OGG VORBIS BLUESPEC IMPLEMENTATIN

MAREK DONIEC

Contents

Key words and phrases. Ogg, Vorbis, Bluespec, Verilog, Hardware, FPGA.

The author would like to thank Myron King and Kermin Fleming for making their IMDCT implementation available. The IMDCT core used in this project is developed entirely by them. For continious support throughout the project I would like to thank Prof. Arvind and Abhinav Agarwal.

1. Introduction

In this work a Bluespec[1] implementation of an Ogg Vorbis[3] decoder is presented. Ogg Vorbis is an open source, royalty and licence free audio codec developed and maintained by the Xiph.org Foundation [2]. Because of its very serial structure most currently available implementations, even those for embedded systems, are processor based, frequently with an external inverse modified discrete cosine transform [7]. This project aims to move all parts of the decode process onto an fpga with the ultimate goal to explore the possibily of parallizing the decode process and making it run in hardware at low clock frequencies. As a first step a Bluespec implementation is presented and tested in simulation. The fixed point decoder *Tremor* is used as a reference decoder [6]. All modules but the inverse modified discrete cosine transform are implemented purely in Bluespec and can be thus compiled into verilog and further into hardware. A short analysis is given on what needs to be done to make this implementation fit into an FPGA.

This report is structures as follows. The second section Describes the Ogg Vorbis audio format and give a rough outline of the decode process. The third section presents a Bluespec implementation of an Ogg Vorbis decoder. In the forth section the test environment is described. The final section presents results and gives a short discussion.

2. Ogg Vorbis

The Ogg Vorbis audio codec actually consists of two formats, the Ogg media container format and the Vorbis audio codec. Both formats have been developed by *Xiph.org*, are open source, as well as royalty and license free [2].

2.1. The Ogg Container Format.

Ogg is a multimedia container format. I is used to encapsulate compressed data from other codecs. This can be a simple Vorbis audio stream, mutliple audio streams, or mutliplexed audio and video streams. Ogg splitts the data stream into ogg packets. The structure of each packet is as shown in table 1. Each ogg packet can contain up to 255 segments of the underlying data stream.

2.2. The Vorbis Audio Codec.

Vorbis is the actually audio codec used to compress the audio signal. The current specification is available at [4]. This section startes with a very short overview of the decode process and proceeds to explain more technical details in the following subsections. For an in-depth definition of Vorbis please refer to the specification.

2.3. Overview of the decode process.

A single Vorbis stream can contain up to 255 audio channels. The encoded audio stream is splitt into frames of a given blocksize, where up to two different blocksizes are used in one Vorbis stream. Each frame contains information for all encoded autio channels and is stored in a seperate Vorbis packet. For each channel in every frame a floor curve and a residue vector are reconstructed from the logical bitstream. The floor curve contains the rough audio spectrum, while the residue vector contains the details. If multiple audio channels are being encoded then inverse coupling is performed after the residues have been decoded. After this the floor and residue curves are combined for each channel by performing a per entry dot product of the two. The result is the audio spectrum for each channel in this frame and is fed into the inverse modified discrete

pos	bytes	field
0x0000	4	capture pattern "'Oggs"'
0x0004	1	stream structure version
0x0005	1	header type flag
0x0006	8	absolute granule position
0x000e	4	stream serial number
0x0012	4	page sequence number
0x0016	4	page checksum
0x001a	1	page segments n
0x001b	n	segment table
0x001b + n	m	packet data segments

TABLE 1. Ogg multimedia container format packet.

cosine transform (IMDCT). Finally the output of the IMDCT is windowed using a fixed window function and overlapped with data from the previous frame to form the output.

The data for the floor curves and residue vectors is encoded using Huffman coding and vector quantization (VQ) tables. However as oposed to other audio codecs, like MP3, Vorbis does not assume a fixed probability or physioaccoustic model. This means that the Huffman trees and VQ tables used are not know a priori but are also encoded in the Vorbis stream. In particular, the first three packets of the stream are header packets that contain this setup information. All the packets after this are audio packets.

2.4. **Vorbis Header Packets.** Every Vorbis audio stream contains three header packets, namely the identification header, the comment header, and the setup header. These packets have to be stored in just that order. Otherwise the stream is not decodable. Every packet starts with a packet type byte (0x01, 0x03, 0x05 respectively) and the 6 bytes long identification string "'vorbis"'. The following informatin is specific to each packet.

2.5. **Vorbis Identification Header.** The identification header contains information about the Vorbis version used, the number of audio channels, the sample rate, information about the bitrate of the encoded stream, and finally the two blocksizes used in this pariticular stream. The Vorbis version supported by this decoder is the only currently existing version, namely Vorbis I, encoded with a 0.

2.6. **Vorbis Comment Header.** The comment header contains metadata like song name, artists name, or record label. This field is not needed for decoding, but needs to be identified and skipped properly to guarantee alignment.

2.7. **Vorbis Setup Header Decode.** The setup header contains the majority of the information needed to initialize the decoder. It contains the codebooks, the VQ tables, floor setup information, residue setup information, mappings, and the audio packet modes used.

First up to 256 Huffman codebooks are encoded using a path-length coding scheme. For more details please refer to the Vorbis I specificatin [4]. For some of the codebooks a VQ table is also stored in this header. The VQ table is used to map the codewords from the codebook into floating point vectors. The size of these vectors is called dimension. Two different ways are used to store the corresponding vectors. In type 0, if a codebook has n entries, and dimension d, the for every codeword the entire floating point vector of d values is encoded in the

header, resulting in a total of $d * n$ values for that particular codebook. In type 1 only $log_d(n)$ floating point values are encoded along with the codebook, and the codeword in base d representation is used to index into these values.

Next the floor setup information is stored. Up to 64 different floors, each of one of the two possible types can be stored. Since floor type 0 is computationally much more infeasible that floor type 1, floor type 0 is only used in old versions of the official Vorbis encoders. Thus most decoders only support floor type 1 and floor type 0 is skipped in this analysis. Floor type 1 is encoded using a piecewise straing-line representation. The setup header contains all the x coordinates of the support points, while the Y values are encoded in each audiopacket. Further the setup header contains the codebook number to be used for each of the different floors.

After the floor setup the residue setup information is stored. Up to 64 different residues, each of one of three possible types can be stored. While the types differ slightly in the permutation of the encoded vectors, there is little difference in how these vectors are computed and no difference in the setup information. Setup information for each residue contains the size of that residue vector, into how many partitions it is splitt, and what classbooks are used to decode that residue.

