IMPLEMENTATION OF REAL TIME DECODING OF OGG VORBIS ON ANALOG DEVICES SHARC ADSP-21364

PROJECT REPORT

Submitted by

Akella Karthik (04EC03)
Ashish Shenoy (04EC19)
Sanjay Rajashekhar (04EC52)
Siddhartha Sampath (04EC59)

Under the guidance of

Prof. Sumam David S
Department of Electronics and Communications Engineering
National Institute of Technology – Karnataka

And

Mr. Aseem V Prabhu
Analog Devices Inc., Bangalore

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF
BACHELOR OF TECHNOLOGY
IN
ELECTRONICS AND COMMUNICATIONS ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL
SRINIVASNAGAR – 575025 KARNATAKA, INDIA

OggVorbis

DEDICATED

TO

ALL OUR TEACHERS,

THE INDUSTRY,

OUR FRIENDS AND

THE CODERS OF THE OPEN SOURCE COMMUNITY.

www.xiph.org

DECLARATION

We hereby declare that the Project Work Report entitled **IMPLEMENTATION OF REAL TIME DECODING OF OGG VORBIS ON ANALOG DEVICES SHARC ADSP 21364** which is being submitted to the **National Institute of Technology Karnataka, Surathkal** for the award of the Degree of **Bachelor of Technology** in **Electronics & Communications Engineering** is a bonafide report of the work carried out by us. The material contained in this Project Work Report has not been submitted to any University or Institution for the award of any degree.

Register Number, Name & Signature of the Students:

(1) 04EC03, Akella Karthik

(2) 04EC19, Ashish Shenoy P

(3) 04EC52, Sanjay R

(4) 04EC59, Siddhartha Sampath

Department of Electronics & Communication Engineering

Place: NITK, SURATHKAL
Date: 28th March 2008

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA, SURATHKAL
SRINIVASANAGAR – 575025, KARNATAKA

CERTIFICATE

This is to certify that the project entitled **IMPLEMENTATION OF OGG VORBIS ON ANALOG DEVICES SHARC ADSP – 21364 PROCESSOR** is a record of the bona fide work done by **AKELLA KARTHIK** (*Reg. No* 04EC03), **ASHISH SHENOY** (*Reg. No.* 04EC19), **SANJAY R** (*Reg. No.* 04EC52*) and* **SIDDHARTHA SAMPATH.** (*Reg. No* .04EC59) in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in Electronics and Communication Engineering at the National Institute of Technology Karnataka, Surathkal during the academic year 2007 – 2008.

Mr. Aseem V Prabhu
Project Manager, DSP Applications,
Analog Devices India Pvt. Limited,
Bangalore.

Dr. Sumam David
Professor,
Dept of ECE,
NITK, Surathkal.

Dr. B. Shankarananda
Head of Department,
Dept of ECE,
NITK, Surathkal

Place: NITK, Surathkal
Date: 28th March 2008

Acknowledgements

It gives us immense pleasure to thank all those who helped us during the course of our project.

We would like to thank our guide and mentor Dr. Sumam David, Professor, Department of ECE, NITK Surathkal. She will be a constant source of inspiration for us. Her treasured support and guidance through the course of the project was instrumental in the project work.

We would like to thank Mr. Aseem V Prabhu, ADI, Bangalore, for providing us this opportunity to work on an Analog Devices' sponsored project. We would like to thank him for his support during the different stages of the project. We are deeply grateful to Mr. Kulin Seth, ADI, Bangalore and Mr. Bharan Giridhar, TI, Bangalore for providing us with all the technical support that we needed during the course of the project, without which we would have been stuck up at various junctures of the project.

We also thank Dr. B. Shankarananda, Head of Department, Department of ECE, NITK Surathkal, for ensuring that all the required facilities in the department were available during project work.

We would also like to thank Ms. Vagdevi Prabha of DSP Lab, ECE Dept, and Mr. Kranti, M.Tech student, ECE Dept, NITK Surathkal for the persevered, relentless help they gave by keeping the lab open.

We extend our gratitude to all coders of the open source forum for all the technical support we received from them.

Finally, we would also like to convey our gratitude to our parents and all our friends for their moral support, without which, we wouldn't have tread so far.

Abstract

The objective of this project is to implement real-time decoding of an Ogg Vorbis audio file on the floating point processor SHARC ADSP-21364 using EZ-KIT LITE evaluation board. Ogg Vorbis is a general-purpose perceptual compressed audio format for mid to high quality audio and music at fixed and variable bit rates. The dynamic probability model of the codec results in a very large working memory. A technique for significantly reducing this memory requirement is presented here.

The module-wise memory requirement analysis of the codec reveals that vector table decode occupies maximum heap memory. In the reference implementation, the vector table decode is done during codec initialization. In order to reduce memory requirement the vector table decode is moved from initialization to the real-time decode loop using a custom implementation. Alternate representations of data structures, like compressed-array-representation of a tree, are used to reduce the execution time of critical section of the code.

It is for the first time that the Ogg Vorbis floating point decoder has been implemented in real time on a DSP, at approximately 80 MIPS and 0.7 Mbits of heap.

Contents

1 Introduction **1**

1.1 Objective . . . 1

1.2 Motivation/Need analysis . . . 1

1.3 State of the art . . . 2

1.4 Ogg Vorbis Vs other audio codecs . . . 2

1.5 Organisation of the report . . . 4

2 Ogg Vorbis Specifications **5**

2.1 Ogg specifications . . . 5

2.1.1 Introduction . . . 5

2.1.2 Ogg bitstream . . . 5

2.1.3 Ogg encapsulation process . . . 8

2.1.4 Ogg page format . . . 11

2.1.5 Ogg decoding . . . 14

2.2 Vorbis specifications . . . 14

2.2.1 Overview . . . 14

2.2.2 Vorbis encoding . . . 16

2.2.3 Vorbis decoding . . . 17

2.2.4 Packet decode process . . . 19

3 ADSP-21364 processor and EZ-KIT LITE specifications **23**

3.1 EZ-KIT LITE evaluation board specifications . . . 23

3.1.1 Board overview . . . 23

3.1.2 External memory . . . 26

3.1.3 DAI interface . . . 26

3.1.4 Analog audio . . . 26

3.2 SHARC ADSP-21364 processor . . . 27

3.2.1 Overview . . . 27

3.2.2 Architectural features . . . 29

3.2.3 ADSP-21364 memory and I/O interface features . . . 29

3.3 Development tools . . . 32

4 Implementation **34**
4.1 Approaches to problem solving 34
4.2 Implementation details 35
4.2.1 Stage 1 35
4.2.2 Stage 2 35
4.2.3 Stage 3 38
4.2.4 Stage 4 39

