# Ogg Vorbis Audio Decoder

Jon Stritar and Matt Papi

6.111

December 14, 2005

**Abstract**

The goal of this project was to design and implement an Ogg Vorbis decoder in hardware. Ogg Vorbis is a highly dynamic audio encoding format - the framework was designed to be customizable enough to still be in use 20 years from now. The Vorbis decoding process was broken up into two parts; the front-end which configures the decoder, and the back-end which decodes the audio data into PCM. The front-end was completed successfully, but due to the unforeseen complexity of the audio decoding process, the back-end could not be debugged in time.

# Contents

**1 Introduction** **5**

**2 The Ogg Vorbis Audio Codec** **5**
2.1 The Front-end . . . . . 5
2.2 The Back-end . . . . . 6

**3 Front-end Modules** **6**
3.1 ROM Reader . . . . . 6
3.1.1 Overview . . . . . 6
3.1.2 Design . . . . . 6
3.1.3 Testing and Debugging . . . . . 7
3.2 Packet Filter . . . . . 7
3.2.1 Overview . . . . . 7
3.2.2 Design . . . . . 8
3.2.3 Testing and Debugging . . . . . 8
3.3 Identification Packet Processor . . . . . 9
3.3.1 Overview . . . . . 9
3.3.2 Design . . . . . 9
3.3.3 Testing and Debugging . . . . . 9
3.4 Configuration Module . . . . . 9
3.5 Setup Processor . . . . . 10
3.5.1 Overview . . . . . 10
3.5.2 Design . . . . . 10
3.5.3 Testing and Debugging . . . . . 11
3.6 Codebook Processor . . . . . 11
3.6.1 Overview . . . . . 11
3.6.2 Design . . . . . 12
3.6.3 Testing and Debugging . . . . . 14
3.7 Transform Processor . . . . . 15
3.7.1 Overview . . . . . 15
3.7.2 Testing and Debugging . . . . . 15
3.8 Floor Processor . . . . . 15
3.8.1 Overview . . . . . 15
3.8.2 Design . . . . . 15
3.8.3 Testing and Debugging . . . . . 16
3.9 Residue Processor . . . . . 17
3.9.1 Overview . . . . . 17
3.9.2 Design . . . . . 17
3.9.3 Testing and Debugging . . . . . 18
3.10 Mapping Processor . . . . . 18
3.10.1 Overview . . . . . 18

3.10.2 Design . . . 19
3.10.3 Testing and Debugging . . . 19
3.11 Mode Processor . . . 19
3.11.1 Overview . . . 19
3.11.2 Design . . . 19
3.11.3 Testing and Debugging . . . 19
3.12 Test Modules . . . 19
3.12.1 Fake Minor FSM . . . 19
3.12.2 Fake Major FSM . . . 20
3.12.3 Test RAM . . . 20

**4 Back-end Modules** **20**
4.1 Standardization . . . 20
4.1.1 The Serial Interface . . . 20
4.1.2 The Control Interface . . . 21
4.1.3 The Memory Interface . . . 21
4.2 Why not stereo? . . . 21
4.3 Audio Packet Processor . . . 22
4.3.1 Overview . . . 22
4.3.2 Mechanics . . . 22
4.3.3 Dataflow . . . 24
4.3.4 Testing and Debugging . . . 24
4.4 Floor Decoder . . . 25
4.4.1 Overview . . . 25
4.4.2 Mechanics . . . 25
4.4.3 Dataflow . . . 27
4.4.4 Testing and Debugging . . . 28
4.5 Residue Decoder . . . 28
4.5.1 Overview . . . 28
4.5.2 Mechanics . . . 29
4.5.3 Dataflow . . . 30
4.5.4 Testing and Debugging . . . 30
4.6 Window Select/Look-up . . . 30
4.6.1 Overview . . . 30
4.6.2 Mechanics . . . 31
4.6.3 Dataflow . . . 31
4.6.4 Testing and Debugging . . . 32
4.7 Inverse Modified Discrete Cosine Transform . . . 32
4.7.1 Overview . . . 32
4.7.2 Mechanics . . . 32
4.7.3 Dataflow . . . 33
4.7.4 Testing and Debugging . . . 33
4.8 Dot Product . . . 33

4.8.1 Overview . . . . . . . . . . 33
4.8.2 Mechanics . . . . . . . . . . 34
4.8.3 Dataflow . . . . . . . . . . 34
4.8.4 Testing and Debugging . . . . . . . . . . 34

**5 Back-end Helper Modules 34**
5.1 Huffman Decoder . . . . . . . . . . 34
5.1.1 Overview . . . . . . . . . . 34
5.1.2 Testing and Debugging . . . . . . . . . . 36
5.2 Bitstream Reader . . . . . . . . . . 36
5.2.1 Overview . . . . . . . . . . 36
5.2.2 Testing and Debugging . . . . . . . . . . 36
5.3 Trig Look-up . . . . . . . . . . 36
5.3.1 Overview . . . . . . . . . . 36
5.3.2 Testing and Debugging . . . . . . . . . . 37
5.4 Sort . . . . . . . . . . 37
5.4.1 Overview . . . . . . . . . . 37
5.4.2 Testing and Debugging . . . . . . . . . . 37
5.5 Neighbor Search . . . . . . . . . . 38
5.5.1 Overview . . . . . . . . . . 38
5.5.2 Testing and Debugging . . . . . . . . . . 38
5.6 Line Renderer . . . . . . . . . . 38
5.6.1 Overview . . . . . . . . . . 38
5.6.2 Testing and Debugging . . . . . . . . . . 38
5.7 Point Renderer . . . . . . . . . . 39
5.7.1 Overview . . . . . . . . . . 39
5.7.2 Testing and Debugging . . . . . . . . . . 39

**6 The Mini-Decoder 39**
6.1 Testing and Debugging . . . . . . . . . . 40

**7 Analysis 40**
7.1 What Went Right . . . . . . . . . . 40
7.2 What Went Wrong . . . . . . . . . . 41

**8 Conclusion 41**

**A The Huffman Decoding Algorithm 42**

# List of Figures

1 Front-end Block Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2 Setup Processor State Transition Diagram . . . . . . . . . . . . . . . . . . . . . . . 10
3 An example timing for the serial interface. . . . . . . . . . . . . . . . . . . . . . . 20
4 An example timing for the control interface. . . . . . . . . . . . . . . . . . . . . . 21
5 Arrow notation as used in back-end diagrams. . . . . . . . . . . . . . . . . . . . . 22
6 The floor decoder and its helper modules. . . . . . . . . . . . . . . . . . . . . . . 26
7 The IMDCT module and its helper modules. . . . . . . . . . . . . . . . . . . . . . 32
8 The Huffman decoder state transition diagram. . . . . . . . . . . . . . . . . . . . 35

# List of Tables

1 Floor RAM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2 Residue Vector RAM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3 Residue Constants RAM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

# 1 Introduction

Ogg Vorbis is a lossy audio codec that is similar in principle to formats like MP3. Ogg Vorbis has recently been receiving a great deal of attention for a number of reasons, including the fact that it is free and open, and that it tends to provide a smaller file size for similar quality encodings compared to other lossy codecs. Currently, a considerable number of software audio players support the Vorbis format. Unfortunately, there few hardware players that support Vorbis, and those that do tend to be of the larger, more expensive, and more power-hungry variety.

The project itself is divided into two major parts. The first of these is the front-end, which is responsible for decoding the Vorbis header packets that contain a wealth of information for configuring the decoder and identifying the Vorbis stream. Vorbis, unlike most other audio codecs, packs all information needed to decode a stream into the stream itself, which gives it the freedom to optimize numerous aspects of the encoding. The front-end is an interesting system: for a five-minute audio sample, the front-end will run for but a fraction of a second, while the decoder back-end runs for five minutes. Yet without the processing done by the front-end, the decoder would be completely unable to even begin decoding the audio stream.

The second part of the decoder is the back-end, which is responsible for approximating the original audio waveform from a bitstream of data and the information provided by the front-end. The back-end performs a number of complex computations, involving finite state machines with tens of states, parallel computations, searching, sorting, Huffman decoding, and line rendering, to name a few. One of Vorbis's strengths is higher-than-normal quality at low bitrates, and this relates directly to the complexity of the back-end.

# 2 The Ogg Vorbis Audio Codec

## 2.1 The Front-end

The Vorbis stream is broken up into packets of data. There are four different types of packets. The first three comprise the header — the identification, comments, and setup packets. There is one of each header packet. The remaining packets in the stream are all audio data. The identification packet identifies the data stream as valid Vorbis and contains information about the encoded data. The comments packet contains text fields such as the title, artist and album of the song. The setup packet sets the decoder up to be able to decode the audio data.

The front-end configures the Vorbis decoder by extracting the various settings from the header packets. These settings include the sample rate, number of audio channels, the maximum, minimum and typical bit rates, and the codebooks.

The codebooks make up a majority of the setup packet. Each Vorbis stream contains an arbitrary number of codebooks that are used to decode the audio packets. The codebooks represent the probability model used to encode the audio data. Each codebook is made up of two parts - the Huffman coding and the Vector Quantization (VQ). A Huffman coding is a

form of encoding used for lossless compression. The codewords are essentially variable length bit sequences which directly map to source sequences. By representing source sequences which appear more often with shorter bit sequences, Huffman codings reduce the size of the bit sequence. VQ is a way of representing the data with a small set of points — that is, it quantizes the data. When the data becomes quantized, there are more repeats meaning the Huffman representation takes up even less space.

## 2.2 The Back-end

Essentially, audio data in a Vorbis stream consists of frequency domain data that has been separated into two separate components: the "floor", or base energy curve of a signal, and the "residue", the data that remains from subtracting out the floor from the signal. The floor data is packed in a Huffman-encoded interleaved form. The floor data itself does not represent the entire floor itself, but instead contains just enough information to reconstruct the floor (endpoints of lines, for example). The residue data is quantized and Huffman-encoded. In the cases of both the floor and residue, the data is filled with a mix of additional parameters and flags that guide the decode process.

# 3 Front-end Modules

The front-end initiates the bitstream and configures the Vorbis decoder with data from the header packets. See Figure 1 for the block diagram of the front-end. Also, refer to Figure 3 for details on the bitstream handshaking.

## 3.1 ROM Reader

### 3.1.1 Overview

The ROM reader is responsible for initializing the bitstream. The module converts the 32 bit wide data coming from the ROM into the more easily handled bitstream. The ROM reader activates the packet filter, starting the whole decode process.

### 3.1.2 Design

The ROM reader is a two state finite state machine (FSM), consisting of an active and inactive state. When activated, the ROM reader maintains a buffer consisting of the current 32 bits of data that the bitstream is reading from. When the module sends the last bit in the buffer, it replaces the buffer with the data currently on the ROM data bus. The ROM reader then increments the address so that the data line then contains the next 32 bits of data.