Information about up to 64 different mappings follows. Mappings specify how inverse coupling is performed in the case of multichannel audio. Each mapping also specifies what floor curve and what residue vector is to be used for every channel.

Finnaly up to 64 possible modes are stored inside the setup header. A mode specifies which of the two possible blocksizes and what mapping to use for a particular audio frame.

2.8. **Vorbis Audio Packet Decode.** Each Vorbis audio packet encodes one frame of audio for all audio channels encoded in the current stream. Packets start with a single type bit that always needs to be 0. Then the mode number used in this packet is stored. The selected mode contains information on what floor curves and residues to use during decoding and how to perform inverse coupling before feeding the resulting audio spectrum into the IMDCT module. If the mode dictates the larger blocksize for this frame, two addional bits in the audio packet specify if the previous and/or next frames will be of smaller blocksize in which case special care needs to be taken during windowing to align the frames correctly. After these two bits have been read the packet is valid no matter what happens. If the packet end unexpectedly after this point, then an empty frame is returned or whatevery has been decoded so far. This is possible, because audio packets contain the corse information at the begining (floor curves) and the finer grained information is stored towards the end (residue curves). What adds further to this fact is that residue curves are stored in up to 8 passes, refining the signal further in each pass, and the passes are encoded one after the other.

2.8.1. *Floor 1 Decode.* In Vorbis the floor is a vector that very roughly approximates the frequency domain vector for a given sound frame (called spectral envelope curve). During floor 1 curve decode the codebook selected by the used floor curve is used to decode as many Y values from the data stream as there are X values from the setup header. An algorithm called Bresenham's algorithm and closely specified in the Vorbis specification is then used to generate the entire floor curve.

(a) Windows of the same size.

(b) Windows of different size.

FIGURE 1. Windowing and output overlap. Source: [4]

2.8.2. *Residue Decode.* In Vorbis the residue is a vector that represents the fine detail of the audio spectrum for a given audio frame that is left after the encoder subtracts the floor curve. There are three types of residues which differ only in the way the vectors are interleaved during decode. Residues are decoded in up to 8 passes per audio channel. The residue selected by the mapping specifies into how many partitions the residue vector is splitt and in how many passes each partition is contained. If the dimension of the codebook used is d then for every pass and partition of size p a total of p/d codewords are read and transformed into vectors using the VQ lookup table attached to the current codebook. Initially the entire residue vector is filled with zeros and the decoded values from each pass are added to the current vector.

After the residues have been decoded there is no more information need from the audio packet, it is only terminated by a framing bit that should read 1. The entire further decode process is purely computational.

2.8.3. *Inverse Coupling and Dot Product.* Acording to the mapping specified residue vectors can be coupled in this step. For each coupling step the mapping specifies which two residue vectors to combine. These vectors values are then compared pairwise and added/subtracted from each other according to a simple algorithm described in the Vorbis specification. This step has only meaning if there are two or more audio channels encoded in the data stream.

After inverse coupling the floor vectors are multiplied on a per-element basis with the residue vectors for each channel. This results in the fully reconstructed audio spectrum for this audio frame.

2.8.4. *Inverse Modified Discrete Cosine Transform.* The IMDCT is performed for each channel as described in Sporer et al. [10]. Not that if the blocksize of this frame is n, then the data input to the IMDCT is of size $n/2$ and the result is of size n.

2.8.5. *Windowing and Output.* The resulting audio frames from the IMDCT are overlapped for each channel with half of the previous frame if both are of the same size. If both frames are of the same size, they are overlapped halfway as shown in figure 1(a). If they are differently sized they are aligned as shown in figure 1(b). The function used for windowing is

$$\sin(0.5 \cdot \pi \cdot \sin^2(\frac{(x+0.5)}{n} \cdot \pi))$$

, however most decoders use a lookup table.

3. Implementation

This section describes the structure of the decoder implementation and gives a high level walkthrough of the code.

3.1. **The Decoder.**

The module *mkVorbisDecoder* contains the entire decoder. To the outside it receives an Ogg Vorbis stream and outputs decoded a decoded PCM audio stream. It has the following interface *IVorbisDecoder*:

```
interface IVorbisDecoder;
  method Action reset();
  interface Client#(InstReq,InstResp) imem_client;
endinterface
```

Notice that this is the minimal interface used for development and testing. If the decoder is to be used in an FPGA for example, a seconder memory interface or a Get method will be needed to extract the PCM data. Currently all outputs are written as debug information using the `$display` command.

After power up the decoder is inactive. A call to the method `reset()` will initialize the state of the decoder and start it. After a call to *reset() mkVorbisDecoder* will start sending memory requests starting at address `0x00000000`. When these requests are answered *mkVorbisDecoder* immediatly proceeds with decoding the received data. If the decoder is already running when *reset()* is called (i.e. *reset()* is called a second time) the call will cause the current decoder process to stop immediatly and the decoder will restart decoding at address `0x00000000`. As mentioned already above, this is a minimal interface. If needed, simple enable and stop methods can be added easily since *mkVorbisDecoder* is heavily state based internally and the methods need only change the `state` variable.

The decoder module containes many submodules that facilitate certain parts of the decode process. These modules and their functionality are:

(1) **mkOggReader** serializes the incomming data stream and decodes and removes the Ogg headers from it. The information from the Ogg headers is used to segment the resulting stream into Vorbis packets which are then read by mkVorbisDecoder.
(2) **mkVorbisMemoryBank** contains read and write methods for most of the variables used by the decoder and serves as a storage component.
(3) **mkVorbisTables** contains the windowing function lookup tables.
(4) **mkVQTable** computes and stores the VQ tables from the setup header information. This module is also used during residue vector decode.
(5) **mkFloor1Table** computes lookup values from the setup header information needed for quick floor1 decode. This module is also used during floor vector decode.
(6) **mkCoupling** performs the inverse channel coupling after the residues have been decoded.
(7) **mkIMDCT** performs the inverse modified discrete cosine transform. It is the only module that is not written in pure Bluespec, but uses C based lookup functions.
(8) **mkOutput** takes care of audio frame overlapping. It is also the module that should contain the output interface. Currently the PCM stream is output using `$display` statements.