5 Summary of results **48**

6 Conclusions and scope for future work **50**
6.1 Conclusions 50
6.2 Scope for future work 51

List of Figures

2.1 Multiplexed logical bitstreams in a physical bitstream 6
2.2 Packet segmentation and resulting lacing values 8
2.3 Formation of a multiplexed Ogg page from data packets by segmentation 10
2.4 Ogg page format 11
2.5 Flowchat for Ogg decoding 15
2.6 Vorbis encoding functional diagram 16
2.7 Vorbis decode flowchart 18
2.8 Vorbis packet decode flowchart 19
2.9 Equally sized window overlap 20
2.10 Unequally sized window overlap 20

3.1 EZ-KIT LITE board schematics 24
3.2 Digital Applications Interface (DAI) schematics 24
3.3 SHARC ADSP-21364 architecture 27
3.4 Internal I/O processor architecture 28

4.1 Heap requirement estimation in MSVC++ 36
4.2 Heap reduction after byte packing at synchronization and streaming layers 36
4.3 Total heap usage without vector table generation 38
4.4 Internal memory bus architecture of ADSP-21364 41
4.5 Changing core clock frequency using PMCTL 43
4.6 Statistical profiling results before CAR optimization (incomplete list) 44
4.7 Huffman tree translation to CAR 45
4.8 Statistical profiling results after CAR optimization (incomplete list) 46
4.9 Memory map of ADSP-21364 47

Chapter 1

Introduction

1.1 Objective

The objective of this project is to implement real-time decoding of an Ogg Vorbis audio file on the SHARC ADSP-21364 using EZ-KIT LITE evaluation board. This is brought about by developing a comprehensive understanding of the Ogg Vorbis codec, the SHARC ADSP-21364 architecture and the VisualDSP++ integrated development environment.

1.2 Motivation/Need analysis

Ogg Vorbis is a fully open, non-proprietary, patent-free, and royalty-free, general-purpose compressed audio format for mid to high quality audio and music at fixed and variable bitrates. This places Vorbis in the same competitive class as audio representations such as MPEG-4 (AAC), and similar to, but higher performance than MP3, TwinVQ (VQF), WMA and PAC.

The following briefly describe the benefits of this codec

- As already specified, Ogg Vorbis specification is in the public domain; it is free for commercial or non-commercial use, under both GPL and BSD licenses. So the source code is readily available and for free use.
- Listening tests have suggested that this codec shows a good all-round performance for moderate to high bit rates.
- As this codec is specified in the public domain, the documentation and specifications are well written.
- This codec is also supported by most portable digital audio players (DAPs), such as those provided by Samsung, Rio and iRiver.

- This codec is extremely appropriate and well-suited for internet-streaming (via Ice cast and other methods) because of the inherent packet-based processing nature of the codec.
- This codec also provides for a fully gapless playback.
- This codec also provides a high potential for further tuning.
- This codec is also structured to allow the design for a hybrid filter bank (Vorbis II).

For all the above mentioned reasons, implementation of Ogg Vorbis decoder on embedded platforms has a high market potential. The decoder has not yet been implemented in real-time on ADSP 21364.

1.3 State of the art

Vorbis is the first of a planned family of Ogg multimedia coding formats being developed as part of Xiph.org's Ogg multimedia project. Vorbis development began following a September 1998 letter from Fraunhofer Gesellschaft announcing plans to charge licensing fees for the MP3 format. Soon after, founder Christopher Montgomery commenced work on the project. A stable version of the codec was released on July 19, 2002. Today a number of digital audio players support Ogg Vorbis. Vorbis is well-supported on the Linux platform in programs like XMMS, xine, and many more.

This project has been given the necessary background by the implementation of Tremor (fixed point implementation of the decoder) on the VDSP++ ADSP 21364 emulator. However, the implementation was not real time. Research has been done regarding the memory shortage issues on the platform. A memory optimization procedure has been designed for the Tremor code which has to be applied, with suitable changes, to the floating point decoder implementation.

1.4 Ogg Vorbis Vs other audio codecs

The Ogg Vorbis format has proven popular among supporters of free software. They argue that its higher fidelity and completely free nature, unencumbered by patents, make it a well-suited replacement for patented and restricted formats like MP3. However, MP3 has been widely used since the mid-1990s and as of 2007, continues to dominate in the consumer electronics industry. Of the consumer products supporting lossy compressed digital audio, virtually all support playback of MP3 audio while relatively few support alternative formats like Ogg Vorbis.

In the commercial sector, Vorbis support is on the rise. Many mainstream video game titles such as Unreal Tournament and Grand Theft Auto: San Andreas store in-game audio as Vorbis. Popular software players support Ogg Vorbis playback either natively or through an external plugin. A number of Web sites use it, such as Jamendo and Mindawn, as well as several national radio stations like CBC Radio JazzRadio and Virgin Radio.

For many applications, Vorbis has clear advantages over other lossy audio codecs in that it is patent-free and has open-source implementations and therefore is free to use, implement, or modify as one sees fit, yet produces smaller files than most other codecs at equivalent or higher quality.

Listening tests have attempted to find the best quality lossy audio codecs at certain bit rates. Some conclusions made by recent listening tests:

- Low bit rate (less than 64 kb/s): the most recent public multiformat test at 48 kb/s shows that aoTuV Ogg Vorbis has a better quality than WMA and LC-AAC, the same quality of WMA Professional, and a lower quality than HE-AAC.

- Mid to low bit rates (less than 128 kb/s down to 64 kb/s): private tests (80 kb/s, 96 kb/s) shows that aoTuV Ogg Vorbis has a better quality than other lossy audio codecs (LC-AAC, HE-AAC, MP3, MPC, WMA).

- Mid bit rate (128 kb/s): most recent public multiformat test at 128 kb/s shows a four-way tie between aoTuV Ogg Vorbis, LAME-encoded MP3, WMA Pro, and QuickTime AAC, with each codec essentially transparent (sounds identical to the original music file).

- High bit rates (more than 128 kb/s): most people do not hear significant differences. However, trained listeners can often hear significant differences between codecs at identical bit rates and aoTuV Ogg Vorbis performs very well, i.e. better than other formats such as AAC, MP3, and MPC.