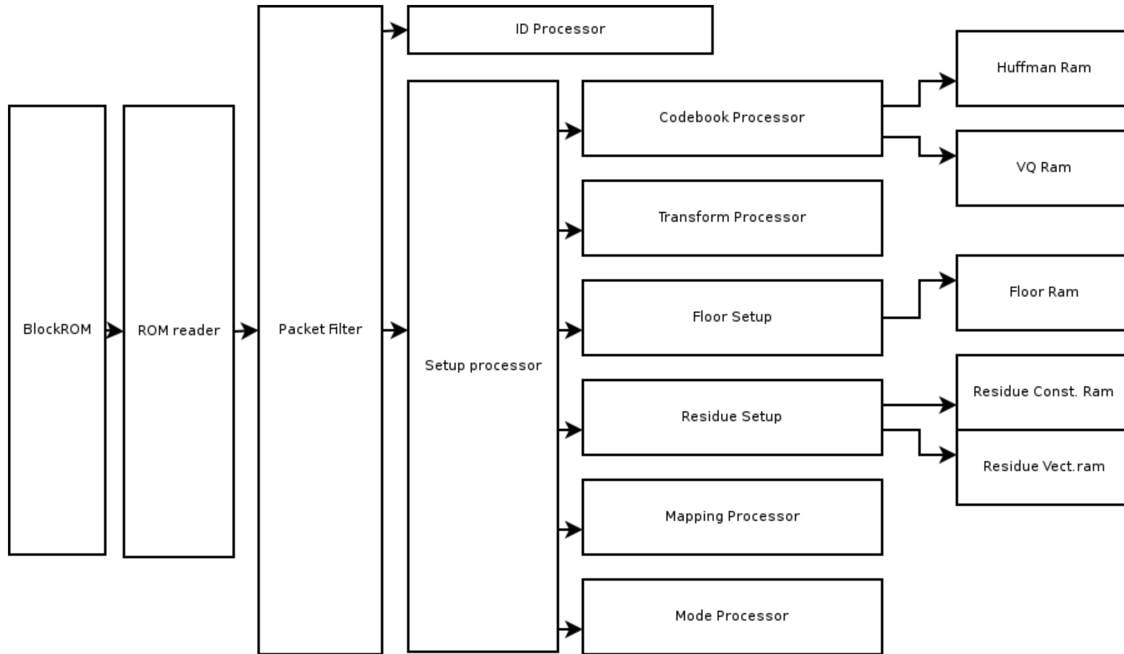


Figure 1: Front-end Block Diagram

### 3.1.3 Testing and Debugging

Since the ROM reader starts the data stream, it is important that it does not skip any bits or addresses. The ROM reader was run under simulation with various fake minor FSMs (Section 3.12.1). The bit count was then checked to make sure it incremented correctly. The ROM reader was also tested on the lab kit with the fake minor FSMs. The bit count was displayed on the hex display, making it easy to verify if it stopped at the correct location.

One problem that initially occurred was that the buffer was not being initialized when the ROM reader first started up, thus causing the first 32 bits of the stream to be incorrect. This problem was easily fixed by adding initial values to the address registers and making sure to set the buffer on activation.

## 3.2 Packet Filter

### 3.2.1 Overview

The packet filter is responsible for identifying the type of packet and verifying bitstream sync. The module is a finite state machine (FSM) that coordinates the activities of the various packet processors. When the ROM reader activates the packet filter, indicating that packets are available, the packet filter begins reading from the bitstream. The module identifies what packet type it is and then activates the processor associated with that one. The packet filter serves as one of the links between the front and back-end by diverting the bitstream to the back-end when processing audio data packets.

### 3.2.2 Design

The packet filter is a seven state FSM.

- `STATE_IDLE`: The packet filter begins in `STATE_IDLE`. While in this state the module outputs a low busy signal, indicating that it is ready to receive a packet. If the ROM reader asserts the start signal, then the packet filter transitions to `STATE_READ_HEADER`.
- `STATE_READ_HEADER`: While in `STATE_READ_HEADER` the packet identifies the packet type and verifies a packet sync sequence. The module reads in a total of 56 bits. The first 8 bits are read as an unsigned integer. The following 48 bits are read and compared to the sync sequence "0x76, 0x6f, 0x72, 0x62, 0x69, 0x73". If the sync fails, then the FSM enters and remains in `STATE_ERROR`. Otherwise, the FSM transitions depending on the packet type. A packet type of 1 indicates that it is the identification packet, and the FSM transitions to `STATE_PACKET_ID`. If the packet type is 3, then the packet filter enters state `STATE_PACKET_COMMENTS`. If the packet type is 5, then the packet is the setup packet and the FSM enters `STATE_PACKET_SETUP`. The packet filter considers all other packets audio data, and thus transitions to `STATE_PACKET_AUDIO`.
- `STATE_PACKET_ID`, `STATE_PACKET_COMMENTS`, `STATE_PACKET_SETUP`, `STATE_PACKET_AUDIO`: These states indicate that the packet type has been identified and that the bitstream is currently being processed by either the id processor (Section 3.3), comments processor, setup processor (Section 3.5), or audio packet processor (Section 4.3). While in any of these states, the bitstream control lines coming from the ROM reader are connected to the control lines of the respective minor FSM through combinational logic. The bitstream has an intrinsic clock cycle delay because registers change value after the positive edge of the clock. By using combinational logic to guide the stream, the delay remains constant when adding modules in the path. When the activated packet processor deasserts its busy signal, the bitstream has reached the end of the packet and the FSM transitions back to `STATE_IDLE`.
- `STATE_ERROR`: The packet filter enters the error state when it cannot verify the packet sync sequence. Once in the error state, the error output is constantly high and the FSM will not exit the state until reset.

### 3.2.3 Testing and Debugging

After completion, the packet filter was tested in simulation using the stub major and minor FSM modules. The major FSM implemented the bit supply side of the bitstream, and the minor FSMs implemented the bit demand side of the bitstream. The major FSM contained fake packets, and the minor FSMs were instructed to read a particular number of bits once activated. The packet filter performed correctly, identifying which packet was present, verifying the sync sequence, activating the packet's processor module, and guiding the bitstream to the minor FSM.

## 3.3 Identification Packet Processor

### 3.3.1 Overview

The identification (id) processor is responsible for parsing important configuration data from the id packet. The processor writes the data to a configuration module (Section 3.4), making it available to the back-end decoder. The processor extracts information including the sample rate, number of audio channels, and the maximum, minimum and typical bit rates.

### 3.3.2 Design

The id processor was designed using an FSM. Each state of the FSM represented a different piece of data to extract from the bitstream[1]. The id processor has a fairly primitive implementation of the bitstream model. Each state of the FSM requests bits and relies on the bits to arrive two cycles later.

### 3.3.3 Testing and Debugging

The id processor was first tested in simulation using a stub major FSM that provided the contents of an id packet. The testing framework verified that each variable was extracted from the stream correctly by displaying the data and address signals. Since the id processor also has to maintain bitstream synchronization, the number of bits the stub FSM fed the processor was measured.

The testing of the id processor made it apparent that maintaining bitstream sync between the states of one FSM and, as importantly, between various FSMs would be a critical task of the front-end. It became clear that the method employed by the id processor was not optimal. As a result, a function state designed to read bits was introduced for the next FSMs. By isolating the bitstream activity to one state, then only that state needed to be debugged if the bitstream was out of sync. If that state worked, and each state invoked the function state in the correct manner, then the module would maintain synchronization.

## 3.4 Configuration Module

The configuration module was initially designed to store many of the constant values needed by the back-end for decoding. The module stores constants in a register array and implements a RAM style interface with write enable, address and data ports. The configuration module was dropped from the final design in favor of storing essential settings directly in the back-end modules. The module was useful in testing the id processor and served as the basis for the test ram module (Section 3.12.3).

[1] See Vorbis I Specification, Section 4.2.2 for details on decoding the identification header.

## 3.5 Setup Processor

### 3.5.1 Overview

The setup processor guides the unpacking and parsing of the setup packet. The module serves a similar role as the packet filter discussed in Section 3.2 — it does not actually process any data, but instead coordinates the activities of minor FSMs. The minor FSMs that process the setup packet make up a majority of the front-end. The setup processor facilitates their actions by diverting the bitstream to the destination module and by activating the modules in sequence.

### 3.5.2 Design

The setup processor is a seven state FSM, as shown in Figure 2.

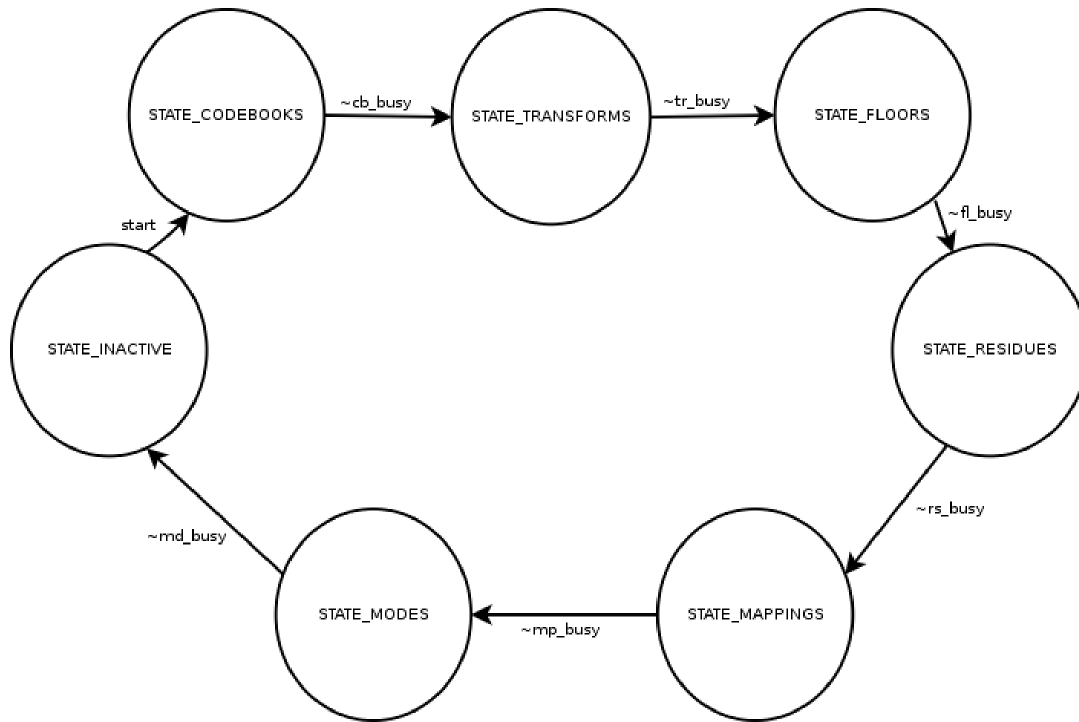


Figure 2: Setup Processor State Transition Diagram

- `STATE_INACTIVE`: The setup processor remains in the `STATE_INACTIVE` state while idle. The module outputs a constant low busy signal, indicating that it is ready to accept a packet. When the packet filter asserts the start signal, the setup processor replies with the busy signal, applies the codebook processor's start signal and transitions to `STATE_CODEBOOKS`.
- `STATE_CODEBOOKS`, `STATE_TRANSFORMS`, `STATE_FLOORS`, `STATE_RESIDUES`, `STATE_MAPPINGS`, `STATE_MODES`: While in each of these states, one of the setup processor's minor FSMs is

activated and processing data. The bitstream control lines of the FSMs are connected through combinational logic to the bitstream lines of the packet filter, thus keeping the clock cycle delay constant by avoiding sequential logic. When transitioning into each of these states, the corresponding start signal is raised for one clock cycle. The minor FSM responds by maintaining an active high on its busy signal until it has completed processing the bitstream. Each minor FSM is responsible for maintaining synchronization in the bitstream when activated. When the busy signal for the active FSM goes low, the setup processor transitions to the next state, as shown in Figure 2.

### 3.5.3 Testing and Debugging

The setup processor was first tested in simulation. The fake major FSM (Section 3.12.2) was used to supply the bitstream, and six instances of the fake minor FSM (Section 3.12.1) were used to verify the module's state transitions and bitstream handling.

Most of the issues revealed during simulation were related to the timings of the busy and start signals, and the bit requests of the minor FSMs. Since there is a clock cycle delay before the busy signals of the minor FSMs go high, each state needs to detect the falling edge of the signal - not simply a low busy. The testing also revealed a flaw in the fake minor FSM. The module lowered its busy signal after requesting the designated number of bits (defined using defparam), rather than after receiving that number. This caused the bitstream to go out of sync since the setup processor would transition and divert the previous FSM's bits to the next module.