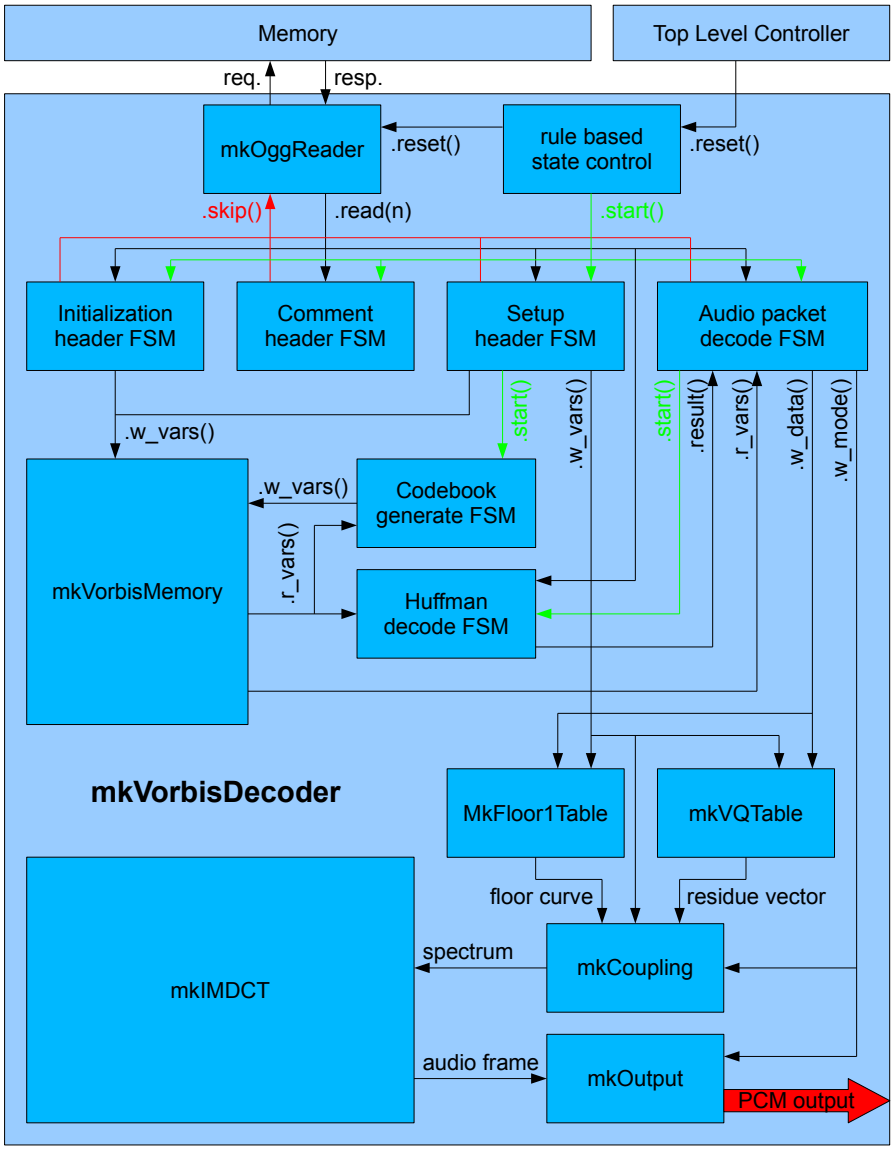

FIGURE 2. *mkVorbisDecoder* module outline. Data lines are black. The *skip segment* signal is red, and finite state machine *start* signals are green. High level control of the decoder is organized as a rule based state machine that activates finegrained controll finite state machines for initialization, comment, setup, and audio packet decode, as well as codebook generation. *mkVorbisMemory* stores setup information related to codebooks and general control. *mkFloor1Table* and *mkVQTable* store setup information for floor and residue generation. They are also invoked by the audio packet decode FSM to compute the floors and residues for a given frame. *mkCoupling* performs inverse coupling and dot producting to create the spectrum for each audio channel. The spectrum is processed by the *mkIMDCT* module to create the current audion frame and overlapped in the *mkOutput* module to generate the final PCM output.

The usage of these modules depends on the state of the decoder. The state of the decoder is controlled by multiple finite state machines (FSMs) that are in turn controlled by a singled state variable, called *state*. Each of the possible states that *state* can be in activates and runs a different FSM. When that FSM is done it changes *state* to the appropriate next state. These states and FSMs are:

(1) **VH_none** is the inactive state in which the decoder does not do any work. No FSM is associated with this state. This state can only be exited with a call to *reset()*.
(2) **VH_initialization** is active while the decoder is processing the initialization header packet of the Ogg Vorbis stream. The FSM *fsm_initialization* is started when this state is entered. Once the initialization packet has been processed the state switches to *VH_comment*.
(3) **VH_comment** is active while the decoder is reading the comment header packet of the Ogg Vorbis stream. The FSM *fsm_comment* is started when this state is entered and switches the state to *VH_setup* once the comment packet has been read.
(4) **VH_setup** is active while the decoder is reading the setup header packet of the Ogg Vorbis stream. The FSM *fsm_setup* is started when this state is entered. *fsm_setup* is responsible for initializing the modules *mkVQTable*, *mkFloor1table*, *mkCoupling*, and *mkOutput*. Also it writes setup information related to the codebooks into *mkVorbisMemoryBank*. When *fsm_setup* is done it sets the state to *VH_runinit*.
(5) **VH_runinit** is used by the decoder to run additional processing of the header data before the audio decoding begins. Currently this include the generation of the Huffman decode trees from the data stored in *mkVorbisMemoryBank*. When this step is done the state variable is set to *VH_audio*.
(6) **VH_audio** is the last state of the high level control. During *VH_audio* the FSM *fsm_audio* is running. Every runthrough of *fsm_audio* corresponds to one Ogg Vorbis audio packet and decodes a single frame. When *fsm_audio* is done it is simply restarted by the high level control. The state *VH_audio* can only be left through outside influences, like a call to *reset()*.

All FSMs are contructed using the *StmtFSM* package, except for high level control, which is done using rules.

The following subsections will give an overview of the data flow from the incoming Ogg Vorbis data to the raw PCM output. It will highlight the role of the different FSMs and modules in *mkVorbisDecoder* at each point. We start with *mkDataReader* and *mkOggReader* that are responsible for serializing the data stream into a bitstream and removing the Ogg headers from it.