Knowledge of Vorbis's specifications is in the public domain. Concerning the specification itself, Xiph.Org reserves the right to set the Vorbis specification and certify compliance. Its libraries are released under a BSD-style license and its tools are released under the GPL (GNU General Public License). The libraries were originally released under the GNU Lesser General Public License, but a BSD license was later chosen with the endorsement of Richard Stallman. The Xiph.Org Foundation states that Vorbis, like all its developments, is completely free from the licensing or patent issues raised by other proprietary formats such as MP3. Although Xiph.Org states it has conducted a patent search that supports its claims, outside parties (notably engineers working on rival formats) have expressed doubt that Vorbis is free of patented technology.

Ogg Vorbis is supported by several large digital audio player manufacturers such as Samsung, Rio, Neuros Technology, Cowon, and iRiver. Many feel that the growing support for the Vorbis codec within the industry supports their interpretation of its patent status, as multinational corporations are unlikely to distribute software with questionable legal status. The same could be said about its growing popularity in other commercial enterprises like mainstream computer games.

1.5 Organisation of the report

Chapter 1 provides a clear objective of the project and further, gives overview of the Ogg Vorbis codec. It provides the background of the codec and also clearly indicates the advantage of this codec over the other lossy codecs. It further discusses the motivation for working on this project.

Chapter 2 discusses the theory and principles involved in the Ogg Vorbis codec. It discusses the Ogg container and the Vorbis audio codec in two separate sections. It also provides detailed flowcharts for Ogg and Vorbis decoding.

Chapter 3 discusses the target SHARC ADSP-21364 hardware and its evaluation board EZ-KIT LITE features in detail. It also deals with memory and I/O interface features of the DSP bearing relevance to this project. It then goes ahead to discuss the industry standard tool VDSP++ IDDE, which was heavily used during the project development for simulation as well as execution on the emulator.

Chapter 4 focuses on memory requirements and constraints, memory optimization techniques, buffer management for file IO removal and other issues while making the implementation real-time. It also looks at an alternative data structure organization for reduction of required MIPS.

Chapters 5 and 6 discuss the results, conclusions and scope for future work.

Chapter 2

Ogg Vorbis Specifications

2.1 Ogg specifications

2.1.1 Introduction

The term ‘Ogg’ refers to a general purpose data container format. Ogg container format encapsulates Vorbis-encoded audio data. Hence the name ‘Ogg Vorbis’.

Ogg uses octet vectors of raw, compressed data or packets. These compressed packets do not have any high-level structure or boundary information; strung together, they appear to be streams of random bytes with no landmarks. But the structure in these data packets is implicit and is based on certain fields in the Ogg page header.

Ogg Vorbis uses the Ogg framework to encapsulate Vorbis-encoded audio data for stream-based storage, such as files, and transport, such as TCP streams or pipes. Ogg can also encapsulate other A/V codecs like Theora, FLAC, Speex etc.

2.1.2 Ogg bitstream

The Ogg transport bitstream is designed to provide framing, error protection and seeking structure for higher-level codec streams that consist of raw, unencapsulated data packets, such as the Vorbis audio codec or Tarkin video codec.

Vorbis encodes short-time blocks of PCM data into raw packets of bit-packed data. These raw packets may be used directly by transport mechanisms that provide their own framing and packet-separation mechanisms, such as UDP datagrams. Vorbis uses the Ogg bitstream format to provide framing/sync, sync recapture after error, landmarks during seeking, and enough information to properly separate data back into packets at the original packet boundaries without relying on decoding to find packet boundaries.

Design constraints for Ogg bitstreams, as described in [1], can be briefly summarized and stated as below:

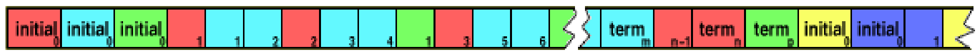


Figure 2.1: Multiplexed logical bitstreams in a physical bitstream

- Framing for logical bitstreams
- Interleaving of different logical bitstreams
- Detection of corruption
- Recapture after a parsing error
- Position landmarks for direct random access of arbitrary positions in the bitstream
- Streaming capability (no seeking is needed to build a 100% complete bitstream)
- Small overhead (using no more than approximately 1-2% of bitstream bandwidth for packet boundary marking, high-level framing, sync and seeking)
- Simplicity to enable fast parsing
- Simple concatenation mechanism of several physical bitstreams

Ogg provides a generic framework to perform encapsulation of time-continuous bitstreams. It does not know any specifics about the codec data that it encapsulates and is thus independent of any media codec.

Logical bitstream

Ogg can carry pages from different codec instances multiplexed to form a single bitstream. A logical bitstream consists of pages, in order, belonging to a single codec instance. Thus, logical bitstream is a sequence of bits being the result of an encoded media stream. As shown in Figure 2.1 each color corresponds to a logical bitstream. A stream always consists of an integer number of pages. Each page is a self contained entity (although it is possible that a packet may be split and encoded across one or more pages); that is, the page decode mechanism is designed to recognize, verify and handle single pages at a time from the overall bitstream.

Physical bitstream

Multiple logical bitstreams can be combined, with restrictions, into a single 'physical bitstream'. Physical bitstream is a sequence of bits resulting from an Ogg encapsulation of one or several logical bitstreams. Logical bitstreams may not be mapped or multiplexed into physical bitstreams without restriction. Each media format defines its own restrictive mapping. A physical bitstream consists of a sequence of pages from the logical bitstreams with the restriction that the pages of each logical bitstream must come in their correct temporal order.

Thus, a physical bitstream consists of multiple logical bitstreams interleaved in pages and may include a 'meta-header' at the beginning of the multiplexed logical stream that serves as an identification. The simplest physical bitstream is a single, unmultiplexed logical bitstream with no meta-header; this is referred to as a 'degenerate stream'.

The logical bitstreams are identified by a unique serial number in the header of each page of the physical bitstream. This unique serial number is created randomly and does not have any connection to the content or encoder of the logical bitstream it represents. Pages of all logical bitstreams are concurrently interleaved, but they need not be in a regular order - they are only required to be consecutive within the logical bitstream.

Bitstream structure

The bitstreams provided by encoders are handed over to Ogg as so-called 'packets' with packet boundaries dependent on the encoding format. From Ogg's perspective, packets can be of any arbitrary size. Ogg divides each packet into 255 byte long chunks plus a final shorter chunk. These chunks are called 'Ogg segments'. The segmentation process is a logical one; it's used to compute page header values and the original page data need not be disturbed, even when a packet spans page boundaries. They do not have a header for themselves. A group of contiguous segments is wrapped into a variable length page preceded by a page header. Both the header size and page size are variable; the page header contains sizing information and checksum data to determine header/page size and data integrity.

