look-up-table. The ISO MP3 standard defines 32 look-up-tables for the high-accuracy regions of the frequency spectrum, and 2 look-up-tables for the low accuracy regions of the spectrum. These tables also contain variable word lengths. Thus, they are implemented based on their table number and word length combinationally (to support Huffman decoding at 25 MHz).

Note that since the Huffman decoding tables map seuqences of bits to pairs of integer values, we used a sequential logic buffer on the end of the Huffman decoder to slow down the release of data into a single bitstream for requantization.

B. Huffman requantization

The Huffman tables correspond to quantized values of energy across an MDCT spectrum (of size 576). The values are first requantized according to their scaling factors in an elementwise manner using the following equation for short windowed blocks:

$$xr_i = sign(x_i) * |x_i|^{\frac{4}{3}} * 2^{\frac{1}{4}(gain-210-8*subgain)} * 2^{-(\alpha*scale)}$$

and with the following equation for long blocks:

$$xr_i = sign(x_i) * |x_i|^{\frac{4}{3}} * 2^{\frac{1}{4}(gain-210)} * 2^{-(\alpha(scale+pretab)}$$

where xr_i is the ith iMDCT input and x_i is the ith value decoded from the bitstream. The first exponential $|x_i|^{\frac{4}{3}}$ is computed via a BRAM lookup table, with up to 1000 values. The lookup table maps 1000 huffman codes to pairs of floating-point values, with 16 bits dedicated to the significand and 10 bits dedicated to the base (in base 2).

The base of the above formulas was first converted to fixed point precision with Q8_2 numbers to accommodate the ¼ multiplier. Then, after computing the base, the final xr_i was computed as a $Q_{2\ 30}$ number to accommodate the large negative base-2 exponential. After this point, all computation were done using fixed-point $Q_{2\ 30}$ numbers.

Finally, in special cases of short block windows and window-block switching, a reordering of the 576 frequency bands was implemented using a BRAM lookup table.

C. Stereo Conversion

For 1 granule, the 576 band MDCT was integrated between channels to account for mixed stereo and joint-stereo encoding. In the former case, the first channel encodes the mean-valued MDCT spectra, while the second channel encodes the difference between the left and right channel. The true left and right channels can be computed according to the following formula:

$$L = (X_{ch1} - X_{ch2}) * \frac{1}{\sqrt{2}}$$

Intensity stereo values can be computed according to the following formula:

$$u_I = \tan(X_{ch1} * \frac{\pi}{12}),\ L = X_{ch1}\frac{u_i}{1+u_i},\ R = X_{ch2}\frac{1}{1+u_i}$$

Here, the tangents were also implemented according to a combinational lookup table..

D. Antialiasing

The antialiasing modules operated by first compiling the stereo conversion results from the previous module into a BRAM of 576 different 64-bit items (the first 32 bits corresponds to the first channel, the second 32 bits correspond to the second channel). An aliasing lookup table BRAM encoded the antialiasing parameters for every frequency band in the MDCT, so once the module compiled an entire granule of data (across 2 channels), it streamed out a new set of anti-aliased values. Since these values were in general not ordered according to their position in the MDCT, a different reordering module saved streamed-out pairs of 32-bit integers into a BRAM and read them off in ascending frequency band order. Also, since the rest of the pipeline processes the left and right channels independently in parallel streams, a 3rd module compiled both the granules from a single channel into one BRAM for readout into the rest of the pipeline.

E. Inverse modified discrete cosine transform

The iMDCT was computed by implementing a high-dimensional sate-space system matrix multiplication, where a store buffer (a BRAM) contained the last state (576 different 32-

bit fixed-point values) and 3 separate BRAMS contained 3 possible sets of 18 different cosine lookup tables. Our system combinationally implemented the matrix multiplication of a vector of 36 32-bit integers and use sequential logic to select the different cosine lookup-tables according to frequency band position. Since the data stream out of the module was again single 32-bit numbers,

Note that to support a more stream-like process, we computed the full iMDCT in chunks according to their matrices, and added the result to one final summation which eventually was saved into a BRAM for output.

F. Frequency Inversion

Our final implemented step in the pipeline was to invert the modified discrete cosine transform results according to the placement in 18 different subbands. More specifically, we multiplied the output of the iMDCT by -1 in certain cases according to their placement. This module assumed a correctly ordered input which reset at the onset of a new frame of data from the SD card.

G. PCM synthesis (by subbands)

Though our implementation did not reach the synthesis step of PCM samples, out plan of action included first compiling subbands of incoming data from the iMDCT, then loading in the subbands, and finally computing a matrix multiplication and linearly combining the result with a stored state vector to output 64 PCM samples at a time. These PCM samples would be encoded in the form of 8-bit integer values, signifying the relative duty cycle of a PCM wave.

This implementation would require 3 more BRAMs, one to store the subband data incoming from the iMDCT module, one to save and retrieve the matric coefficients in $Q_{2\,30}$ format, and one to store the state vector which is linearly recombined.