3.1.1. *Data Serializing and Ogg header Decode.* The outline of the *mkOggReader* module and its submodule *mkDataReader* can be seen in figure 3. *mkOggReader* connects to a memory or cache on one side and offers read methods to access a serial bitstream on the other side. It removes the Ogg headers from the bitstream and stores the segment size tables to allow the Vorbis decoder to skip to the next data packet when needed. This is the case at the end of almost every packet, since the segemnts in an Ogg packet are byte aligned, but the Vorbis stream is bitpacket. Thus when a Vorbis packet has been decoded there are likely a few unused bits left in the last byte of the segment. *mkOggReader* guarantees that the decoder does

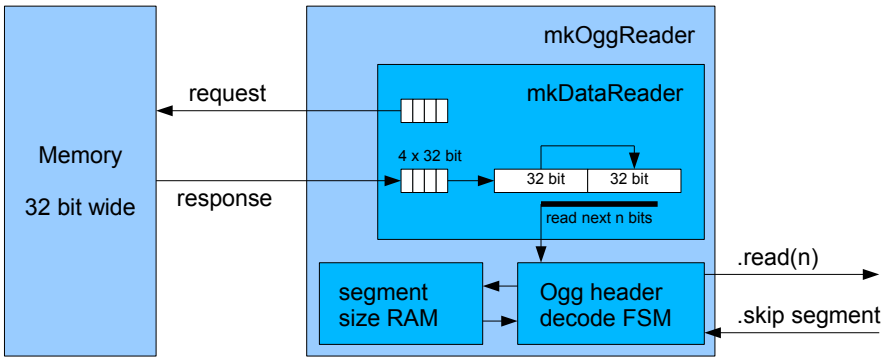

FIGURE 3. *mkOggReader* module outline. The moduel encapsulates *mkDataReader* that acts as a bitstream queue. *mkDataReader* sends requests to memory and receives 32 bit wide words that it stores in a 4 element queue. In addition it has a 64 bit wide register that acts as two 32 bit registers. This register is indexed by the current read position. A call to *.read(n)* will return n bits from the current position and will advance the index pointer. If needed the registers are shiften by 32 bits and the left part is refilled from the queue. *mkOggReader* works with this bitstream. It looks for Ogg headers and decodes them to find the number and sizes of segements to follow. *mkOggReader* also outputs a bitstream that can be read with a call to *.read()*, however it keeps track of segments and allows to skip to the beginning of the next segment with a call to *.skip()*.

not need to take care of this, instead it simply calls *.skip_segment()* when it is done with a segment and continues reading bits immediatly from the next segment.

The current implementation of *mkOggReader* provides a total of 6 different bit reading functions:

```
method ActionValue#(Bit#(8))  read_full_byte();
method ActionValue#(Bit#(16)) read_full_short();
method ActionValue#(Bit#(32)) read_full_int();

method ActionValue#(Bit#(8))  read_byte(Bit#(3) n);
method ActionValue#(Bit#(16)) read_short(Bit#(4) n);
method ActionValue#(Bit#(32)) read_int(Bit#(5) n);
```

The first three methods read 8, 16, and 32 bits respectively and return them directly. The last three methods read between 0-7, 0-15, and 0-31 bits and return a zero-padded value of 8, 16, or 32 bits length respectively. Reading 0 bits does not advance the read position in the file and returns a zero filled value. This is actually usefull in instances where the decoder is supposed to read variable sized entries and a zero-length entry is supposed to have the value 0. This happens for example when the mode number to be used is read at the beginning of every packet.

Note that these 6 methods could all be replaced by a single method with the following signature:

```
method ActionValue#(Bit#(32)) read_int(Bit#(6) n);
```

The decoder would then have to just crop the output to the bit length it needs. The advantage is that only one 32 bit wide bus will be created in hardware, as oposed to a total of 112 bits of buslines for the current 6 method implementation. The current implementation was chosen because of ease of use in the decoder and because it does not significantly effect the simulation negatively. If the decoder is to be run of an FPGA however, the one method version should be enforced.

3.1.2. *Initialization Header Decode.* After a call to the *mkVorbisDecoder reset()* method it resets *mkOggReader* and starts the FSM *fsm_initialization* which waits until it can read bits from *mkOggReader*. *fsm_initialization* then proceeds to check if the first packet is a valid Vorbis information header (type byte is **0x01** folloed by "'vorbis"') and reads the values stored in it. The most important value obtained is the amount of channels encoded in this Vorbis stream and the blocksizes used. All information obtained from the initialization packet is stored in *mkVorbisMemoryBank*. When all values have been read *fsm_initialization* calls *mkOggReader.skip_segment()* and sets the decoders state to *VH_comment*. This starts *fsm_comment*.

3.1.3. *Comment Header Decode. fsm_comment* only checks if the comment header packet is a valid (type byte is **0x03** followed by "'vorbis"') and immediatly calls *mkOggReader.skip_segment()* and sets the decoders state to *VH_setup*. This starts *fsm_setup*.

3.1.4. *Setup Header Decode. fsm_setup* start by checking if the setup header packet is a valid (type byte is **0x03** followed by "'vorbis"'). It then reads the number of codebooks contained. For each codebook it reads the dimensions, the number of entries, and the codeword lengths. There are different ways that this information is stored, but in the end the codeword lengths uniquely identify the Huffman tree. For more information please refer to the Vorbis specification [4]. This information is stored in *mkVorbisMemoryBank*. For every codebook *fsm_setup* also reads the lookup type that specifies if and what VQ table is used with that codebook. If a lookup table is used, all information needed to generate this VQ table is also read from the packet and stored in *mkVQTable* along with the lookup type. *fsm_setup* then calls *mkVQTable.compute_current(i)* passing the number of the table to be computed from the given information.

After this the number of vorbis time domain transforms is read and a type for each. The expected value is 0 time domain transforms, since the current version of Vorbis does not support time domain transforms. This is merely a placeholder for future versions.