## 3.6 Codebook Processor

### 3.6.1 Overview

The codebooks are packed in the first section of the setup header and comprise a majority of the entire packet. The codebooks, as described in Section 2.1, contain the Huffman codings and vector quantizations (VQ).

The codebook processor is the most complicated module on the front-end. The module extracts the Huffman lengths, computes the decoding sums (Section A), determines the VQ lookup type and then unpacks the VQ based on the lookup type[2]. The codebooks can be packed in an ordered or non-ordered manner. If they are ordered, then the number of entries for a given length is read, otherwise the length for a given entry is read from the stream. No Ogg files seem to use the ordered method, so the codebook processor only supports non-ordered codebook lookups.

There are also three different VQ lookup types - "0" for no lookup, "1" for a lattice VQ lookup table, and "2" for a VQ lookup table built from scalar values. Lookup types 0 and 1 were used in all the Ogg files inspected, so the codebook module ignored lookup type 2. The module unpacks the Huffman codings into a format usable by the floor decoder (Section 4.4), and the VQs into a format usable by the residue decoder (Section 4.5).

[2]Refer to Vorbis I Specification, Section 3 for algorithmic details.

### 3.6.2 Design

The codebook processor was implemented using a 16 state finite state machine. By abstracting most of the functions into their own states, the design was simplified down to tractable portions. The function states are essentially used as subroutine calls.

The codebook processor was designed with some limitations and assumptions in place. The theoretical limit on the number of codebooks is $2^8$ codebooks of $2^{24} - 1$ entries and $2^{16} - 1$ dimensions of VQ — something which we couldn't possibly hold. A Python script revealed that the typical Ogg Vorbis audio file has practical bounds of about 50 codebooks of 650 entries and 4 dimensions. The Huffman codings are stored in their own 42-bit wide RAM. The left 10 bits are the entry number, the next 6 are the length, and the right 26 are the decoder sums. As a result, the codebooks are packed in the RAM one after another, with the start addresses written to the beginning of the RAM.

The number of elements in each VQ vector is equal to the codebook dimension, and that number generally does not rise above 4. Since the residue decoder (Section 4.5) needs to read all four at once, the complete vector for a given offset is stored at each address. The width of the VQ ram is 48 bits, allocating 12 for each element of the vector. The VQ codebooks are packed in a similar fashion to the Huffman - the start addresses are recorded at the beginning of the RAM.

This final specification for the RAM was not completely implemented — the Huffman output is only 32 bits and does not have the entry number in it. It was not completed because the change in the specification was made fairly late into the project, and it seemed wiser to spend time on other parts of the project since the floor decoder was not going to be completed. It should be a relatively simple fix since the module only writes to the RAM from one state.

- `STATE_INACTIVE`: The module is outputting a low busy signal and is ready to start processing codebooks. When the start signal is applied, the FSM transitions to `STATE_CODEBOOK_COUNT`.

- `STATE_CODEBOOK_COUNT`: The codebook processor moves to `FN_STATE_READ_BITS` to read the number of codebooks present in the header. Once complete, the FSM transitions to `STATE_CODEBOOK_START`.

- `STATE_CODEBOOK_START`: If the module has unpacked all the codebooks, then the module has completed processing of the bitstream and transitions back to `STATE_INACTIVE`. Otherwise, the FSM reads the codebook sync sequence, and the number of entries and dimensions in the codebook. These are read in using `FN_STATE_READ_BITS` and are written to the ram using `FN_STATE_WRITE_RAM`. If the codebook sync is correct, the FSM begins to read the sparse flag in `STATE_NORDERED_START`, otherwise it enters `STATE_ERROR`.

- `STATE_NORDERED_START`: One bit is read in this state to check if the codebook is sparse. When sparse, a flag preceeds every entry length indicating whether or not to read

the entry length or infer that it is 0. After reading the flag, the module enters the `STATE_NORDERED_CHECK_FLAG` state.

- `STATE_NORDERED_CHECK_FLAG`, `STATE_NORDERED_READ_ENTRY`: The FSM cycles through `STATE_NORDERED_CHECK_FLAG` once for each entry in the codebook. If it is sparse, a one bit flag is read, and, if set, a length is read in via `STATE_NORDERED_READ_ENTRY`. If it is not sparse, then a length is always read in through the `STATE_NORDERED_READ_ENTRY` state. After the length for a given entry is determined, it is written to the ram using `FN_STATE_WRITE_RAM`. Once complete, the fsm transitions to `STATE_CALC_HUFF_SUM` in order to calculate the sums.

- `STATE_CALC_HUFF_SUM`: After all the lengths have been unpacked, the `STATE_CALC_HUFF_SUM` state iterates through each length stored in the RAM, reading them with the `FN_STATE_READ_RAM` state. The module calculates $2^{(max_length+current_length)}$ for each, and adds it to the previous. The module then wries this sum in the right 26 bits of the Huffman ram, with the length in the left 6 bits. Once complete, the Huffman codings are complete and the FSM enters `STATE_VQ_LOOKUP_TYPE`.

- `STATE_VQ_LOOKUP_TYPE`: The codebook processor reads the VQ lookup type in this state. If the type is 0, then there is no VQ table and the FSM can transition back to `STATE_CODEBOOK_START` to start the next codebook. Otherwise, it transitions to `STATE_VQ_1_START` to begin unpacking the VQ table.

- `STATE_VQ_1_START`: In `STATE_VQ_1_START`, the FSM reads variables that are used to generate the vectors in the VQ table. Two of the values are unpacked using the `float32_unpack` function defined in the Vorbis specification. The codebook processor converts these values into unsigned integers since those were the only ones encoded in the stream. The FSM also reads in the value for $value_bits$ which is used to extract the VQ multiplicands in `STATE_VQ_MULTIPLICANDS`. Once complete, the FSM enters the function state `FN_STATE_LOOKUP1_VALUES`. After the function call is complete, the FSM transitions to `STATE_VQ_MULTIPLICANDS` to read in the multiplicands used to calculate the VQ.

- `FN_STATE_LOOKUP1_VALUES`: This state is responsible for determining how many VQ multiplicands must be read from the bitstream when the lookup type is 1. The function is defined as $lookup_values(entries, dimensions) = return_value$ where $return_value$ equals the greatest integer such that $return_value^{dimensions} \leq entries$. The value is computed by checking each integer starting from 1. Once the correct value is determined, the FSM transitions to the state contained in register `lv_state` with the return value in `lv_ans`.

- `STATE_VQ_MULTIPLICANDS`: While in `STATE_VQ_MULTIPLICANDS`, the FSM reads $lookup_values$ multiplicands of length $value_bits$. The codebook processor is finally ready to calculate the actual vectors in the VQ table and transitions to `STATE_VQ_CALCULATE_VALUES`.

- `STATE_VQ_CALCULATE_VALUES`: In `STATE_VQ_CALCULATE_VALUES`, the VQ vectors are calculated algorithmically from the multiplicands read in `STATE_VQ_MULTIPLICANDS` and the values unpacked in `STATE_VQ_1_START`. As the vectors are calculated, they are written to the VQ RAM to be used by the back-end. After all have been calculated, the codebook is complete and the processor cycles to `STATE_CODEBOOK_START` to work on the next cycle.

- `FN_STATE_READ_BITS`: This state is used to read bits from the bitstream. The calling state fills the `br_bits` register with the number of bits to be read, and the `br_state` with the state to return to. The calling state need not worry about reseting any counters or buffers. Then, the `FN_STATE_READ_BITS` state fills the `br_buffer` register with the bits received, transitions back to `br_state` and asserts the `br_done` signal for one cycle.

- `FN_STATE_WRITE_RAM`: This state is used to write data to the RAM. The calling state fills the `ram_selector`, `ram_addr`, `ram_data` registers with the corresponding data. The selector is used to select between constants, Huffman and VQ, then the address represents the entry to write to. By keeping this interface generic enough, it is simple to change memory representations - only the `FN_STATE_WRITE_RAM` state needs to change. After complete, the FSM transitions back to the state held in `ram_state` and asserts the `ram_done` signal for one clock cycle.

### 3.6.3 Testing and Debugging

The codebook processor was a fairly large module to debug. Once again, the module was supplied a bitstream of known data in simulation. A test RAM module, consisting of a large array of registers to simulate a block RAM, was used during testing. While this works in simulation, it was not feasible for actual implementation since large arrays of registers seem to take hours to synthesize.

The codebook processor was fairly difficult to debug. When there are so many states in an FSM, it is easy to lose track of the connections between the states. The code becomes separated enough that it becomes more difficult to reason about as a whole. The use of function states made the job significantly easier.

The function states isolated the more complex operations that were commonly performed. The timing associated with reading and writing to the RAM only needed to be worked out in one state. Also, by abstracting the writing to the RAM, it is very easy to change the storage schematic. The calls to the write state are generic enough only the contents of the write state would have to be changed. The read state would also only need to be changed. At one point we did change the memory scheme, and the subroutine calls minimized the amount of work required by the change.

## 3.7 Transform Processor

### 3.7.1 Overview

The transform processor is responsible for reading the time domain transforms from the setup header. Vorbis I does not contain support for dynamically encoded transforms, so this processor is simply a placeholder - reading the designated number of bits[3]. This module is necessary in order to maintain bitstream synchronization.

### 3.7.2 Testing and Debugging

The transform processor was verified to read the correct number of bits in simulation. If any transform has a nonzero type, then it transitions to `STATE_ERROR` and outputs an active high on the error port. Since the module does not write to any RAMs, no other outputs needed to be verified. The module fairly simple, so it was not tested in isolation on the lab kit, only once integrated with the setup processor.

## 3.8 Floor Processor

### 3.8.1 Overview

The floors are the third pieces of data located in the setup packet. The floor processor connected directly to the setup processor. The module extracted various vectors from the setup header which the floor decoder (Section 4.4) used to reconstruct the floor curve.

### 3.8.2 Design

The floor processor generally extracts a variable number of scalar values into a vector in multiple different places. The module also constructs a two dimensional vector from the bitstream. This presented an interesting challenge — storing the data in a way that was both simple to write to and read from, but also space efficient. After dissecting many typical Ogg Vorbis files, it appeared that there was usually no more than two floors. Each of the vectors seemed to be bounded relatively low, so we decided to store all the vectors in the same block RAM at fixed offsets. It seemed that little space could be saved by using a more complex vector packing scheme. The benefit did not outweight the increased difficultly in interfacing to a more complex scheme.

The memory scheme is described in Table 1. The module does not implement this scheme completely since it is a relatively new specification, and the floor decoder was not going to be done in time to use it. The changes would be fairly simple and would need to be done to the `FN_STATE_WRITE_RAM` state.

The floor processor was implemented using a 13 state FSM, where each state performed a different part of the algorithm.

[3]See the Vorbis I Specification, Section 4.2.4.2. The number of time domain transforms are read, then the type of each is read and verified to be "0".