Each Ogg page contains only one type of data as it belongs to one logical bitstream only. Pages are of variable size and have a page header containing encapsulation and error recovery information. The encapsulation specification for one or more logical bitstreams is called a 'media mapping'. A specific media mapping will define how to group or break up packets from a specific media encoder. See [1] for more details.

```
raw packet:
  ___________________________________________
 |_____________packet data__________________| 753 bytes

lacing values for page header segment table: 255,255,243

 raw packet:
   ______________________________
  |_______packet data___________|          255 bytes

 lacing values: 255, 0
```

Figure 2.2: Packet segmentation and resulting lacing values

2.1.3 Ogg encapsulation process

Ogg pages can have a maximum size of about 64 kB, sometimes a packet has to be distributed over several pages. As mentioned earlier, the raw packet is logically divided into n 255-byte segments and a last fractional segment of < 255 bytes. A packet size may well consist only of the trailing fractional segment, and a fractional segment may be zero length. These values, called 'lacing values', are then saved and placed into the header segment table.

Packets are not restricted to beginning and ending within a page, though individual segments are required to do so. The Ogg bitstream specification strongly recommends nominal page size of approximately 4-8 kB. After segmenting a packet, the encoder may decide not to place all the resulting segments into the current page; to do so, the encoder places the lacing values of the segments it wishes to belong to the current page into the current segment table, and then finishes the page. The next page is begun with the first value in the segment table belonging to the next packet segment, thus continuing the packet (data in the packet body must also correspond properly to the lacing values in the spanned pages. The segment data in the first packet corresponding to the lacing values of the first page belong in that page; packet segments listed in the segment table of the following page must begin the page body of the subsequent page).

The last mechanic to spanning a page boundary is to set the header flag in the new page to indicate that the first lacing value in the segment table continues rather than begins a packet; a header flag of 0x01 is set to indicate a continued packet. Although mandatory, it is not actually algorithmically necessary; one could inspect the preceding segment table to determine if the packet is new or continued. Adding the information to the packet header flag allows a simpler design with no overhead that needs only inspect the current page header after frame capture. This allows faster error recovery in the event that the packet originates in a corrupt preceding page, implying that the previous page's segment table cannot be trusted. The above spanning process is repeated for each spanned page boundary. A 'zero termination' on a packet size that is an even multiple of 255 must appear even if the lacing value appears in the next page as a zero-length continuation of the current packet. The header flag should be set to 0x01 to indicate that the packet spanned, even though the span is a nil case as far as data is concerned.

A segment table in the page header tells about the lacing values of the segments included in the page. A flag in the page header tells whether a page contains a packet continued from a previous page or not. Note that a lacing value of 255 implies that another lacing value follows in the packet, and a value of less than 255 marks the end of the packet after that many additional bytes.

A packet of 255 bytes (or a multiple of 255 bytes) is terminated by a lacing value of 0. A 'nil' (zero length) packet is not an error; it consists of nothing more than a lacing value of zero in the header. For example a packet of length 753 bytes would would result in lacing values 255, 255, 243 in the segment table. A packet of length 255 bytes would result in lacing values 255, 0. This process is illustrated in Figure 2.2.

Each logical bitstream in a physical Ogg bitstream starts with a special page (BOS = Beginning Of Stream) and ends with a special page (EOS = End Of Stream). The first page of a logical Ogg bitstream consists of a single, small 'initial header' packet that includes sufficient information to identify the exact codec type and media requirements of the logical bitstream. BOS page is the initial page of a logical bitstream, which contains information to identify the codec type and other decoding-relevant information. It should contain the sample rate and number of channels. By convention, the first bytes of the BOS page contain magic data that uniquely identifies the required codec. Ogg Vorbis provides the name and revision of the Vorbis codec, the audio rate and the audio quality in the Ogg Vorbis BOS page. There is no fixed way to detect the end of the codec-identifying marker. The format of the BOS page is dependent on the codec and therefore must be given in the encapsulation specification of that logical bitstream type.

Ogg also allows, but, does not require, secondary header packets after the BOS page for logical bitstreams and these must also precede any data packets in any logical bitstream. These subsequent header packets are framed into an integral number of pages, which will not contain any data packets. Ogg Vorbis uses two additional header pages per logical bitstream.

So, a physical bitstream begins with the BOS pages of all logical bitstreams containing one initial header packet per page, followed by the subsidiary header packets of all streams, followed by pages containing data packets. The Figure 2.3 shows a schematic example of a media mapping using Ogg and grouped logical bitstreams. Different types of multiplexing logical bitstreams are discussed in [1].