In the next step floor setup information is read from the packet. This includes the number of floors, a floor type for each floor, and the appropriate information according to floor type. As mentioned before both floor type 0 and floor type 1 are supported at this stage of the decoder, however only floor type 1 can actually be decode from the audio packets. Floor type 1 information needed to later decode the Y support points of the floor curve is stored in *mkVorbisMemoryBank* since it will be directly accessed by *mkVorbisDecoder*. This is mainly information regarding as

to what codebooks to use for decoding. Floor type 1 information needed to renter the decoded curve is stored in *mkFloor1Table*, which will actually perform the rendering. These are mainly the X support points for each floor curve. After every floor has been read *fsm_setup* makes a call to *mkFloor1Table.compute_neighbors(i)* passing the current floor number. This starts an FSM inside *mkFloor1Table* that computes the neighborhood funtion described in the Vorbis spec that is later needed for decode. In addition a sort index for the X support points is computed for each floor after the neighborhood funtion. This is done with a simple bubble sort algorithm, since the number of values usually gravitates around 20, and is at most 255 by definition. The hierarchie of the floor information stored and used can also be seen in figure 6.

After all floor information has been read from the setup header *fsm_setup* reads the number of residues used. For every residue it reads the residue type, the size of the residue, its partition size, classifications, cascades, and codebook numbers to be used. Classifications specifies the number of cascades, and each cascade specified in which passes a particular partition of the residue vector is updated. All this information is stored in *mkVorbisMemoryBank*. The hierarchie of the residue information stored and used can be seen in figure 7.

In the next step the number of mappings is read from the stream. For every mapping a type is read and since there is only one mappings type this type should always be 0. After the type the number of mapping submaps and the number of coupling steps is read, flowed by the magnitude residue and angle residue indexes for every coupling. Next muxing information is read that describes what submap is used by which audio channel. Finally the floor and residue numbers to be used with each submap are read from the data stream. All this information is stored in *mkVorbisMemoryBank*.

Finally the number of modes is read from the stream and for every mode we read the mapping and blocksize to be used in that mode, along with some other information that is unused in the current Vorbis version. Mode informatino is stored in *mkVorbisMemoryBank*. Figure 4 shows how mode and mapping information is stored and how it eventually effects floor and residue selection during the decode process.

After all the information contained in the setup header has been extracted *fsm_setup* reads a final framing bit that should always be 1. It then calls *mkOggReader.skip_segment()* and sets the decoders state to *VH_runinit*. This starts *fsm_cb* which uses the just decoded codebook information to generate the Huffman trees used during decode.

3.1.5. *Codebook initialization and Usage.* The structure and usage of codebooks is outlined in figure 5. The problem is to transform the codeword length based representation of the Huffman trees into a representation that is still memory efficient but allows for a quick decode time. Since there seems to be currently no linear time algorithm that allows decoding with just the length representation or a pure codeword representation a binary decode tree representation was chosen. However to save space every node in the tree contains only one 16 bit entry into the RAM table as oposed to a normal pointer based binary tree that needs a pointer to the left child and a pointer to the right child. The MSB is used to indicate if the node has children (0) or is a leaf and thus contains data (1). If the node contains data, the data is stored in the 15 MSB. That means that only codebooks with at most

$2^{15} = 32K$ words are supported, as opposed to the Vorbis standard that allows up to $2^24 = 16M$. However in practice the number of entries in a codebook seems to be below 650, so normal Ogg Vorbis streams should not exceed this limitation. If the node has children, then the left child is always stored at the next RAM address realtive to the current node, wheres the right child is stored at the current address + the 15 LSB + 1. Because the specificatin guarantees that the left children are always constructed first during decode this tree can be constructed in a very simple fashion by simply following nodes for one codeword after the other and every time an unused node is hit the tree is expanded. This can be easily done by keeping track of the first unused entry in RAM. To first transform the list of codeword lengths into a list of codewords the algorithm give in the *tremor* implementatin is used [6].

The Huffman trees for all codebook are constructed and stored sequentially in the same ram, with a small offset table at the start of the RAM that stores the beginning of each tree. Decode can now happen in linear time in the number of bits read from the data stream. A special decode FSM called *fsm_decode* is used every time a codeword needs to be read and decoded from the bitstream. That FSM loads the offset of the active codebook and reads the data stream bit by bit progressing through the tree until a data entry is found. For a n bit codeword, the FSM needs to make $n + 2$ memory accesses, thus running in linear time. Figure 5 gives an example of a small Huffman tree with 4 entires and what the corresponding table in the RAM would look like.

Once the huffman trees have been initialized *fsm_cb* sets the decoder state to *VH_audio*. This starts *fsm_audio* which decodes the actuall audio packets.

3.1.6. *Audio packet decode.* A highlevel overview of how data the data obtained from the setup header packet is used during audio packet decode is given in figure 4. The beginning of every audio packet is a single bit that needs to read 0. After that the mode number is stored with which the decoder is setup for this particular frame (see figure 4). Decode proceeds by readings the Y-values of the appropriate floors for each channel in channel order as described in figure 6. After that residues are decoded from the stream in submap order as shown in figure 7. After the floors and residues have been read from the audio packet there is no more data in the packet, except for the framing bit. To generate the current frame the decoded residues are inverse coupled according to the current mapping (see figure 4) and dot prodcuted with the floor curves. The spectrum generated for each channel is inverse transformed by the IMDCT module, windowed using the *mkVorbisTables* lookup module and overlapped with the previous frame by the *mkOutput* module. Currently the resulting data is simply printed as 4-digit hexadecimal values per sample using `$display`.

3.1.7. *Floating Point and Fixed Point Representation.* According to the Vorbis specification floating point values should be used. This decoder follows the example of the *Tremor* implementation and uses a mixed version of floating point and fixed point representation. When residues are decoded from the setup header packet, the are stored as vectors according to the dimension of the codebook. Each entry in this vector is stored in a 32 bit wide signed mantissa with a common floating point for the entire vector stores in an extra 8 bits value. For ease of implementation currently these 8 bits are attached to each mantissa to form a 40 bit wide RAM.