Table 1: Floor RAM

| Description | Width | Length | Element Address |
|---|---|---|---|
| constants | 7 | 4 | — |
| floor1_partition_class_list | 4 | 10 | [i] = i + 4 |
| floor1_class_dimensions | 3 | 5 | [i] = i + 13 |
| floor1_class_masterbooks | 8 | 5 | [i] = i + 18 |
| floor1_class_subclass_books | 8 | 5 | [i],[j] = (i ≪ 3) + j + 23 |
| floor1_X_list | 8 | 30 | [i] = i + 63 |
| floor1_class_subclasses | 2 | 5 | [i] = i + 93 |

- `STATE_INACTIVE`: In this state the FSM is ready for new data and transitions to `STATE_COUNT` when the start signal is asserted.
- `STATE_COUNT`: In this state, the module reads the number of floors from the bitstream. Decodes each floor by cycling through the subsequent states.
- `STATE_FLOOR_START`: In `STATE_FLOOR_START`, the FSM either reads scalar values which determine how many iterations it will perform in the next state, `STATE_CLASS_LIST`, or it enters the inactive state if it just completed the last floor.
- `STATE_CLASS_LIST`: In this state, the floor processor fills a vector by iterating over the value read in the previous state. The module writes the vector to the floor RAM module via `FN_STATE_WRITE_RAM`.
- `STATE_CLASS`, `STATE_MASTERBOOK`, `STATE_SUB_CLASSES`: The FSM oscillates between these states as it fills three different vectors interleaved in the bitstream. On a given iteration, if the value entered in `STATE_SUB_CLASSES` is non-zero, then an additional iteration occurs in `STATE_MASTERBOOK` to enter values into one of the two dimensional arrays[4].
- `STATE_X_INFO`, `STATE_X_VALUES`: These states are used by the floor processor to unpack the list of $x$ values for which the floor applies to. These values are then used by the floor decoder to reconstruct the actual floor spectrum curve.

### 3.8.3 Testing and Debugging

The testing of the floor processor proceeded very similarly to the other minor FSMs of the setup processor. It was very important to verify that the processor changed states correctly when filling the three interleaved vectors - an incorrect transition could render the stream

[4]See Vorbis I Specification, Section 7.2.2.

undecodeable. It was also crucial to verify that the vector elements were being written to the right addresses.

The floor module was tested alone on the lab kit with the fake major FSM supplying data. The outputs were verified on the logic analyzer. The module was also tested in conjunction with the setup processor and the rest of the front-end.

## 3.9 Residue Processor

### 3.9.1 Overview

The residues are located after the floors in the setup header. The residue information located in the setup packet is used by the residue decoder (Section 4.5) in reconstructing the residue spectral curve from the audio data. The residue processor is very similar to the floor processor in that it essentially extracts a few vectors from the bitstream and places them in the block RAM for the back-end to use.

### 3.9.2 Design

The same data management issues that arose when designing the floor processor also apply to the residue processor. The vector produced by the module is two dimensional. Most Vorbis files have at most two residues with bound vector sizes. As a result, we chose to store the residue information at fixed memory locations as shown in Tables 2 and 3. Since the constants are so much wider than the actual vectors, they are kept in a separate block ram.

Table 2: Residue Vector RAM

| Description | Width | Length | Element Address |
|---|---|---|---|
| residue_cascade | 8 | 64 | [i] = i |
| residue_books | 8 | 512 | [i], [j] = $(i \ll 3) + j + 65$ |

Table 3: Residue Constants RAM

| Description | Width | Length | Element Address |
|---|---|---|---|
| vorbis_residue_count | 0 | 7 | 0 |
| vorbis_residue_type | 1 | 16 | 1 |
| residue_begin | 2 | 24 | 2 |
| residue_end | 3 | 24 | 3 |
| residue_paritition_size | 4 | 25 | 4 |
| residue_classifications | 5 | 7 | 5 |
| residue_classbook | 6 | 8 | 6 |

Like in the floor, we did not have time to convert the RAM writes to the specification in Tables 2 and 3. Since we knew the residue decoder was not going to be completed in time, we focused on other aspects of the system. It would have presumably been a straight forward task since the writes to the RAM were isolated into the `FN_STATE_WRITE_RAM` state.

The residue setup module is implemented using an 8 state FSM[5]. Like the other minor FSMs attached to the setup processor, one state is dedicated to reading bits from the bitstream and one is used for writing to ram.

- `STATE_INACTIVE`: The module remains in this state while idle. When presented the start signal, the FSM will transition into the `STATE_READ_COUNT` state and begin processing the bitstream.
- `STATE_READ_COUNT`: The number of residues is read in this state. Upon completion, the residue processor enters `STATE_RESIDUE_START`.
- `STATE_RESIDUE_START`: If there are additional residues to parse, then the residue type is read and then the header information is read in `STATE_HEADER_INFO`. Otherwise, the residue setup module becomes inactive.
- `STATE_HEADER_INFO`: A few different variables are read in this state. They are written to their designated addresses in RAM, and upon completion, the FSM enters `STATE_BITMAP`.
- `STATE_BITMAP`, `STATE_BOOKS`: These two states in conjunction create a two dimension vector of residue books. A bitmap is unpacked from the stream and then used to determine what to read to create the residue books.

#### 3.9.3 Testing and Debugging

The residue processor was first tested in simulation to verify state transitions and output values. It was relatively easy to verify that the module was working correctly since the contents of the residue books vector were highly dependent on the correctness of the preceding steps. The residue processor required minimal debugging due to the similarity to the floor processor.

### 3.10 Mapping Processor

#### 3.10.1 Overview

The mapping processor unpacks the mappings from the Vorbis stream. The mappings are used to set up the pipelines used for encoding multichannel audio. Since our decoder was designed for mono audio files, no channel coupling had to take place and thus there was little to no mapping data present in the setup header.

[5] See Vorbis I Specification, Section 8.6.1

### 3.10.2 Design

Although most of the mapping data was not needed by the back-end, the unpacking algorithm[6] had to be implemented in order to maintain bitstream synchronization.

### 3.10.3 Testing and Debugging

The mapping processor was tested in simulation to verify that it read the correct number of bits. The simulation used the fake major FSM to feed the mapping processor data. The test module was then ported to the lab kit to verify that the bitstream sync was maintained before plugging it into the setup processor.

## 3.11 Mode Processor

### 3.11.1 Overview

A Vorbis file can have multiple modes which define the set of configurations to use. For example, a mode has a blocksize, window type, and mapping associated with it. The modes are the last set of data in the setup header.

### 3.11.2 Design

The mode processor reads the number of modes, then it reads a fixed number of bits for each mode. The various configurations are then written to a block RAM from which the back-end can access as needed.

### 3.11.3 Testing and Debugging

The mode processor was tested in simulation and in isolation on the lab kit to verify that it was reading the correct number of bits and correctly identifying the various settings.

## 3.12 Test Modules

A few modules were created in order to facilitate the testing of the front-end. The modules and their uses are described below.

### 3.12.1 Fake Minor FSM

The fake minor FSM was a stub module that mimicked a minor FSM. The module's `BITS_NUMBER` parameter controls how many bits it tried to read when activated. The module was particularly useful in testing the ROM reader, packet filter and setup processor.

[6] See Vorbis I Specification, Section 4.2.4.5.

### 3.12.2 Fake Major FSM

The fake major FSM implemented the bit supply end of a bitstream connection. The module could be instantiated with a sequence of bits, which when requested, it would respond with. Like the fake minor FSM, this module proved vital in verifying that many of the front-end modules worked in simulation and alone on the lab kit.

### 3.12.3 Test RAM

The test ram module implemented a simple block RAM interface. The module used an array of registers to store the data. The length and width could be set with the `RAM_SIZE` and `RAM_WIDTH` parameters, making it versatile in many testing situations.

# 4 Back-end Modules

## 4.1 Standardization

In order to facilitate a consistent design and to maximize ease of debugging, a standard set of interfaces was designed for use by modules that needed their sort of functionality. The three major standardized interfaces used by the back-end were the serial, control, and memory interfaces.

### 4.1.1 The Serial Interface

The serial interface is used by modules to provide a means for synchronously requesting and obtaining data from a bitstream-providing module. It consists of three signals: a bit request signal, a bit availability signal, and a bit value signal. The bit request must be an input for a bitstream provider and an output for the bitstream recipient, and the bit availability and value signals must be outputs for bitstream providers and inputs for bitstream recipients. Essentially, the bitstream provider will assert the bit availability signal and set the bit value appropriately on the clock edge after the edge on which it received the bit request. When the recipient gets an active bit availability signal on a positive clock edge, it should latch or use the bit value during that clock cycle, since the provider is allowed to keep the value there for only one cycle.

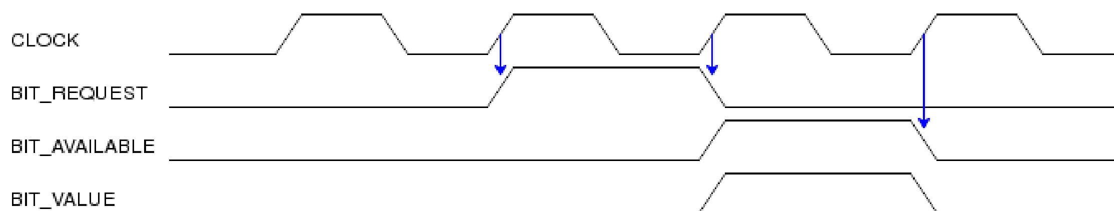


Figure 3: An example timing for the serial interface.

### 4.1.2 The Control Interface

The control interface is used by most modules that provide a specific functionality or act as a minor FSM. It consists of three signals: a start signal, which is an input to the controllee and an output of the controller, a busy signal, which is an output of the controllee and an input to the controller, and a done signal, which is also an output of the controllee and an input to the controller. In general, receipt of the start signal by the controllee module indicates that it should raise its busy signal for the next positive clock edge, and that it should begin computing its function or transitioning through its states. The busy signal should remain high until the controllee is done computing its function, in which case the busy signal is de-asserted for the next clock edge and the done signal is raised for exactly one cycle. It is not a requirement of the interface that a module operate the same way under a single-cycle start pulse or a multi-cycle one, though in all cases in the back-end modules operate on an "if start is high and busy is low" condition that makes this the case.

In cases where the control interface is used to emulate a mechanism similar to a software function call (see 5.6 for an example), the function module's input registers must be loaded with the input data such that the data is present at the start of the clock cycle on which the start is asserted (so data is present as soon as the module is started). When the module must return data, the data must be valid for the cycle during which the done signal is asserted.

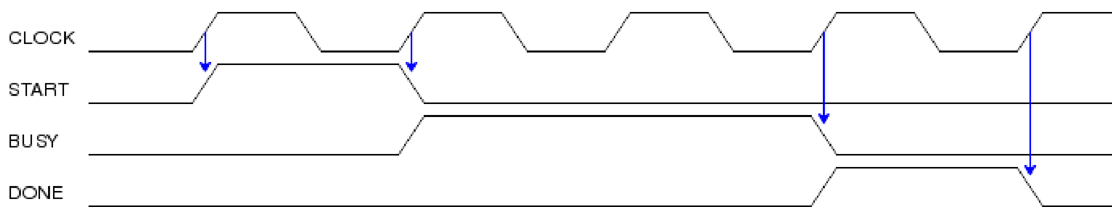


Figure 4: An example timing for the control interface.

### 4.1.3 The Memory Interface

The memory interface is used by modules that need to access, or provide multiplexed access to, a particular block RAM module. The interface consists of two signals, address (or select, usually suffixing a Verilog signal with "`_sel`") and data (usually suffixing a Verilog signal with "`_data`"). The interface does not impose timing requirements, other than that a module must provide data at the selected address within some number of clock cycles after the address is latched; for most modules that provide access to or use the block RAMs, this number is two.

## 4.2 Why not stereo?

Our implementation of the project does allow multi-channel input files. Whether or not the reasons for doing this are obvious, it is nonetheless important to explain our motivations in

doing this. There are two significant motivating factors that drove our decision to implement a mono-only decoder.

The first was complexity - stereo files were far more complicated than mono files (not just twice so) from the perspective of the back-end. Stereo files interleaved data in numerous places, which meant that additional states (and residue types) would have been necessary to produce the correct data. In addition stereo files often used magnitude and angle to represent residues, requiring channel coupling (a fair amount of extra computation) to occur between generation by the residue decoder and storage by the dot product.