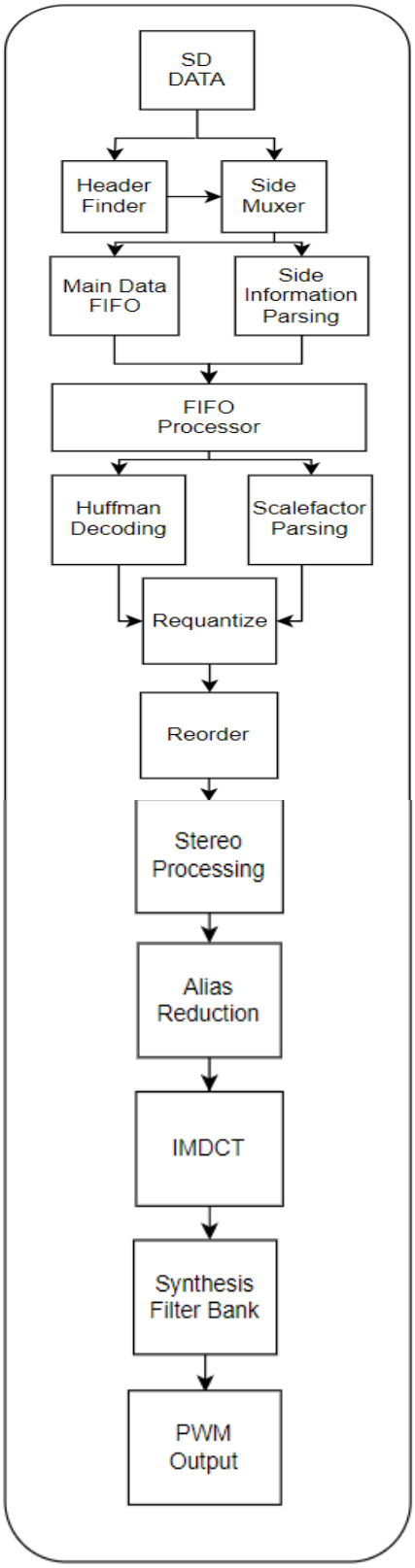

Fig 3. Audio Reconstruction Pipeline: The mp3 bitstream has is decoded over 7 consecutive modules, some of which require

coupled channel data, and some of which require coupled granule data. Thus, our system implements several intermediate modules which criss-cross the data streams after fully compiling them. After a typical frame is detected, our system requires $148\mu s$ to reach the output of the

V. Audio Playback

Leveraging an onboard audio jack, we planned to convert the frame's audio sample into a PCM to be ported to the speaker system. Since our algorithm handles MP3 files encoding multiple audio channels (including dual channel, joint stereo, and intensity stereo), the waveforms will be first combined into a single waveform and then shipped to the audio playback module.

VI. User Interface

We planned to implement a user interface system that consisted of a play button, a pause button, a song skip button. The play and pause can be implemented by starting or halting the frame reading process once a single frame is completely read through. The song skip button will be implemented as an integer defining the address of the starting frame of a song number (encoded as a lookup table in the first chunk of SD card data).

Not that our frame parsing implementation innervates all downstream processing modules (after the side information parser) with the valid frame signal, and consequently uses that signal to reset the state variables of the module where possible (in some instances, a BRAM storage vector cannot be overwritten upon new frame information, but this effect should only contribute to interfering with a single frame, or 26 ms, or audio data upon pressing the play button).

VII. Retrospective System Improvements

Although our design is fully synthesizable, it exceeds the resources that are available for the Nexys A7 board. In our case, we almost double the available lookup tables, as well as the amount of DSP blocks on the board. Our implementation of the Huffman lookup tables combinationally likely overused the number of necessary carry bit adders by a factor of 15 times. Additionally, our hybrid synthesis procedure combinationally implemented a vector multiplication of 2 64 item (32-bits per item) vectors that were consequently sequentially saved into a third 64-item 32-bit vector. Though our implementation allowed for a massive hardware speed up (the decoding up to frequency inversions occurred in $148\mu s$, when a single frame of data plays in the PCM for $26x10^3\mu s$), this came with a massive resource usage boost. We see our implementation as possibly supported multiple mp3 data streaming operations sequentially.

Given more development time, we could reduce the carry-bit usage by the Huffman lookup-tables by a factor of 15 by employing a more flexible lookup table operation that can take in a variable number of significance bits (which is otherwise set as a parameter right now).

Our code can be found at github.com/ samuelkravitz/ 6.111_Final-Projects, and an implementation of the SD card usage can be found under the folder Sam/ sd_tester, and an implementation of our full stack decoding process can be found under Alex/ test_builds/ final_build/.

VIII. Reflection

For this project, I worked on the Verilog implementation of handling the mp3 bitstream after it is inputted into the system and a valid header has been determined. Sam worked on developing the SD card interface and the initial parsing measures to robustly detect the bitstream.

References

[1] Information Technology—Coding of moving pictures and associated audio for digital storage media at up to about 1.5 MBits—Part 3. ISO/IEC 11172-3. September 1993.

[2] Sripada, Praveen, MP3 decoder in theory and practice. *Computer Science*, 2006.

[3] Hoshi, Hajime, Go-mp3 Decoder, github.com/hajimehoshi/go-mp3