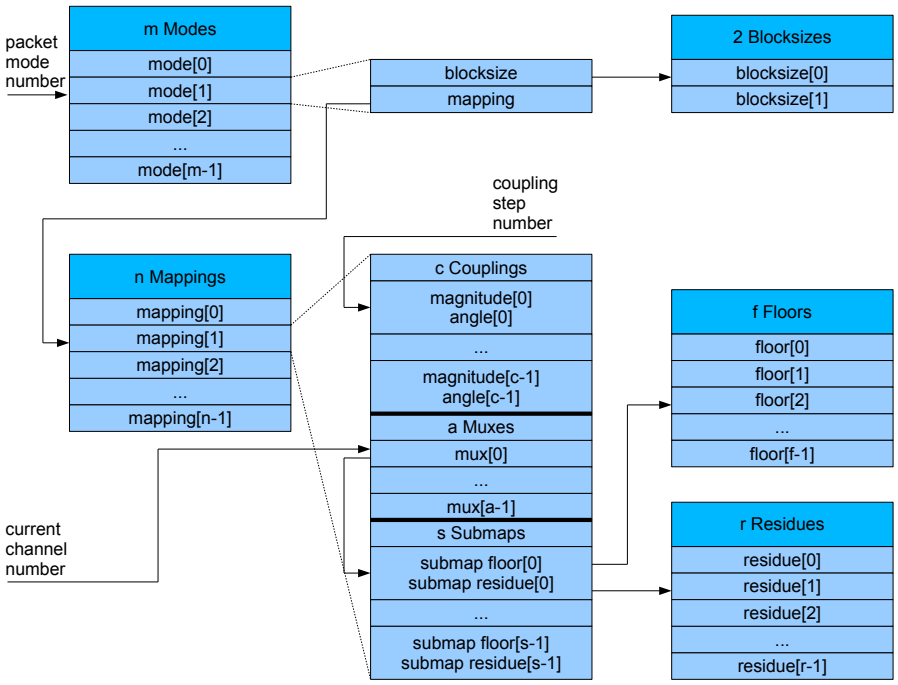

FIGURE 4. Floor and residue selection during audio packet decode. The numbers of modes m, mappings n, floors f, residues r, and audio channels a are fixed for the entire stream. The numbers of couplings c and submaps s can be different for every mapping. For every audio packet a packet mode is selected that dictates what blocksize and mapping to use. For every audio channel the mapping information is used to select the appropriate submap which dictates what floor and residue to use for this particular audio packet and channel. Once the floors and residues have been computed for all channels the current mapping is used to set the number of inverse coupling steps and select the magnitude residue and angle residue vectors for each inverse coupling step.

However all vectors are normalized to have the point at the same place for each entry.

When residues are genereated during audio packet decode, the entire residue vector is normalized to a 32 bit fixed point value with 8 bits before the point and 24 bits after the point. Floor curves are generated using integers values between 0 and 255 and a lookup table filled with the same 8/24 fixed point representation is used to generate the final floor curve during rendering. The IMDCT block also works with 8/24 fixed point values. The final output is generated by croping each value to ± 1 and cutting it down to 16 bits.

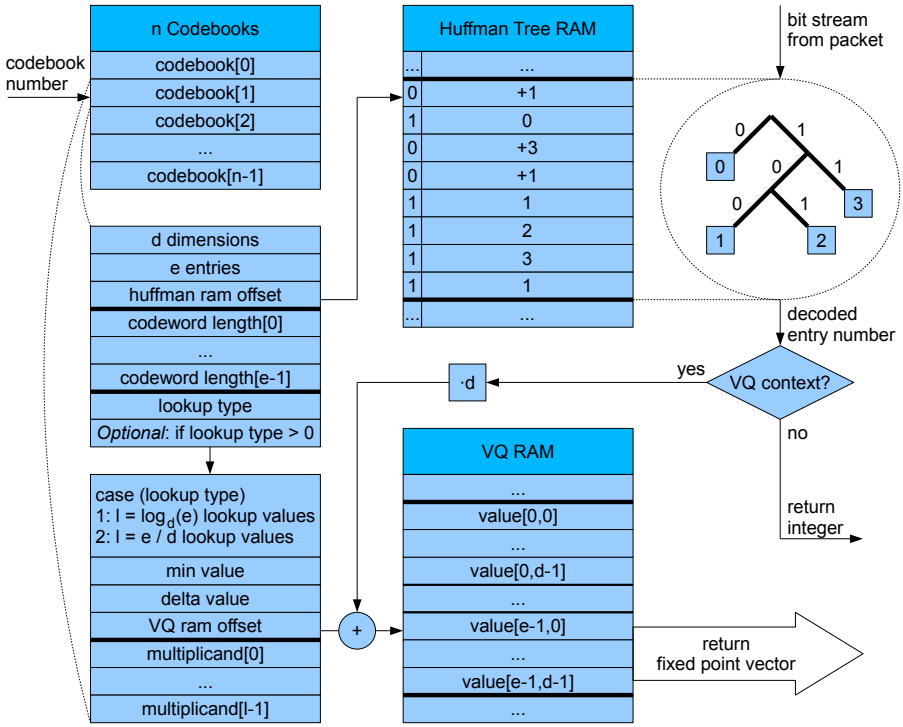

FIGURE 5. Codebook and VQ table structure. The number of codebooks n is fixed for the entire stream. The values codebook dimensions d and codebook entires e can be different for every codebook. The value lookup values l is computed based on d and e. Note that the entries codeword lengths, min value, delta value, and multiplicands are only used to compute the appropriate entires in the Huffman tree RAM and VQ RAM. When a particular codebook is selected it provides an offset into the Huffman tree RAM to the position of the decode tree. The tree is used to decode one codework from the incoming bitstream. If we are decoding in VQ context that entry is further used in conjunction with the VQ ram offset (codebook dependent) to return a fixed point value vector of size d.

3.2. **Debug Mode.** Most of the modules take a boolean parameter called debug when created. If debug is set to True the modules will print a lot of data as they decode streams and generate the floors, residues, spectra, and output vectors. If debug is set to True for *mkVorbisDecoder* then almost all debug information available is printed to the output stream. This is *A LOT* of information, but all of it is human readable. Most of this information is self explanatory, if not, please look inside the code to see what is being shown. When debug is disabled, only the PCM output vectors are being printed to the output stream.