The second was processing power - the vast majority of computations had to be performed once for each channel, so in general the decoder would have run at half the speed. Due to variations in the maximum clock frequency that occurred as we added more modules, it was highly unlikely that the decoder would have been able to decode a stereo file in realtime.

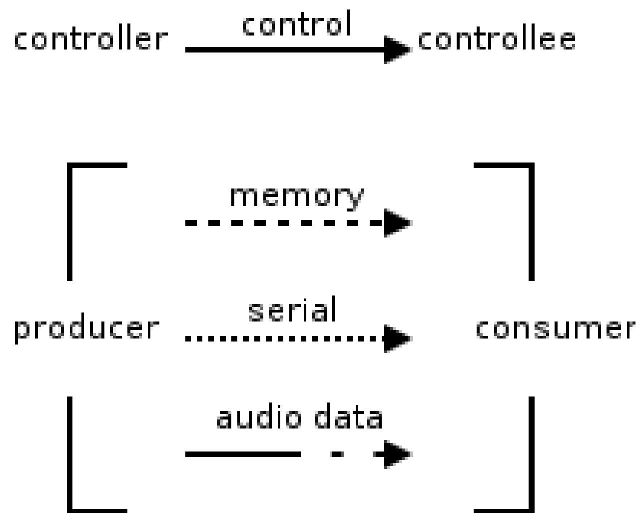


Figure 5: Arrow notation as used in back-end diagrams.

## 4.3 Audio Packet Processor

### 4.3.1 Overview

The audio packet processor module controls the process of audio decoding. When started, the module starts the necessary submodules at the appropriate times. The audio packet processor serves to act as a complete abstraction for the audio packet decoding process; it provides a simple interface for starting the decode process, passing a bitstream, and receiving notification that the module has finished decoding a packet.

### 4.3.2 Mechanics

The audio packet processor is essentially a pair of finite state machines containing instances of those submodules that have a major role in audio decoding - the floor decoder, the residue

decoder, and the IMDCT. The audio packet processor contains a total of 18 states, two of which correspond to one state machine and 16 of which are used by the other. Most of the states transition in a linear fashion, looping back to the first state only from the last state, unless otherwise noted. It is important to note that two FSMs are used because the window select process can occur in parallel with other computations.

- `STATE_IDLE`: an idle state; the audio packet processor is in this state whenever the decoder is reset, before it has started decoding a packet, or immediately after it has finished decoding a packet.
- `STATE_READ_MODE_COUNT`: this state is used to read the mode count from the RAM. The mode count is used to determine how many bits are read for the packed mode number at the start of a packet, and this mode number ultimately determines what methods and codebooks will be used to decode the packet.
- `STATE_READ_BS0`, `STATE_READ_BS1`: these states are used to read the values of the two block size types from RAM. The block sizes determine what window will be applied to the time-domain waveform at the end of processing.
- `STATE_READ_BLOCKFLAG`: this state is used to read the blockflag parameter from RAM; the blockflag is also involved in computing the window.
- `STATE_READ_DELAY_0`, `STATE_READ_DELAY_1`: these states are used to give the RAM enough time to return the values that were selected for in previous states, since the values are used in computations immediately and they must propagate from the block RAM and get latched before they can be used.
- `STATE_INITIALIZE`: this state is used to verify the mode bit that is selected during the last read delay state; if the bit is not zero, than an error condition occurs, and the FSM enters the error state.
- `STATE_INITIALIZE_COUNT`: the number of bits to read for the mode count is computed. This calculation involves taking the integer log of the mode count read in an earlier state. Since the count is bounded, it is computed using comparators and a multiplexer.
- `STATE_INITIALIZE_BITS`: the computed count is placed in a register for the bitreader, and the bitreader is signaled to start.
- `STATE_INITIALIZE_DONE`: this state waits for the bitreader to finish computing, and then parses the result to set window-related flags.
- `STATE_GET_SUBMAP`: this state is used to read the submap number from RAM. The submap is used to determine the floor number. In our implementation, it is a placeholder, since the submap number is zero for almost all single-channel files.
- `STATE_GET_FLOOR`: this state is used to read the floor number from RAM.

- `STATE_FLOOR_DELAY`: this state is used to wait for the floor number to propagate from RAM. It also signals the floor decoder to start computing. Our implementation only decodes floor type 1, since floor type 0 was made obsolete in the 1.0 implementation of the official Vorbis software encoder. Thus, floor type select logic is not included.

- `STATE_FLOOR_DECODE`: the FSM remains in this state while waiting for the floor decode module to complete.

- `STATE_ERROR`: this state is used to indicate that an error in packet decode occurs. The FSM does not leave this state, and when in this state it raises an error signal, so that the user/controller will know to reset the decoder.

- `STATE_WINDOW_STATE_IDLE`: this state indicates that the second audio packet processor FSM is currently waiting to compute the window, or that it has already finished computing the window.

- `STATE_WINDOW_STATE_COMPUTING`: this state indicates that the window select module is currently computing the window.

### 4.3.3 Dataflow

The audio packet processor contains the standard serial communication interface (see Section 4.1.1) for obtaining the bitstream. The bit request output is connected via a multiplexer to the submodules that require interfacing to the bitstream (namely the floor and residue decoders). The multiplexer switches on the state of the primary FSM, so that the proper submodule has complete access to the serial interface at a given time. The audio packet processor also wires address and data lines to the major memories (see 4.1.3) directly to the proper submodules, multiplexing on state if necessary. The standard control interface (see Section 4.1.2) for starting and stopping the decode process is also used by the audio packet processor.

### 4.3.4 Testing and Debugging

The audio packet processor was not difficult to test, since many of its state transitions occur instantaneously (for instance, when the FSM transitions from a read state to a delay state on the first clock cycle after entering it, and then transitions to another read state on the first cycle after that). The only states for which this was not the case were states that transitioned on done signals asserted from submodules, and this activity was easily simulated as well. On the lab kit itself, the audio packet processor was verified against the logic analyzer.

The audio packet processor itself was not actually completed, due to the fact that a fully complete and verifiable implementation was dependent on the presence of the residue decoder.

## 4.4 Floor Decoder

### 4.4.1 Overview

The floor decode module is used to construct an approximation the base energy curve (in frequency) of the original audio spectrum from the bitstream. It decodes only floor type one; Vorbis specifies types zero and one, but the specification notes that as of the official 1.0 release of the Vorbis software encoder, floor type zero is not used.

This module involves an large number of relatively complex states, but conceptually there are three important parts to the process. [7] First, for each floor partition, a value is computed that determines which codebook to read from, and that codebook is used for Huffman-decoding the bitstream to construct a vector of Y values. Next, each of the values are iterated over and a search algorithm ("low/high neighbor" in the Vorbis spec) is employed with some conditional calculations to transform the Y values vector into a "final" Y values vector. Third, each of the vectors in use at this point are sorted (they are originally interleaved), linear interpolation is performed to fill in gaps in the "final" Y vector, and a look-up is performed against the "inverse dB" table to replace each of the values in the "final" vector to produce the (true) final floor curve.

### 4.4.2 Mechanics

The floor decoder is essentially a very large state machine that coordinates memory reads, bitstream reads, various nested iterations, and dataflow to and from several helper function modules. Most of the state transitions are linear, with a handful of "iterate" states that loop back to a previous state. Many of the states start a submodule and then wait for completion of its task. The states involved in floor decoding are:

- `STATE_READ_NONZERO`, `STATE_READ_FLOOR_0`, `STATE_READ_FLOOR_1`: these states are used to read values from the bitstream via the bitstream reader submodule. The values read are, respectively, a "this frame contains no audio energy" flag, and the first two values of the floor intermediate values vector.

- `STATE_READ_2`: this state reads the number of floor values for iteration in the subsequent states.

- `STATE_ITERATE_READ_0`, `STATE_ITERATE_READ_1`, `STATE_ITERATE_READ_2`, `STATE_ITERATE_READ_3`: these states read the a number of index values and parameters from the floor RAMs that determine how many and from which codebook the Huffman-encoded intermediate floor Y values are read.

- `STATE_ITERATE`: this state activates the Huffman decoder and waits for it to return a count for the next part of the decode process.

[7] Refer to the Vorbis I Specification, Section 7.2, for a detailed look at the process.

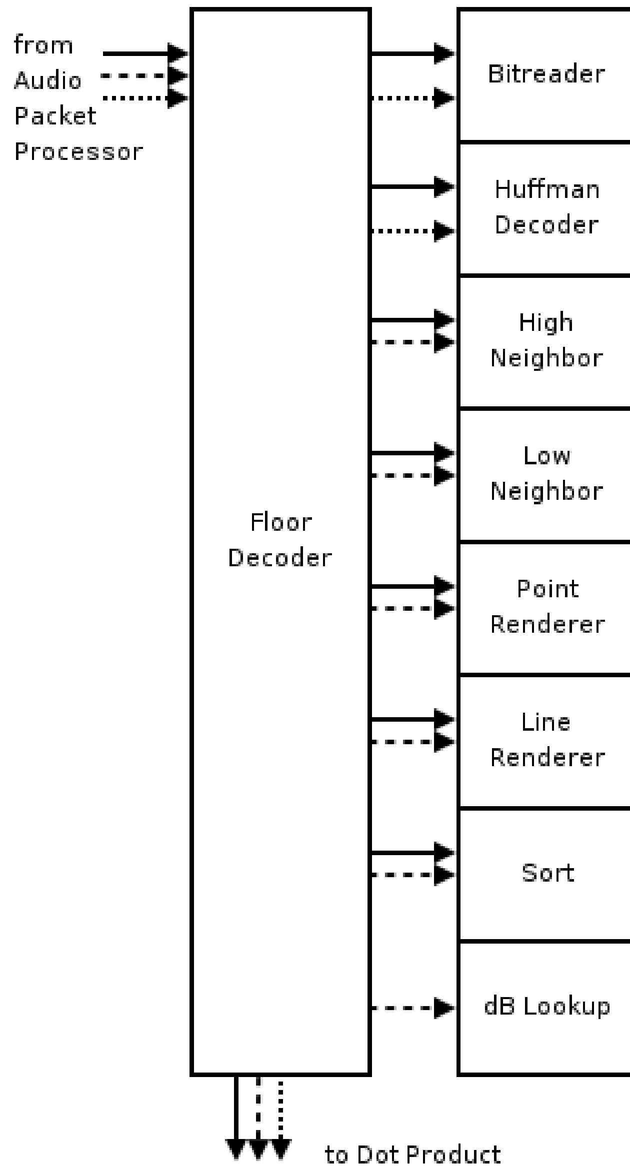


Figure 6: The floor decoder and its helper modules.

- `STATE_VECTORIZE_READ_BOOK`: this state is used to read the codebook value that will determine how the Huffman-encoded parts of the bitstream in subsequent states will be read.

- `STATE_VECTORIZE`: this state is used to manage the loop branching condition and iteration of the "vectorize" loop (the set of states responsible for decoding the intermediate Y-values for the floor energy curve). It additionally uses the Huffman decoder module to obtain these Y-values.

- `STATE_VECTORIZE_READ`: this state is used to store the Huffman-decoded data produced by the Huffman decoder according to the parameters it was given in the previous state. The values are placed in an intermediate register vector.

- `STATE_CURVE_START`: this state is used to set up the point renderer, and it additionally starts and waits on the two neighbor searches ("high" and "low").

- `STATE_POINT`: this state uses the neighbor search values and the point-renderer to find a prediction point, that is optionally used depending on the settings of flags later in the decode process.

- `STATE_PROCESS`: this state performs a series of computations that determine the values that are placed into the next intermediate vector of Y-values.