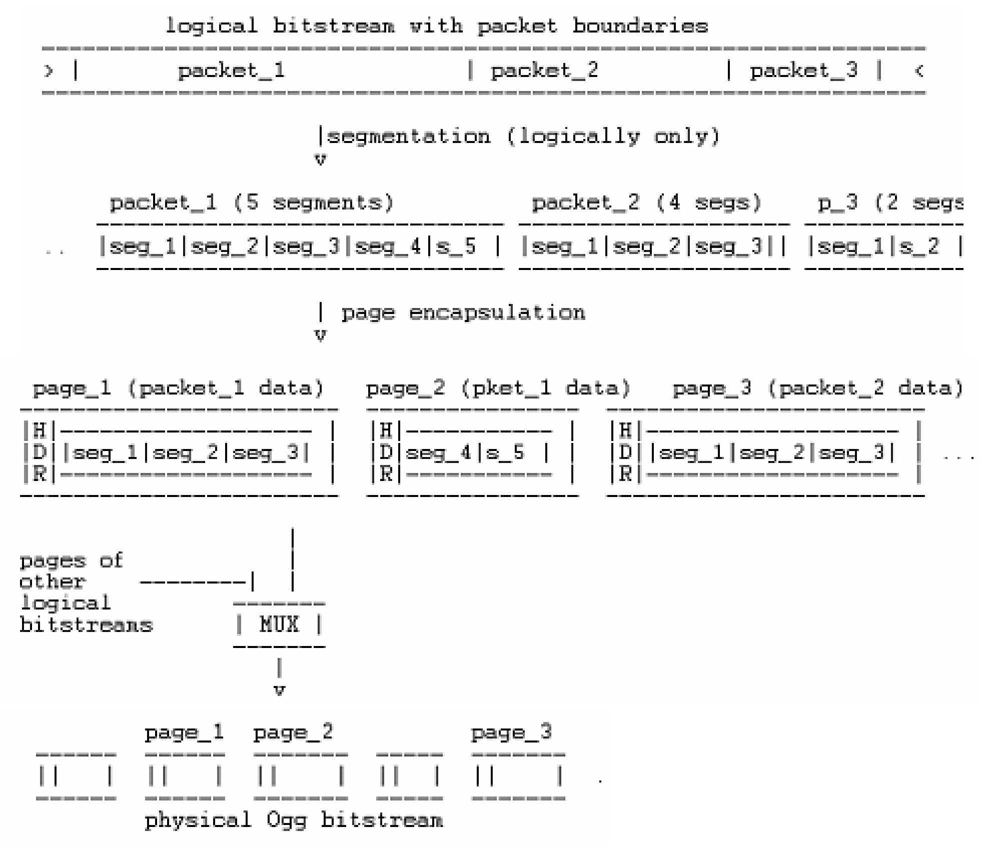


Figure 2.3: Formation of a multiplexed Ogg page from data packets by segmentation

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1| Byte
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| capture_pattern: Magic number for page start "OggS"           | 0-3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| version       | header_type   | granule_position              | 4-7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               | 8-11
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               | bitstream_serial_number       | 12-15
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               | page_sequence_number          | 16-19
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               | CRC_checksum                  | 20-23
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |page_segments  | segment_table | 24-27
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| ...                                                           | 28-
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.4: Ogg page format

2.1.4 Ogg page format

A physical Ogg bitstream consists of a sequence of concatenated pages. Pages are of variable size, usually 4-8 kB, and maximum 65307 bytes. A page header contains all the information needed to demultiplex the logical bitstreams out of the physical bitstream and to perform basic error recovery and landmarks for seeking. Each page is a self-contained entity such that the page-decode mechanism can recognize, verify, and handle single pages at a time without requiring the overall bitstream. The Ogg page has the format shown in Figure 2.4.

The LSB (least significant bit) comes first in the Bytes. Fields with more than one byte length are encoded LSB (least significant byte) first.

The fields in the page header as shown in Table 2.1 have the following meaning:

1. Capture pattern:
   A header begins with a capture pattern that simplifies identifying pages; once the decoder has found the capture pattern it can do a more intensive job of verifying that it has found a page boundary. It is a 4 byte field that signifies the beginning of a page. It contains the magic pattern ‘OggS’. It helps a decoder to find the page boundaries and regain synchronization after parsing a corrupted stream. Once the capture pattern is found, the decoder verifies page sync and integrity by computing and comparing the checksum.

2. Stream structure version:
   It is a single byte signifying the version number of the Ogg file format used in this stream (specified here as version 0). The capture pattern is followed by the stream structure version.

3. Header type flag:
   The bits in this 1-byte field identify the specific type of this page (the page’s

Table 2.1: Ogg page header contents

Byte	Value	Remarks
0	0x4F	‘O’
1	0x67	‘g’
2	0x67	‘g’
3	0x53	‘S’
4	0x00	Stream structure version
5	0x01	set—continued packet
		unset—fresh packet
	0x02	set–first page of logical bitstream (BOS)
		unset—not first page of logical bitstream
	0x04	set—last page of logical bitstream (EOS)
		unset—not last page of logical bitstream
6—13	0xXX LSB — 0xXX MSB	Page dependent, Absolute granule position
14—17	0xXX LSB — 0xXX MSB	Page dependent, Bitstream serial number
18—21	0xXX LSB — 0xXX MSB	Page sequence number
22—25	0xXX LSB — 0xXX MSB	32-bit CRC checksum
26	0x00—0xFF (0—255)	Number of segments
27	0x00—0xFF (0—255)	Segment table
...	...	
n	0x00—0xFF (0—255)	(n = Number of segments + 26)

context in the bitstream).

4. Absolute granule position:
   This is an 8-byte field containing positional information. This is packed in the same way as the rest of Ogg data is packed. The 'position' data specifies a 'sample' number. The position specified is the total samples encoded after including all packets finished on this page (packets begun on this page but continuing on to the next page do not count). For audio stream, it may contain the total number of PCM samples encoded after including all frames finished on this page. This is a hint for the decoder and gives it some timing and position information. Its meaning is dependent on the codec for that logical bitstream and specified in a specific media mapping. A special value of -1 in two's complement indicates that no packets finish on this page.

5. Bitstream serial number:
   This is a 4-byte field containing the unique serial number by which the logical bitstream is identified. Ogg allows for separate logical bitstreams to be mixed at page granularity in a physical bitstream. The most common case would be sequential arrangement, but it is possible to interleave pages for two separate bitstreams to be decoded concurrently. The serial number is the means by which pages of a physical stream are associated with a particular logical stream. Each logical stream must have a unique serial number within a physical stream.

6. Page sequence number:
   This is a 4-byte field containing the sequence number of the page so the decoder can identify page loss. This sequence number is increasing on each logical bitstream separately. Thus, it is a page counter which lets us know if a page is lost (useful where packets span page boundaries).

7. CRC checksum:
   A 4-byte field containing a 32-bit CRC checksum of the page (including header with zero CRC field and page content).The generator polynomial is 0x04C11DB7.

8. Number of page segments:
   It is a single byte giving the number of segment entries encoded in the segment table. The maximum number of 255 segments (255 bytes each) sets the maximum possible physical page size at 65307 bytes or just under 64 kB.

9. Segment table:
   It consists of number-of-page-segments bytes containing the lacing values of all segments in this page. Each byte contains one lacing value. The lacing

values for each packet segment physically appearing in this page are listed in contiguous order.

Total page size is calculated directly from the known header size and lacing values in the segment table. Packet data segments follow immediately after the header. The total header size in bytes is given by:
$header\_size = number\_page\_segments + 27$
The total page size in bytes is given by:
$page\_size = header\_size + sum\_of\_lacing\_values$

2.1.5 Ogg decoding

This decoding is based around the Ogg synchronization layer. We read data into the synchronization layer, submit the data to the stream, and output raw packets to the decoder. The bitstream is captured (or recaptured) by looking for the beginning of a page, specifically the capture pattern. Once the capture pattern is found, the decoder verifies page sync and integrity by computing and comparing the checksum. At that point, the decoder can extract the packets themselves. Decoding through the Ogg layer follows a specific logical sequence.