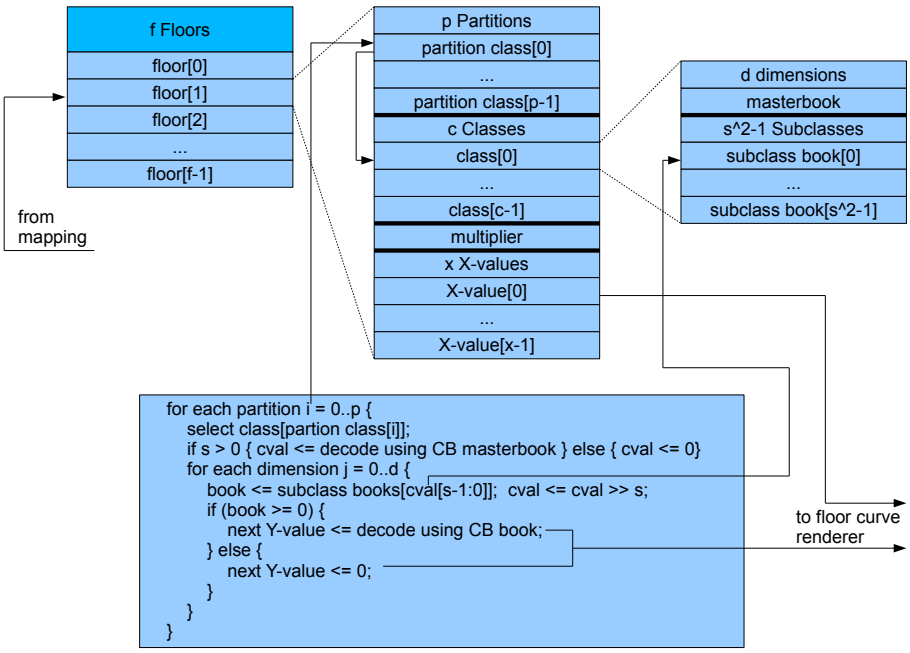

FIGURE 6. Floor curve generation. The number of floors f is the same for the entire stream. The number of partitions p, classes c, and X-values x can be different for different floors. The number of dimenions d and subclasses s (where there are $s^2 - 1$ subclasses) depends on the floor number and class number. The FSM given is a simplified part of *fsm_decode* responsible for extracting the Y support points from the data stream. The final floor curve is rendered using Bresenhams algorithm.

3.3. **Supported of the Ogg Vorbis Specification.** In its current state *mkVorbisDecoder* is capable of decoding all types of supported Vorbis I headers. This includes floor types 0 and 1, residue types 0, 1, and 2, and mapping and mode types 0. Types undefined in the Vorbis I spec currently issue a debug warning and are further ignored. In an FPGA version this should eventually cause a decoder interrupt / error condition. The audio decode side currently supports floor type 1 and residue types 0 and 1. Floor type 0 is not supported, since this type is not used anymore in current Vorbis encoders and since it requires a great deal more computation that floor type 1. Decoding residue type 2 is based on the residue type 1 decoder. It was not added since none of the test streams required it, however if it needed it should not require too much work to added suport for residue type 2 decoing. The decoder is written to support mutlichannel streams, however it was not tested in this configuration. Mutlichannel decoding only differs from single channel audio in that inverse coupling is used and multiple residues have to be decoded at once.

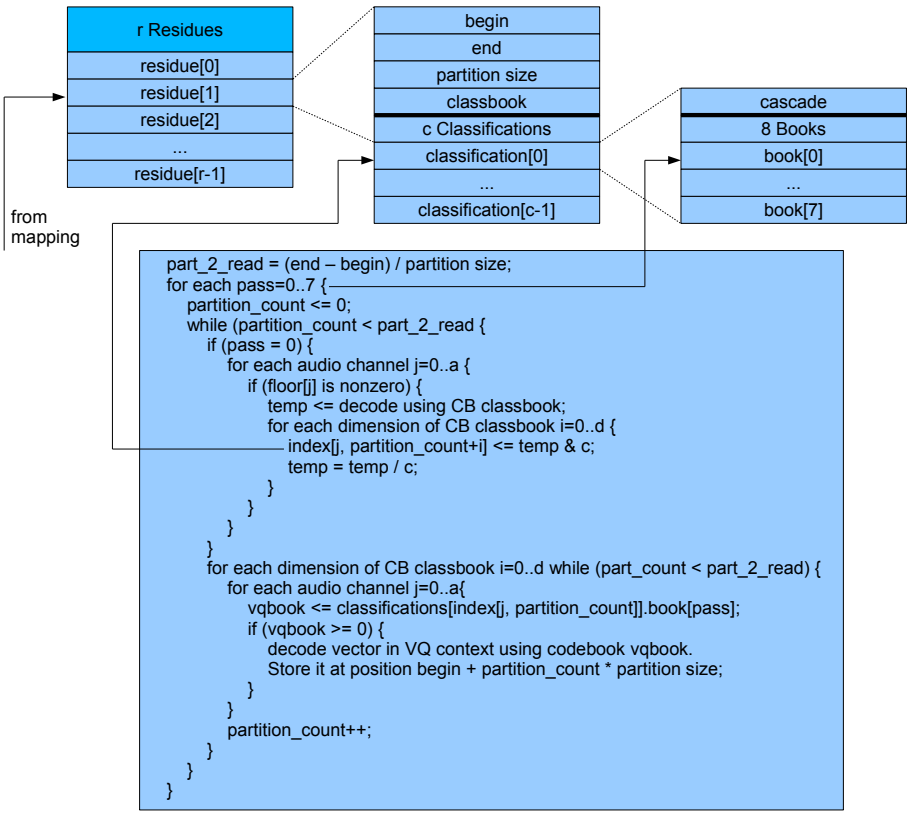

FIGURE 7. Residue vector generation. The number of residues r is the same for the enitre stream. The number of classifications c can be different for different residues. Every classification contains up to 8 codebook numbers for up to 8 passes where cascade is an 8 bit vector specifying what passes are active. The FSM given is a simplified part of *fsm_decode* responsible for residue vector decode.

It is important to note that the decoder very closesly follows the Vorbis specification and the Tremor implementation. This is especially the case for functions that have not been described closely in this manual, as they are often stating the specification verbatim, only in Bluespec.

3.4. **Possibilities for Parallelization.** Most of the above modules are stand alone enough to be instantiated multiple times in order to reduce computation time and allow for real time decoding of more audio channels should this be necessary. As mentioned above, clearly the IMDCT, which is the computationally most intensive module, can be instantiated multiple times to parallelize decoding. However also the floor and residue decoder can be parallelized. For the floor decoder, once the y-values for a floor curve have been read from the audio packet, that particular floor can be generated independently of the rest of the packet. Therefore multiple

floor decoder instantiations can be possibly used to speed up the decoder. Similarly for residue curves, a great amount of RAM that is currently used for lookup tables can be saved at the expense of additional divisions for every decoded partition of a residue vector (residue vectors are split into partitions, each of which is encoded using a single Huffman encoded codeword). However these partitions can be decoded independently of each other, an so multiple such decoders can be used in parallel to decode the residue vector. Each partition decoder requires a division circuit and a small FSM and thus relatively little area compared to the RAM used. This can easily save on the order of dozens to hundreds of kilobytes of RAM for lookup tables and increase overall throughput.