- `STATE_CURVE_ITERATE`: this state decides whether or not the algorithm must repeat again from the `CURVE_START` state or progress to the next state.

- `STATE_SORT`: this state is used to invoke the sorter, which sorts three intermediate data vectors based on the results of comparisons on one. The sorter module is described in greater detail in section 5.4; it does not perform the actual reads and writes itself, but rather requests data, makes comparisons on the data it receives, and then asserts a swap signal if the values should be swapped, thus implementing the bubble sort algorithm.

- `STATE_FINAL`: this state starts performing the final iteration over intermediate data, this time filling the floor final values RAM with data from the line renderer.

- `STATE_FINAL_ITERATE`: this state determines whether or not more line rendering must occur; if so, it returns to the `FINAL` state instead of moving on.

- `STATE_FINAL_CLEANUP`: in some cases it is necessary to perform a final line rendering to finish the floor curve; if it is, this state is responsible for invoking the line renderer to complete that task.

- `STATE_FINAL_SUBSTITUTE`: this state performs the final processing of the data, iterating over the final line and substituting its values (which are just indices for values from the floor dB look-up table) for the correct values. When this is complete, the floor decode process has finished.

#### 4.4.3 Dataflow

The floor decoder is connected to the serial interface (see section 4.1.1) and floor block RAMs (see section 4.1.3) indirectly, via the audio packet processor. It also uses the standard control interface (see section 4.1.2), as a controllee of the audio packet processor, and also as a controller of the numerous submodules that it uses. The floor decoder module, upon starting, begins requesting bits, never directly but either by means of the bitreader module

or the Huffman decoder module depending on the part of the decode algorithm that is running. The floor decoder has a few small arrays of registers for storing intermediate data, as well as a large block RAM for processing and finalizing its outputs before it sends them off for storage by the dot product module. The floor decoder additionally produces a pair of outputs for viewing its internal state on the logic analyzer.

### 4.4.4 Testing and Debugging

Testing the floor decoder module was easily one of the most time-consuming parts of the project. It was incredibly difficult to test the floor decode module, since proper operation of the module depended on meticulously coordinated data from RAMs and a Huffman-encoded bitstream that was encoded using one of tens of codebooks. It was easy to demonstrate that the floor decode module correctly transitioned states, but since in practice obtaining a valid set of bits (the data for floor decoding was often not byte-aligned) was difficult, as was manually extracting the codebooks and other configuration parameters from an actual Vorbis file. Once this was accomplished for the first time, it was often the case that the floor decoder's outputs would match up perfectly with those of a reference software decoder for certain audio files but not for others.

After finally perfecting the decoder in simulation, it was discovered that a register-based implementation did not synthesize on the lab kit in any reasonable amount of time. Thus, the floor decoder had to be heavily modified to make use of the block RAMs. In performance-critical components of the back-end such as this one, use of the block RAMs often created timing problems, since the RAMs' registered outputs created the need for inserting delays between states. (This was often necessary because data was fetched from memory and then used immediately while running in a tight loop.) The debugging process had to be repeated once again for the RAM-based floor decoder. It was finally demonstrated that the floor decoder transitioned states correctly on the lab kit, and then that it produced the correct values, but ultimately more than half of the time spent implementing the back-end was spent in the modify-optimize-debug cycle for the floor decoder.

## 4.5 Residue Decoder

### 4.5.1 Overview

The residue decoder module is used to produce residue vectors from the bitstream. These vectors are Huffman-encoded and packed in one of three different ways. Ultimately, the decode process for all three types is similar; the only differences between the types occur as a post-processing step of the main residue decode algorithm. Our implementation only decodes type 1 residues, since none of the Ogg Vorbis audio files in our possession used type 0 residues, and type 2 residues were only used by stereo files (which we chose not to support, given the large amount of extra processing required for multi-channel files; see 4.2).

The residue decoder was not nearly as complex as the floor decoder, but was a sizable

module nonetheless. Essentially, the high-level process [8] of residue decoding involves iterating over the partitions of the residue vectors and setting up an array of classifications, then iterating over the partitions again then reading vector quantizations from codebooks selected by these classifications using Huffman-encoded scalar values as offsets in the arrays of vector quantizations. For residue type 1, a post-processing step is required to de-interleave the computed residue vector. [9]

### 4.5.2 Mechanics

The residue decoder essentially consists of a large state machine. Like the floor decoder, it also implements the serial interface (see section 4.1.1), the memory interface (see section 4.1.3) for reading from residue, Huffman, and VQ block RAMs, and the control interface (see section 4.1.2, as both a controller and a controllee. The residue decoder makes heavy use of the Huffman decoder module via this interface, since nearly all of the computationally intensive work in producing the residue vectors is Huffman decoding.

The finite state machine employed by the residue decoder transitions through the following states:

- `STATE_IDLE`: the FSM is in this state whenever the residue decoder has been reset, is waiting to start, or has just finished decoding a residue vector.
- `STATE_PASS_0`: the residue decode algorithm requires special pre-processing of data when in the first ($0^t h$ if considering the index) pass of the eight passes the algorithm makes. This state begins that pre-processing by Huffman-decoding a value from the bitstream.
- `STATE_PASS_0_PROCESS`: this state is used to process the state read in the previous state; a temporary register is set and the classifications vector (as described above) is updated.
- `STATE_PASS_0_ITERATE`: this state is used to control the iteration that occurs in the pass-0 pre-processing. It checks for a transition condition and increments the loop counter.
- `STATE_PROCESS`: this state performs the data processing that occurs in each of the eight passes, reading scalars from the Huffman decoder and using them as offsets in the VQ RAM to obtain VQ vectors.
- `STATE_PROCESS_ITERATE`: this state is used to control iteration during the data processing phase of the computation. Similar to other iteration states, it checks a condition, transitioning if necessary, and increments a loop counter.

[8] The Vorbis I Specification, section 8.6.2, provides a detailed look at the residue decode process.

[9] The Vorbis I Specification, section 8.6.4, describes the additional steps for decoding residue format 1 vectors.

- STATE_FORMAT_0, STATE_FORMAT_2: these states are placeholders for the post-processing steps required by these residue formats. See above (section 4.5.1) for an explanation of why our implementation does not decode these residue types).
- STATE_FORMAT_1: this state is used for the de-interleaving process required by type 1 residues. It reads Huffman-encoded scalars that determine the vector indices for data in the final vector.
- STATE_FORMAT_1_PROCESS: this state is used to perform the required accumulations and temporary calculations required for producing the final residue vector.
- STATE_FORMAT_1_ITERATE: this state is used to determine whether or not the format 1 post-processing needs to be repeated, transitioning to STATE_FORMAT_1_PROCESS if it does. If not, then residue decoding is complete, and the decoder returns to STATE_IDLE.

### 4.5.3 Dataflow

Using the control interface, the audio packet processor starts the residue decoder when the floor decoder has finished decoding the floor. Data is sent from the residue decoder to the dot product module, which stores it in its internal block RAMs, for later use by the IMDCT.

### 4.5.4 Testing and Debugging

Due to the incredible amount of time taken to decode the floor module, and the linear approach to developing the back-end modules, there was not enough time to complete the residue decode module. Some limited simulation was performed on the module, and it was demonstrated that the module was correctly transitioning states and look-up indices. However, the module was never connected to the VQ or residue RAMs, so complete testing of the module in simulation, or any testing on the lab kits, was never accomplished.

## 4.6 Window Select/Look-up

### 4.6.1 Overview

The window select/look-up module is used to provide Y values for specified X values of a window function. In Vorbis, there are two types of window: long and short. Each packet specifies its window type, and the packet block size and various bits are passed to the window select module to realize this window. The module is more than a simple look-up table; the values that it returns are conditionally from a look-up table, depending on the exact shape of the window itself. The module alleviates the need for the computationally-intensive IMDCT (which is where the window function values are applied) to compute these values. Moreover, to increase back-end efficiency, the window select module can be set up in parallel with floor and residue decode. The window select module needs only to prepare itself once per audio packet, and then it can be used rapidly by the IMDCT as often as needed.

### 4.6.2 Mechanics

Internally, the window select/look-up module consists of a small state machine, a look-up table, and a small amount of combinational logic. When the module is started with a set of input flags, it sets up the window constraints as outlined in the Vorbis I specification [10]. The module implements the standard control interface (see section 4.1.2). It consists of a three-state FSM, of which the states are:

- `STATE_START`: when in this state, the module waits for a start signal to arrive, upon which the FSM transitions to `STATE_SETUP`.
- `STATE_SETUP`: when in this state, the module computes the window constraints based on the input parameters of the module. This operation takes a single clock cycle in the current implementation, but for flexibility it is given its own state, so that implementations dependent on external RAMs or resource-constrained devices (i.e. without the space for the module's large look-up table) can prepare the window values. When setup is complete the FSM transitions to `STATE_READY`.
- `STATE_READY`: this state indicates that the window select module is no longer busy, and can produce valid output values when presented with input requests.

The look-up table in the current implementation is a 32-bit wide, 4096-element deep ROM (implemented as a read-only block RAM) that stores pre-computed values of the Ogg Vorbis window function. [11] They are stored according to a maximal fixed-point representation, where the value 1.0 is stored as $2^32 - 1$, the value $-1.0$ is stored as $-2^32$, and values in between are stored approximately as their value times $2^32$.

The combinational logic in the window select module is used to determine whether an input value is in range of the window itself; if the input value is not inside the range defined by the input parameters (according to the start and end values that were produced during the setup state), either 0 (32-bit hexadecimal `0x00000000`) or 1 (32-bit hexadecimal `0x7fffffff`). Otherwise, the appropriate value of the window function is read from the look-up table and returned.

### 4.6.3 Dataflow

The window select module produces data that it utilized solely by the IMDCT module. However, the module's control interface is tied to the audio packet processor, so that it can enter its setup phase and have its computations complete and ready by the time the IMDCT is started.

[10] The Vorbis I Specification, section 4.3.1

[11] The window function is defined in the Vorbis I Specification, in section 4.3.1, to be $y = sin(0.5 * \pi * sin^2(\frac{x+0.5}{n} * \pi)$.

### 4.6.4 Testing and Debugging

The window select module was primarily tested via tests of the IMDCT. Since it was used exclusively there, and was a primary factor in computing the IMDCT's results, invalid or incorrect window select outputs resulted in similarly invalid or incorrect IMDCT outputs. Verification of the IMDCT in simulation helped to identify problems with the window select module, and checking the IMDCT again on the lab kit via the logic analyzer confirmed that the window select module was functioning properly there as well.

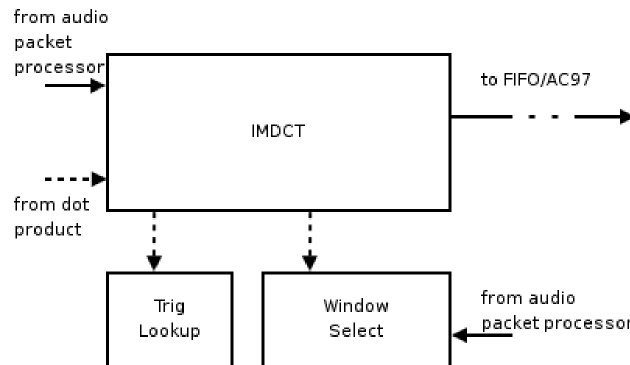


Figure 7: The IMDCT module and its helper modules.

## 4.7 Inverse Modified Discrete Cosine Transform

### 4.7.1 Overview

The IMDCT module is used to transform the dot product of the generated floor and residue vectors from frequency domain back to time domain. The transform itself is essentially an inverse, discrete, and real version of the Fourier transform. It is additionally modified so that a vector twice the length of the input vector is produced.