The flowchart for Ogg decoding is shown in Figure 2.5. The steps to be followed are as follows:

- Expose a buffer from the synchronization layer in order to read data.
- Read data into the buffer, using `fread()` or a similar function.
- Call a function to tell the synchronization layer how many bytes you wrote into the buffer.
- Write out the data by outputting a page from the synchronization layer.
- Submit the completed page to the streaming layer (using appropriate function).
- Output a packet of data to the codec-specific decoding engine.

2.2 Vorbis specifications

2.2.1 Overview

Vorbis is a general purpose perceptual audio codec having these features.

- Allows maximum encoder flexibility.

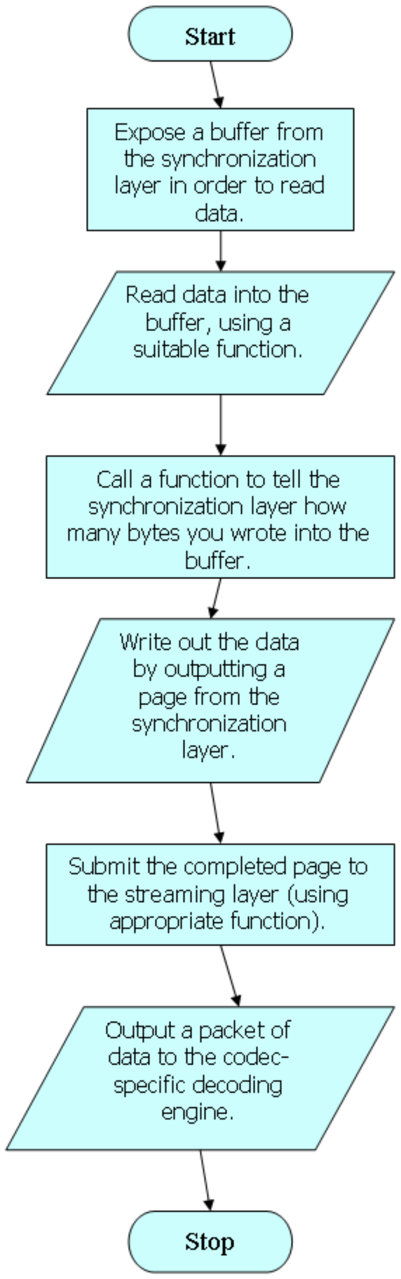

Start
Expose a buffer from the synchronization layer in order to read data.
Read data into the buffer, using a suitable function.
Call a function to tell the synchronization layer how many bytes you wrote into the buffer.
Write out the data by outputting a page from the synchronization layer.
Submit the completed page to the streaming layer (using appropriate function).
Output a packet of data to the codec-specific decoding engine.
Stop

Figure 2.5: Flowchat for Ogg decoding

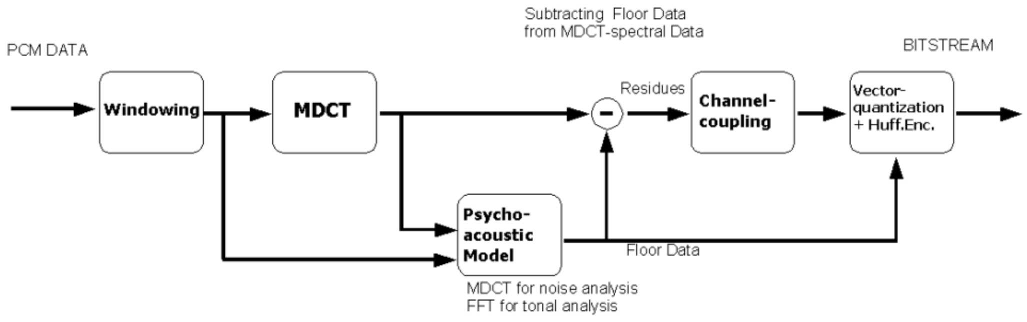


Figure 2.6: Vorbis encoding functional diagram

- Can be scaled over exceptionally wide range of bitrates.
- Is of the same league as MPEG-2 and MPC.
- Vorbis I — a forward adaptive monolithic transform codec based on Modified Discrete Cosine Transform.
- Vorbis II — a codec structured to allow addition hybrid wavelet filter banks to offer better transient response and reproduction using a transform better suited to localized time events.

2.2.2 Vorbis encoding

Vorbis encoder involves the following stages. The functional diagram of the encoder is shown in Figure 2.6

1. Analysis Stage:
   - Block Switching
   - MDCT
   - Psychoacoustic model
2. Coding:
   - Floor generation
   - Channel coupling/residue generation
   - Encoding (Vector Quantization)
3. Streaming
   - Pack to Ogg stream

Analysis

Analysis begins by separating an input audio stream into individual, overlapping short-time segments of audio data. These segments are then transformed into an alternate representation, seeking to represent the original signal in a more efficient form that codes into a smaller number of bytes. The analysis and transformation stage is the most complex element of producing a Vorbis bitstream.

Coding

Coding and decoding converts the transform-domain representation of the original audio produced by analysis to and from a bitwise packed raw data packet. Coding and decoding consist of two logically orthogonal concepts, back-end coding and bit packing. Back-end coding uses a probability model to represent the raw numbers of the audio representation in as few physical bits as possible; familiar examples of back-end coding include Huffman coding and Vector Quantization. Bit packing arranges the variable sized words of the back-end coding into a vector of octets without wasting space. The octets produced by coding a single short-time audio segment constitute one raw Vorbis packet.

2.2.3 Vorbis decoding

Details of the vorbis decoding procedure are given in [2]. What follows is a summary of the description in [2].

Decode setup

Before decoding can begin, a decoder must initialize using the bit stream headers matching the stream to be decoded. Vorbis uses three header packets; all are required, in-order, by this specification. Once set up, decode may begin at any audio packet belonging to the Vorbis stream. In Vorbis I, all packets after the three initial headers are audio packets. The header packets are, in the order, as follows.
Identification header is used to identify bit stream, Vorbis version, sample rate and number of channels.
Comment header includes user text comments ('tags') and a vendor string for the application/library that produced the bit stream.
Setup header includes extensive codec setup information as well as the complete VQ and Huffman codebooks needed for decode.

