

Project Proposal

Description

For years MPEG Audio Layer 3 (MP3) has been the dominating audio encoding. Due to legal restrictions with the MP3 format, many people are turning to a free and open alternative known as Ogg Vorbis. While there are now many software Vorbis decoders, hardware implementations are still scarce and underdeveloped. We will create a Vorbis decoder that fully implements the Vorbis I specification in hardware. The decoder will initially read Vorbis files from a preprogrammed Flash ROM. If time permits, we will develop the decoder into a complete Ogg Vorbis player with standard playback controls, possibly adding an equalizer and an interface to read files from a SecureDigital card.

Implementation

The Vorbis decode process consists of many steps that must be executed in an intricate order. Decoding an audio file will involve processing 3-5 MB of data within a few minutes. By performing abstract mathematical functions on such a large amount of data, a good digital design will be crucial to the success of the project. Without the ability to test each part of the algorithm, and possibly each subpart, it would be virtually impossible to track down and fix bugs. For this reason, we will implement the decoder in a very modular manner. We will test each module with reference data to verify that each works to specification.

The decoder will be broken up into roughly four sections. The decoder will handle Vorbis audio that is embedded in an Ogg stream. Ogg is the most common transport stream used to encapsulate Vorbis. The data will first enter the Ogg preprocessor. Since we are dealing with a very large number of bits, the data will be held in the ZBT RAM and block memories. The Vorbis packets are extracted from the Ogg stream and its integrity is checked. The Ogg Preprocessor extracts the Vorbis packets and stores them in the ZBT RAM. The Ogg Preprocessor will then initialize the Vorbis decoder front end.

The front end reads the configuration packets, preparing the decoder for the audio data packets. The probability model, including the Huffman coding and the vector quantization codebooks, is dynamically generated from the header packets of each Vorbis file. The codebooks will be stored in the RAM. A memory abstraction layer will be used to let modules get and set the data according to its type and structure. For example, Huffman codings produce a natural tree structure, so the codebook will be stored in a way to take advantage of this. The memory abstraction layer will then provide a higher level method of accessing the value associated with each coding. The layer will also provide a simple interface for accessing the various configuration settings (floor type, residue type, etc.), which will be held in the block memories.

As the front finishes preprocessing the data packets, the contents of the packets are passed to the Vorbis CODEC's back end. The back end is responsible for reconstructing the audio signal from the encoded bits. This process includes reconstructing and combining floor and residue signals, performing the monolithic transform back to the time domain, and overlapping the audio from the consecutive audio blocks. As the PCM stream is recreated, the back end will send it to the ac97 chip to be played on the speakers.

Specifications

The top-level Ogg Vorbis audio decoder module is designed to take in a bitstream of Vorbis-encoded audio data, encapsulated in accordance with the Ogg specification. It produces as output a bitstream of PCM audio data. The sampling rate, channels, and sample size of the PCM audio data are determined by the properties of the input – a stream representing a 44.1 kHz stereo 16-bit signal will produce 44.1 kHz stereo 16-bit PCM data.

In addition to the Ogg bitstream, the decoder module will take start, clock, and reset signals. The start signal, when raised, will begin decoding of the data on the bitstream input. The single clock input signal is used for synchronous operations within and between the decoder's submodules. The reset signal is used to revert the decoder to its power-on state. When the reset signal is asserted, any decoding that is occurring stops, all configuration data and buffers are destroyed, and the internal states of each of the decoder's submodules return to their initial values. Note that the start and reset signals are synchronous, and that the start, reset, and clock signals are globally available to each module.

Specifically, the process of decoding begins with the Ogg preprocessor. Ogg is simply a data encapsulation format, used for transmitting and verifying data. The preprocessor is used to perform the necessary verifications and extract the encapsulated Vorbis-encoded data packets from the Ogg bitstream. The preprocessor uses external memory – in this case, ZBT RAM – to store the Vorbis packets. When a packet has been fully placed into memory, the preprocessor signals for the front-end module to begin processing the Vorbis packet, and waits for a done signal from the front-end module. The front-end module determines the type of the Vorbis packet and delegates further processing to the appropriate submodule.

Vorbis specifies four types of packets. The first, the identification header packet, is used to provide general stream-related information like the bitrate of the encoded data and the block sizes used for back-end processing. The second, the comment header packet, provides optional human-readable descriptions of the audio signal, such as track title, artist, or genre. The setup header packet contains a wealth of information related to decoding, including Huffman and vector quantization codebooks, a set of base energy curve configurations (called “floors”) and a set of configurations for the spectra resulting from floor removal (called “residues”). The final type, the audio packet, contains a portion of the compressed audio itself.

When processed, the first three types of packet ultimately place the decoded information and configuration data in external memory. Audio packets are delegated to the back-end, which uses the stored configuration data in conjunction with the audio packet contents to reconstruct a frequency-domain representation of the audio signal for each channel. Finally, an inverse modified discrete cosine transform is applied to the frequency-domain representation to obtain a time-domain representation, which is slightly overlapped with the previous time-domain fragment and is placed on the top-level module's output as finished PCM audio data.

Testing

Since this project involves a large number of small, extremely specialized modules, testing will thus primarily involve the development of a set of testbenches for each individual module. Testbenches will provide a set of inputs to each module, and will compare the module outputs with data obtained from a reference software Vorbis decoder. Testbenches will additionally test functionality not explicitly related to stream data, including proper switching of states and assertion of inter-module signals.

Higher-level encapsulating tests will also be used to ensure the correct inter-operation of

submodules. For example, a front-end test might send some reference Vorbis packets to the front-end module, and ensure that the appropriate memory locations are written to (for header packets) or that the back-end module is properly signaled (for audio packets). The Ogg preprocessor can be tested to ensure that all encapsulation data is removed and that the appropriate signals are asserted when invalid data is encountered.

Finally, since it is extremely unlikely that our decoder will produce PCM output that is bit-for-bit identical to the software reference decoder, overall operation of the decoder will be verified via a listening test. Especially when working with audio encoded at a low bitrate, it is often easy to detect the relative level of quality of a decoder. Beyond simply verifying that something resembling the encoded data is produced, a thorough comparison of outputs of the hardware decoder and software reference decoder for different sampling rates, bitrates, and sample sizes will suggest the need for fixes or implementation improvements.

Project Assignments

Jon will be assigned to the front end, including the Ogg preprocessor and memory abstraction layer.
Matt will be assigned to the back end.

Block Diagram