4. Testing Environment

The hierarchy of modules and their functionality for the testing environment is as follows:

(1) **mkTestBench** is the *toplevel module* in Bluespec simulation.
   (a) **mkMainMem** is the RAM that holds the Vorbis file to be decoded.
   (b) **mkCore** instantiates the decoder and connects it to the RAM passing though a cache.
      (i) **mkInstructionCache** is the cache used. It allows for request tagging. This is used to discard memory requests that happend before a reset of the decoder.
      (ii) **mkVorbisHeaderReader** is the decoder described in the sections above.

All test modules and all decoder modules are stored in the `src` directory. There is a make file located in `build` `build-bluesim` that will generate the simulator and run it on a list of test files specified inside the make file. The Ogg Vorbis test files are located in the `tests` directory. They are encoded from a 5 minute long sound sample consisting of five 1 minute long song samples. Each test file has been encoded from the same sound sample, but using different quality settings, ranging from -1 (lowest quality) to 10 (highest quality). To be input into the encoder the need to be stored in *vmh* format, since this is the format that Bluespec register files can be initialized with.

To simply generate the simulator type

```
make
```

This will generate the executable *a.out*. To run the simulator on all the test files specified in the make file type

```
make run-ogg-tests
```

For each *.vmh* testfile a *.out* output file will be generated.

In addition there are three C programs located in the `tools` directory that facilitate verification of the output. *wavegen* takes a filename as a first parameter and the number of samples to encode from it and generates a wave file that can be played in almost any media player. Notice that *wavegen* is written for mono wave files, encoded at 16 bits per samples and 22050 Hz, however this can be easily change inside the c code that is less than 100 lines long. *ogginfo* is also usefull, as it generates human readable information from all 3 Vorbis headers. Finally there is a modified version of *Tremor* located in

`tests`
`tremor` that will also take an output file as an argument and an ogg stream on *stdin*. This tremor implementation compares the PCM output it generates with the PCM values stored in the file that was passed to it. It outputs a single line for every PCM sample that contains the output tremor generated, the output from the *.out* file, and the difference between the two. On *stderr* it output only lines where the difference was larger than 1 LSB. By using a simple tool as *wc* on *sdterr* the total number of errors (count lines) can be easily computed.

5. Results and Discussion

The decoder has been tested with 4 different test files from the test directory with quality settings -1, 3, 6, and 10. The PCM output was verified using the modified *Tremor* implementation mentioned above. All errors were within 1 LSB, with most values decoded identical. The decoder was also hand verified at the pre-IMDCT stage, at which it is bitwise perfectly identical with *Tremor* pre-IMDCT output. The difference thereafter is to be attributed to the fact that the IMDCT implementation used follows the paper [10] very closely, while *Tremor* makes some optimizations by using sine and cosine lookup tables.

Performance was loosely measured by simulating 200 million cycles and measuring the amount of samples decoded. For lowest quality the decoder output more than 207 seconds of samples, thus it would be able to run at 1 MHz if compiled down to hardware. This is before many optimizations have been made, as the decoder currently is mainly working in serial mode, although many blocks could be parallelized with some additional work. In highest quality more than 100 seconds were decoded, so 2 Mhz would be sufficient for highest quality decoding.

In the end I have implemented a functional Ogg Vorbis Decoder that is almost entirely Bluespec based (all but the IMDCT) and can thus be complied into hardware. But as with many project (especially of this size) much work remains t be done. Multichannel decoding needs to tested and the likely hidden bugs for this functionality need to be fixed. Also there are many blocks that can be parallelized to achieve a significant speedup. Further optimizations can be done regarding memory usage. In summary there are still many memory optimizations to be made before this design will be able to fit into a medium sized-FPGA.

However despite all the work that remains I am very happy with the outcome of this project. A previous group of two people at MIT has tried to implement a Vorbis decoder in pure *Verilog* and was not able to do so within a single term [11]. The *University of Waterloo ASIC Design Team* is trying to implement a *VHDL* based Ogg Vorbis decoder in an FPGA in a time frame of many student-terms [12]. The project has been ongoing for a few terms now. In another hardware implementation called Ogg on a Chip [9] two students moved only the IMDCT onto and FPGA and left the rest of the decoder running on a publicly available LEON software processor inside the FPGA. Ogg on A Chip was implemented using VHDL, the code was written in two months by two people. This Bluespec implementation was written in only 1 month by a single person. However to be fair it has to be said that the current state of the decoder is in the simulation stage. But in spite of all future work I hope that the current implementation will prove useful to many others!

References

[1] *Bluespec ESL Design website*, http://www.bluespec.com/, 2008.
[2] *"The Xiph Open Source Community"*, http://www.xiph.org, 2008.
[3] *"Vorbis website"*, http://www.vorbis.com, 2008.
[4] *"Vorbis I specification"*, http://xiph.org/vorbis/doc/Vorbis_I_spec.html, 2008.
[5] *"Ogg Logical Bitstream Framing"*, http://xiph.org/vorbis/doc/framing.html, 2008.
[6] *"Tremor SVN repository"*, http://svn.xiph.org/trunk/Tremor/, 2008.
[7] *"Vorbis Hardware Wiki"*, http://wiki.xiph.org/VorbisHardware, 2008.
[8] *"VLSI Solution website"*, http://www.vlsi.fi/, 2008.
[9] *"Ogg-on-a-Chip Project website"*, http://oggonachip.sourceforge.net, 2008.
[10] T. Sporer, K. Brandenburg, and B. Edler, *"The use of multirate filter banks for coding of high quality digital audio"*, In Proceedings of the 6th European Signal Processing Conference, pages 211-214, 1992.
[11] J. Stritar and M. Papi, *"Ogg Vorbis Audio Decoder"*, http://web.mit.edu/6.111/www/f2005/projects/mpapi_Project_Final_Report.pdf, December 14, 2005.
[12] emphUniversity of Waterloo ASIC Design Team, http://www.asic.uwaterloo.ca/project/ogg.php, 2008.

office 32-376, MIT CSAIL
E-mail address: doniec@mit.edu