### 4.7.2 Mechanics

Internally, the module uses the Xilinx fast Fourier transform core. This core computes an ordinary discrete Fourier transform on a variable-length complex-valued vector. In order to produce the correct values for an IMDCT, real and imaginary values are produced by multiplying the input vector by the result of a cosine or sine (for the real and imaginary values respectively). This computation occurs as values are fed into the FFT core, so no additional clock cycles are introduced. The input vector is additionally mirrored horizontally and flipped vertically; as in the case of the vector splitting this computation is also performed on-the-fly. The module also contains a pair of 32-bit sine/cosine look-up tables. These are used for multiplication against the input vector to produce the real and imaginary vector pairs.

The IMDCT module implements one of the three major computational processes of the back-end as minor FSM. The audio packet processor is responsible for emitting its start signal, and for monitoring its busy and done signals so that it may start other processes at the appropriate time. The majority of the IMDCT module's signals are connected directly to the FFT core, and the rest are pre- and post-filtered with multiplies or additions. Thus, the IMDCT module itself serves mostly as a wrapper for the FFT core, containing very little sequential logic itself.

### 4.7.3 Dataflow

Data from the IMDCT module is ultimately provided by the dot product module. Recall that the dot product module is a memory abstraction for producing the dot product of two vectors. Thus, the IDMCT needs only to produce an address (the dot product vector index), and the value of the dot product is produced and latched by the module on the next clock cycle.

The data produced by the FFT is ultimately placed into a FIFO buffer, which is gradually flushed to the AC97 codec. Additionally, the earlier portion of the FFT results must be overlapped with the end of the previous frame, so the overlapped data is computed before buffering. The later portion of the FFT results must be cached for overlapping with the next frame, so this data is cached before buffering.

### 4.7.4 Testing and Debugging

Testing the IMDCT was rather difficult, since the FFT module could not produce values in simulation, and testing on the lab kit was defeated by the large compile time of this module. Ultimately, the majority of testing occurred by first removing the FFT module and verifying that the inputs to the FFT core were correct. After the core's inputs were verified, it was replaced and programmed onto the lab kit. Connecting the module's outputs to the hex display provided a means of verifying that the IMDCT was producing something, and running those numbers by hand allowed verification of the values themselves.

## 4.8 Dot Product

### 4.8.1 Overview

The dot product module is used to temporarily store the floor and residue vectors until they are needed by the IMDCT. The dot product module is technically a misnomer, since the dot product of two vectors is defined to be a scalar, and this module returns a vector that is an element-by-element product of two vectors, but it is called the dot product module to remain consistent with the Vorbis I specification.

### 4.8.2 Mechanics

The dot product module is essentially a specialized RAM abstraction. The module has ready and data inputs for each of two vectors. When the next value of one of the two vectors is ready to be stored, the ready signal is asserted and the value of the data input during that cycle is latched. The module is designed store vectors sequentially (as they are generated by the floor and residue decoders), so no write address is necessary. An internal counter for each of the two vector inputs is incremented each time a vector's ready signal is asserted.

Additionally, there is a select input and data output for reading back from the RAM. Reading from the module returns the dot product of the two vectors at the select index.

### 4.8.3 Dataflow

The decoder back-end uses the dot product module to store the results of floor and residue decode. Since floor and residue vectors must be generated from the bitstream sequentially, they cannot simply be multiplied on-the-fly and stored. Since the memory requirements for the floor and residue vectors are relatively low (16 KB is an exaggerated upper bound for all but the highest-quality files), the dot product module is a simple solution to the problem.

### 4.8.4 Testing and Debugging

The dot product module was tested and verified using a simulation waveform. It was never added to the main module for synthesis on the lab kit, since it was dependent on the residue decoder module, which remains incomplete.

# 5 Back-end Helper Modules

## 5.1 Huffman Decoder

### 5.1.1 Overview

The Huffman decoder module is an abstraction for restoring unsigned integers from a Huffman-encoded stream of bits. The values produced by the Huffman decoder are almost always used as either counts or array indices, and the Huffman decoder takes advantage of this fact, producing results by a codebook look-up. The decoder interfaces to its controlling modules using the control interface and the serial interface for bitstream reading. When appropriate, modules multiplex the serial interface between the bitreader and Huffman decoder.

In Vorbis I, Huffman trees are always implicitly defined in the codebooks [12]; they are stored as a series of codeword lengths. During implementation of this project, an algorithm was developed that exploits this arrangement to allow for high-performance decoding of even very long (in some cases, up to 19-bit) codewords. This algorithm is described in detail in appendix A.

[12] This is described in the Vorbis I specification, section 3.2.1.1.

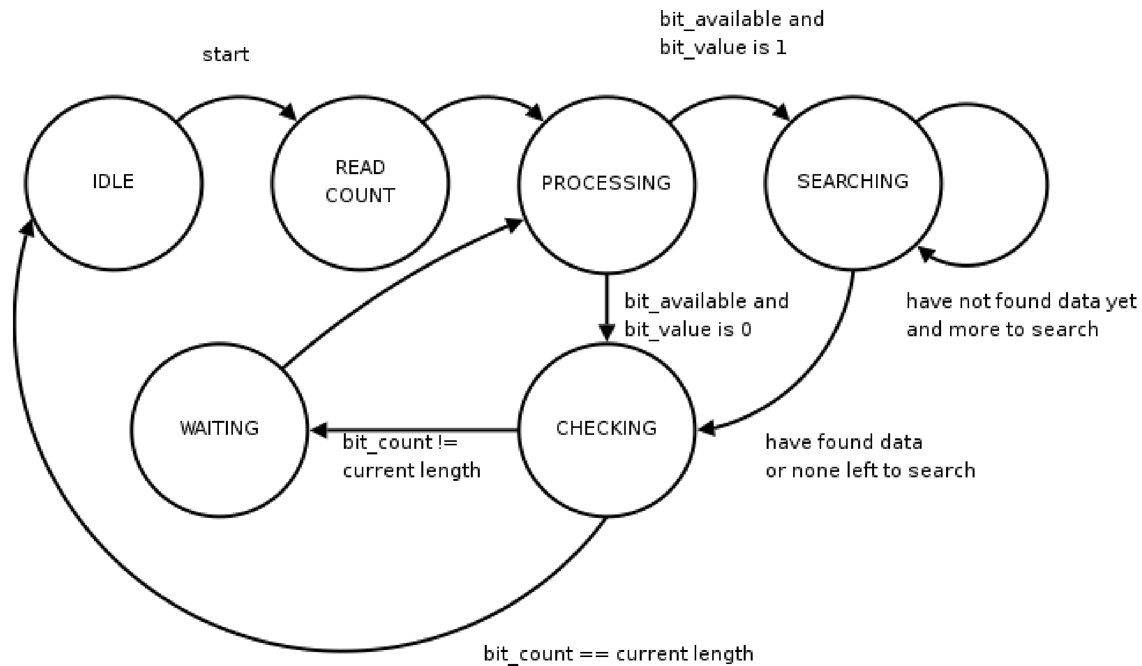


Figure 8: The Huffman decoder state transition diagram.

The Huffman module was implemented using a finite state machine. These states were:

- `STATE_IDLE`: when in this state, the module is waiting for a start signal or has just completed decoding.
- `STATE_READ_COUNT`: when in this state, the module is reading the last value in the codebook, which is used by the algorithm defined in appendix A.
- `STATE_PROCESSING`: when in this state, the module decides whether or not to continue reading bits or to perform a search, based on the bit value signal (if a bit is available).
- `STATE_SEARCHING`: when in this state, the module is performing a binary search on the list of sums using the memory interface (see section 4.1.3). It remains in this state until a value is found or searching cannot continue; for purposes of this algorithm the latter case does not cause an incorrect value to be computed.
- `STATE_CHECKING`: when in this state, the module decides whether or not it should continue decoding values. According the algorithm, if the current number of bits read from the stream is equal to the current value of the length list, it should finish.
- `STATE_WAITING`: this state is used to insert a necessary single-cycle delay before entering `STATE_PROCESSING` if the condition checked in `STATE_CHECKING` fails.

### 5.1.2 Testing and Debugging

The Huffman decoder module took a particularly long time to debug, as refinements to the algorithm were being made as the module was being implemented. The most effective debugging strategy for this module involved the `iverilog` compiler and the use of the Verilog `$display` task to print out as much state information as possible; in this way it was very easy to identify a discrepancy (or an infinite loop, which was a common occurrence immediately after the module was written). Eventually the module was tested as part of the floor decode module on the lab kit, and using the logic analyzer its outputs were verified.

## 5.2 Bitstream Reader

### 5.2.1 Overview

The bitstream reader module was designed to make the task of conforming to the serial interface (see section 4.1.1) easier. The bitstream reader essentially consists of a small FSM that is passed a number of bits to read. When started, the bitstream reader produces the appropriate signals to read a bit, wait for the value, and place the value in a buffer when the value arrives. Since most operations required reading of a sequence of bits as an unsigned integer, the bitstream reader could return a 32-bit integer (padded with zeros) when reading was complete. Each time a bit arrived, the reader would increment an internal counter and set a bit in the buffer according to the current value of the counter.

### 5.2.2 Testing and Debugging

Since it was placed in control of the bit request, availability, and value signals of major modules like the audio packet processor, the floor decoder, and the residue decoder when these modules were in certain states, testing the bitstream reader was mostly done implicitly, by inspecting the values of the bit request, availability, and value signals when these modules were in a particular state. As was often the case, the module was first perfected in simulation, programmed onto the kit (in conjunction with one of the modules that used it), and then debugged on the lab kit using the logic analyzer.

## 5.3 Trig Look-up

### 5.3.1 Overview

The trig look-up module essentially provides an interface for reading from a cosine look-up table. The table contains 256 signed, 16-bit fixed point values (where $2^15 - 1$ represents 1.0 and $-2^15$ represents $-1.0$). The trig module interface allows values to be returned as though calling the cosine function instead of reading from a table. Essentially, the module computes $cos(\frac{n}{256} \cdot \pi)$ for an input $n$. For instance, look-up values greater than 512 are taken modulo 512, and for those still greater than 256, 256 is subtracted and the output data is negated. In this way, passing an "address" that would normally be out-of-bounds still returns a correct result.

### 5.3.2 Testing and Debugging

Given the simplicity of this module, testing was very easy, both in simulation and on the lab kit. Simple tests were created, and both simulation and lab kit outputs were easily verified by hand.

## 5.4 Sort

### 5.4.1 Overview

The Vorbis I Specification requires [13] that during the final stage of curve synthesis during floor decode, three separate lists of intermediate values be sorted according to the values in one of these vectors (i.e. as though we had a map of one vector to the other two and we sorted the map by its keys). This was a particularly interesting challenge in hardware.

The original intention for this module was to implement an insertion sort, since it performs well on short lists like those that needed sorting (despite that it is $O(n^2)$ in the worst-case), and sorting performance would be a critical factor in achieving realtime decoding. However, to facilitate faster completion and testing of the floor decode, a simple bubble sort was implemented and debugged quickly. The bubble sort worked well enough for testing, but due to time constraints and resources spent debugging the floor decode module, the insertion sort implementation was never created.

Regardless, the sort module has some interesting properties. It uses the control interface for working with the floor decode module, and the memory interface for selecting values to sort. The sort module does not perform the sorting itself. Instead, it selects data by address, compares the data when it arrives, and if the data needs to be swapped, it asserts a synchronous "swap" signal. The controller module (in this case, the floor decoder), which is responsible for reading data to the sort module, is also responsible for performing the swap when the signal is asserted.