Any packet that follows these packets will be an audio packet. No audio packet can arrive before all of these packets have arrived. Once the packet type is verified as audio it is passed to the decode engine. The procedure for vorbis decode

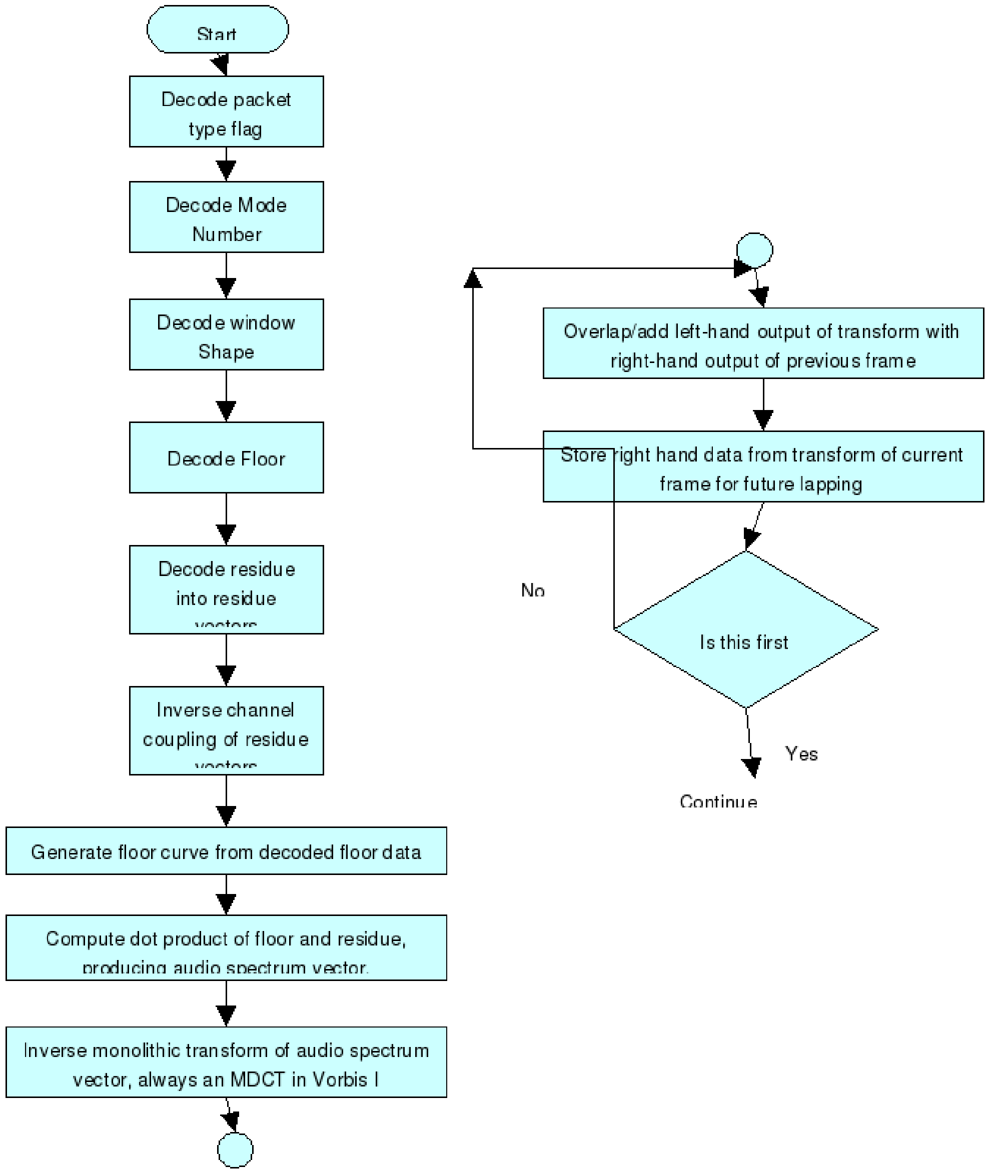

Start
Decode packet type flag
Decode Mode Number
Decode window Shape
Decode Floor
Decode residue into residue vectors
Inverse channel coupling of residue vectors
Generate floor curve from decoded floor data
Compute dot product of floor and residue, producing audio spectrum vector
Inverse monolithic transform of audio spectrum vector, always an MDCT in Vorbis I
Overlap/add left-hand output of transform with right-hand output of previous frame
Store right hand data from transform of current frame for future lapping
Is this first
No
Yes
Continue

Figure 2.7: Vorbis decode flowchart

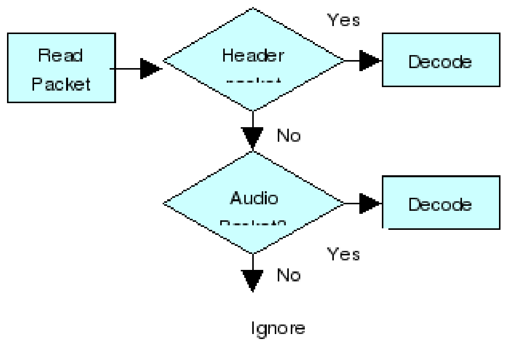


Figure 2.8: Vorbis packet decode flowchart

is shown below. Vorbis uses an MDCT spectrum and psychoacoustical masking techniques to compress audio data. The spectrum is coded as two different entities—floor and residue.

Floor is the envelope of the spectrum. It is the rough variation without the fine local amplitude changes. Residue is the remainder of the spectrum when it is divided by the floor. So floor gives the gross variation and residue gives the fine variations in the spectrum. They are encoded differently.

Decode procedure

Decoding and synthesis procedure for all audio packets is essentially same and is shown in the Figure 2.7

We can take advantage of the symmetry of MDCT and store right hand transform data of a partial 50% inter frame buffer space savings and then we can complete the transform later before overlap/add with the next frame.

2.2.4 Packet decode process

Vorbis I has 4 types of packets—first three types are header packets and are as described above and 4th type is audio packet. All other types are marked as reserved and must be ignored. The flowchart for packet decode is shown in Figure 2.8. The decode procedure for an audio packet will now be discussed.