### 5.4.2 Testing and Debugging

Testing the sort module proved to be a tedious task. It worked perfectly in simulation, yet would not synthesize onto the lab kit, since the compiler did not seem to allow swapping of two values in the same register array. However, the compiler did not make this clear, so a great deal of time was spent attempting to fix the sort module to work on the lab kit. Finally, the floor decoder was modified to store the small intermediate vectors in block RAMs instead of registers, and the sort module was modified to account for the delays in reading from the RAMs and for the inability to write twice in a single cycle. Once these modifications were complete, debugging was performed on the lab kit using the logic analyzer.

[13] The Vorbis I Specification, section 7.2.2.2

## 5.5 Neighbor Search

### 5.5.1 Overview

The neighbor search module implements the `low_neighbor` function, as defined in the Vorbis I Specification.[14] The module uses the standard memory interface (see section 4.1.3) and standard control interface (see section 4.1.2) to perform a search on a vector of values stored in sequential indices in register arrays. The neighbor search function is defined as the greatest vector element for which the element index is less than the argument and the element value is less than the value of the argument value (for `low_neighbor`, or greater than for `high_neighbor`). The module uses a type select input to determine which type of search (0 for low neighbor, 1 for high neighbor) the module will perform, so that a single module can be used for one search type and then the other. In practice, however, the floor decoder uses two neighbor search modules with the types hard-wired to 0 and 1 since the searches can be run in parallel, but for implementations running on area-constrained devices, the type bit might be a convenient option.

### 5.5.2 Testing and Debugging

The neighbor search function was tested in a manner similar to many of the other back-end helper modules, through testbench waveforms in simulation and by confirming values displayed on the logic analyzer when run as part of a larger module.

## 5.6 Line Renderer

### 5.6.1 Overview

The line renderer module implements the `render_line` function, as defined in the Vorbis I Specification.[15] The line renderer, given a pair of endpoint coordinates, computes the Y values that are on the line against a list of X-values and places these values into an intermediate vector. The line renderer uses a set of registers for its data point inputs, and adheres to the standard control interface (see section 4.1.2) for utilization by the floor decoder module. Input values must be present in the registers on the clock edge on which the start signal is asserted.

### 5.6.2 Testing and Debugging

The line renderer module was a simple module to implement, and as such took a very short amount of time to verify. Sample data sets were passed to the module in simulation and were hand-verified. On the lab kit, proper functionality of the line renderer was ensured by tying its outputs to the logic analyzer as it was used in the floor decoder module.

[14] The Vorbis I Specification, sections 9.2.4, 9.2.4.1

[15] The Vorbis I Specification, section 9.4.2.3

## 5.7 Point Renderer

### 5.7.1 Overview

The point renderer module implements the `render_point` function, as defined in the Vorbis I Specification.[16] It computes the Y value for a given X value along a line defined by a pair of given endpoints. The point renderer also uses the standard control interface (as defined in section 4.1.2) for use by the floor decoder module.

### 5.7.2 Testing and Debugging

As one of the less-complex submodules, the point render was easy to test. At first, validation of its outputs took place by confirming the outputs of modules that used it. Checking the outputs of this module by hand was performed in simulation (via a test bench) and on the lab kit (via the logic analyzer) later in the project for consistency.

# 6 The Mini-Decoder

Having determined that it was infeasible to complete the full Ogg Vorbis decoder given the complexity of the back-end and the large amounts of time spent debugging back-end components, we created our own lightweight audio code and a mini-decoder in an attempt to demonstrate many of the working parts of the back-end. Ultimately, we were unable to make the mini-decoder correctly play back a sample compressed by our codec, though we did make great progress in integrating a number of modules to create a complete decoder module, and in interfacing to the AC97 codec; these are two things that we did not have time to complete for the Vorbis decoder.

The codec of which the mini-decoder was a part consisted of a twofold compression. First, audio data was taken 32 samples at a time and transformed into frequency domain using the MDCT, and the frequency values were quantized to 5 bits. Then, the quantized frequency values were analyzed to create an optimal Huffman encoding for the data, and the data was then compressed using that encoding.

Thus, the mini-decoder first had to Huffman-decode 32 values from the bitstream, perform the IMDCT with windowing on these values, and send them to the AC97 codec. The decoder module used an FSM and the serial and control interfaces (see sections 4.1.1 and 4.1.2 respectively) to do this. Iterations were performed over the Huffman encoder until an intermediate data vector was filled with 32 values. The IMDCT was then started, and it selected values as necessary to produce 64 values, half of which it cached for overlap and the other half it added to the last half of the previous frame and output. The IMDCT internally utilized the trig look-up and window select modules from the Vorbis decoder back-end.

The decoder module also contained a FIFO for buffering audio samples to the audio (AC97) module. This was required, since the IMDCT did not necessarily produce values at the times the AC97 became ready for data. The FIFO write enable was connected to the

[16] The Vorbis I Specification, section 9.2.4.2

IMDCT's new data signal, and the FIFO read enable was connected to the audio module's ready-for-data signal. Additional logic used almost-full and almost-empty signals to start and stop the decode process depending on the FIFO's status: if the FIFO was nearly full, it would cease Huffman-decoding new values and passing them to the IMDCT. If it was nearly empty, it would start the process again to fill the FIFO. Though never explicitly measured, it was estimated that the decode process produced data much, much faster than the audio module consumed it, so coordinating the full and empty status of the FIFO was essential.

## 6.1 Testing and Debugging

Due to very large number of clock cycles required before output data was produced, it was difficult to test the entire decoder in simulation. Each of the decoder's core parts (the Huffman decoder, IMDCT, and serial and control interfaces) had been previously verified as parts of the Vorbis decoder. Assembled into the top-level decoder module and synthesized, we were unable to get the mini-decoder working. Debugging was attempted using the logic analyzer, but both Huffman and IMDCT outputs did not appear to be incorrect. Possible causes for the failure of the mini-decoder include interfacing issues with the AC97 codec, and also endianness of the compressed sample in memory.

# 7 Analysis

## 7.1 What Went Right

While we may not have achieved all of our goals in producing the decoder, we think that there were two parts to our design that we would *not* change in retrospect.

The division of modules was one aspect that we feel was particularly good. Modules were separated according to a major FSM/minor FSM pattern, where smaller, specialized modules could be called as though they were functions by other modules, or even invoked in parallel. The fact that the back-end itself was a minor FSM to the front-end, and that the back-end contained utilized separate FSMs for floor and residue decode and the IMDCT, and that these used their own minor FSMs for computing helper functions made it much easier to visualize as well as implement our decoder.

The other aspect that simplified the implementation process and immensely helped avoid bugs was the serial interface. The flow of bits through the decoder was fairly complex, and without a simple but strictly followed method of passing bits from module to module, we probably would not have made as much progress as we did. Passing of bitstreams was particularly essential in the front-end, where the potential of losing sync with the stream (and thus corrupting the decode) was greater.

## 7.2 What Went Wrong

Despite the incredible amount of experience and insight into digital design that we feel we gained, there were several things that, in retrospect, should have been done differently.

First, dividing the labor between the front-end and back-end was not a particularly good idea. In the end, we were unable to finish the back-end, and not only did we have a working front-end, but we had also written a software script earlier in our project to help with debugging the front-end. This script was used to extract front-end data from an Ogg Vorbis file; had we anticipated the amount of work the back-end would have taken to complete, we could have targeted the back-end first, using data extracted using the script to emulate the results of front-end processing. Then, once the back-end was complete and working, we could have moved on to the front-end. Had we proceeded this way, we most likely would have completed the back-end, and would have been able to decode Vorbis against front-end data extracted in software.

Second, the back-end's complexity and algorithms, which seemed to be reasonable to implement in Verilog, turned out to be far more appropriate for running on a processor. An ideal scenario would have been to implement a processor or PIC in Verilog, and to interface this through interrupts to Verilog implementations of performance-critical modules like the IMDCT. Creating a handful of instructions for working with bitstreams (a "read $n$ bits" instruction, for instance), would have made implementation of the back-end's complex data paths much simpler. We feel as though having used a processor of some sort would have made this project much more feasible.

# 8 Conclusion

Ultimately, though we were unable to produce a working decoder, our project was still useful to ourselves, and hopefully to others as well. We learned an incredible amount about digital signal processing, audio compression codecs, and (most importantly) digital design. We gained a relatively significant amount of experience not only writing Verilog code, but also working through the pains of debugging in simulation and then all over again on the FPGA. Finally, in making as much progress as we did in six weeks, we showed that it might not be the case that a CPU and operating system re not necessarily required to decode Ogg Vorbis audio.

# A The Huffman Decoding Algorithm

The Vorbis I specification defines a packing of Huffman data that involves packing only an ordering of the lengths of codewords in a Huffman tree. Thus, the bit sequences of the codewords themselves are implicitly defined. For instance, the Vorbis specification uses an example codebook (packed Huffman tree) containing the values 2, 4, 4, 4, 4, 2, 3, 3. Assigning codewords in order, this codebook implicitly contains the values `00, 0100, 0101, 0110, 0111, 10, 110, 111`.

It is common that a Vorbis audio file might contain tens of these codebooks, some of which contain hundreds of values, of lengths up to around 20. Thus, even for a software-based implementation, creating the bit sequences for these values is an expensive task, and for our hardware implementation, storing these sequences and iterating through them each time we receive a bit would make realtime decoding nearly impossible.

Thus, our Huffman decoder implementation applies an algorithm where the bit sequences are never computed. Instead, we store not just the length of each entry but also an "entry sum" and "entry index". The entry indices are essentially the results of the algorithm; when the algorithm terminates, the entry index at its stopping point is to be returned.

The entry sums are computed as follows: find the maximum codeword length in the codebook (call it $n$) and for each code length $j$ compute $2^{n-j}$. However, instead of storing this number alone, store this number added to the entry sum of the previous entry. For example, given the example listing from the specification, the following entry sums would be produced: `4 5 6 7 8 12 14 16`. The corresponding entry indices would be `0 1 2 3 4 5 6 7`. It is important to note that any length 0 codewords should be omitted; the entry indices vector is created so that length 0 codewords may be skipped and the correct values will still be returned.

In our implementation, all of the previous computation is done by the front-end as it decodes the codebooks, so this data is ready for use by the time the back-end needs to decode a value. When the back-end needs to decode a value, it must keep track of the current bit count and its current position in the list (count is incremented whenever a bit is received; position starts at zero). When it receives a bit, either one of two things happens:

- the bit received is a 0, and the bit count is less than the entry length at the current index; request a new bit.
- the bit received is a 0, and the bit count equals the entry length at the current index; return the entry index at the current index.
- the bit received is a 1; compute the value "max(entry sums) >> bit count" and perform a binary search on the entry sums list for this value plus the value of the previous search (or 0, if no search has occurred yet). When the value is found, set index to be the index of this value plus one. If the bit count is equal to the entry length at the current index, return the entry value at the current index. Otherwise, continue.

Using the example codebook, let us consider the bit sequence `0101`, which corresponds to an entry value of 2.

1. Bit 1 is 0, so our index remains at 0 because the count is one and the current length is 2.

2. Bit 2 is a one, so we compute $16 >> 2$ to get 4. A searching for 4 finds it at index 0, so we add one to this and set our index to 1. The current length is 4 and our count is at 2, so we continue.

3. Bit 3 is a zero, the current length is 4, and our count is at 3, so we continue.

4. Bit 4 is a one, so we compute $16 >> 4$, which is 1, and we add this to the previous search (4) to get 5, and binary search for 5. We find it at index 1, and add one to this and get 2. Since the bit count is 4 and the length at index 2 is 4, we stop and return the entry at index 2, which is 2.

As it turns out, checking the implicit bit sequences defined earlier, the bit sequence `0101` corresponds to the value 2.