Mode decode

Vorbis allows multiple, numbered packet 'modes'. Each Vorbis frame is coded according to a master mode. A bitstream may use one or many modes. The mode mechanism is used to encode a frame according to one of multiple possible methods with the intention of choosing a method best suited to that frame. Different modes are, e.g. how frame size is changed from frame to frame. The mode number of a frame serves as a top level configuration switch for all other specific aspects of

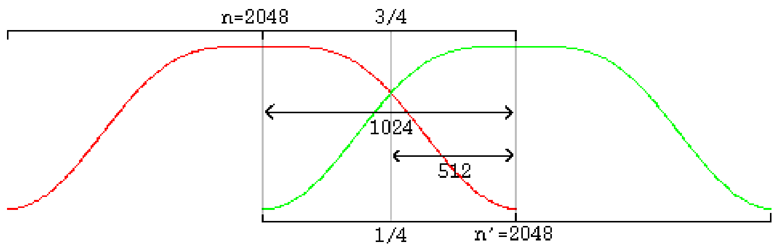


Figure 2.9: Equally sized window overlap

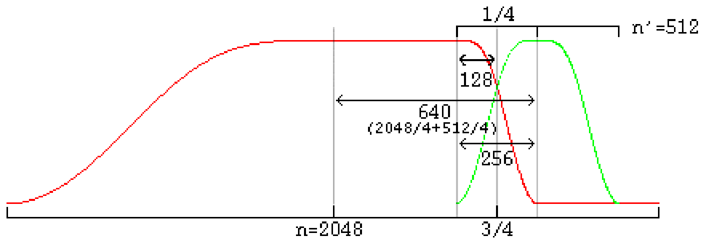


Figure 2.10: Unequally sized window overlap

frame decode. A mode configuration consists of a frame size setting, window type (always 0, the Vorbis window, in Vorbis I), transform type (always type 0, the MDCT, in Vorbis I) and a mapping number. The mapping number specifies which mapping configuration instance to use for low-level packet decode and synthesis.

Window shape decode

Vorbis uses an overlapping transform, namely the MDCT, to blend one frame into the next, avoiding most inter-frame block boundary artifacts. The MDCT output of one frame is windowed according to MDCT requirements overlapped 50% with the output of the previous frame and added. The window shape assures seamless reconstruction as shown in Figures 2.9 and 2.10. Vorbis I uses a Kaiser-Bessel-Derived (KBD) window.

$$y = sin(\frac{\pi}{2} sin^2(\frac{x+\frac{1}{2}}{n}\pi))$$

where n is the window size and x ranges from 0, 1, ..., $n-1$.
In unequal-sized overlap, window shape must be modified for seamless lapping. Vorbis also codes two flag bits to specify pre and post-window shape.

Floor decode

Vorbis encodes a spectral 'floor' vector for each PCM channel. This vector is a low-resolution representation of the audio spectrum for the given channel in the current frame. The values coded/decoded by a floor are both compactly formatted and make use of entropy coding to save space. For this reason, a floor configuration generally refers to multiple codebooks in the codebook component list. Entropy coding is thus provided as an abstraction, and each floor instance may choose from any and all available codebooks when coding/decoding. Each floor is encoded/decoded in channel order, however each floor belongs to a 'sub map' that specifies which floor configuration to use. All floors are decoded before residue decode begins.

Residue decode

The spectral residue is the fine structure of the audio spectrum once the floor curve has been subtracted out. In simplest terms, it is coded in the bitstream using cascaded (multi-pass) vector quantization according to one of three specific packing/coding algorithms numbered 0 through 2. Although the number of residue vectors equals the number of channels, channel coupling may mean that the raw residue vectors extracted during decode do not map directly to specific channels. When channel coupling is in use, some vectors will correspond to coupled magnitude or angle. The coupling relationships are described in the codec setup and may differ from frame to frame, due to different mode numbers. Vorbis codes residue vectors in groups by sub map; the coding is done in sub map order from sub map 0 through $n - 1$. This differs from floors which are coded using a configuration provided by sub map number, but are coded individually in channel order.

Inverse channel coupling

Vorbis coupling applies to pairs of residue vectors at a time; decoupling is done in-place a pair at a time in the order and using the vectors specified in the current mapping configuration. The decoupling operation is the same for all pairs, converting square polar representation (where one vector is magnitude and the second angle) back to Cartesian representation. After decoupling, in order, each pair of vectors on the coupling list, the resulting residue vectors represent the fine spectral detail of each output channel.

Generate floor curve

The decoder generates floor curve at appropriate times. It is generated when floor data is decoded from the raw packet, or after inverse coupling. Both floor 0 and floor 1 generate a linear-range, linear-domain output vector to be multiplied (dot product) by the linear-range, linear-domain spectral residue.

Compute floor residue dot product

The decoder multiplies the floor curve and residue vectors element by element, producing the finished audio spectrum of each channel.

Inverse monolithic transform

The audio spectrum is converted back into time domain PCM audio via an inverse Modified Discrete Cosine Transform (MDCT). Note that the PCM produced directly from the MDCT is not yet finished audio; it must be lapped with surrounding frames using an appropriate window (such as the Vorbis window) before the MDCT can be considered orthogonal.

Overlap add data

Windowed MDCT output is overlapped and added with the right hand data of the previous window such that the $3/4^{th}$ point of the previous window is aligned with the $1/4^{th}$ point of the current window (as illustrated in the window overlap diagram). At this point, the audio data between the center of the previous frame and the center of the current frame is now finished and ready to be returned.

Cache right hand data

The decoder must cache the right hand portion of the current frame to be lapped with the left hand portion of the next frame.

Return finished audio data

The overlapped portion produced from overlapping the previous and current frame data is finished data to be returned by the decoder. This data spans from the center of the previous window to the center of the current window. In the case of same-sized windows, the amount of data to return is one-half block consisting of and only of the overlapped portions. When overlapping a short and long window, much of the returned range is not actually overlap. This does not damage transform orthogonality. The amount of data to be returned is

$$(previous\_window\_blocksize)/4 + (current\_window\_blocksize)/4$$

from the center of the previous window to the center of the current window.

Data is not returned from the first frame; it must be used to 'prime' the decode engine. The encoder accounts for this priming when calculating PCM offsets; after the first frame, the proper PCM output offset is 0.

Chapter 3

ADSP-21364 processor and EZ-KIT LITE specifications

3.1 EZ-KIT LITE evaluation board specifications

3.1.1 Board overview

The EZ-KIT LITE has been designed to demonstrate the capabilities of the ADSP-21364 processor. The processor core is powered at 1.2V, and the I/O is powered at 3.3V. Two 0Ω resistors give access to the processor's power planes and allow measuring the power consumption of the processor. The R79 resistor provides access to the I/O voltage of the processor, and the R80 resistor provides access to the core voltage plane of the processor. The schematics of the evaluation board are shown in Figure 3.1. The detailed description of the board is given in [3]. A brief outline of important features is given here.

- Analog Devices ADSP-21364 processor
  - 136-pin BGA package
  - 300 MHz Core Clock Speed
- Synchronous Random Access Memory (SRAM)
  - 512 Kbit x 8-bit
- Flash Memory
  - 1M x 8-bit
- Serial Peripheral Interconnect (SPI) Flash Memory
  - 512 Kbit

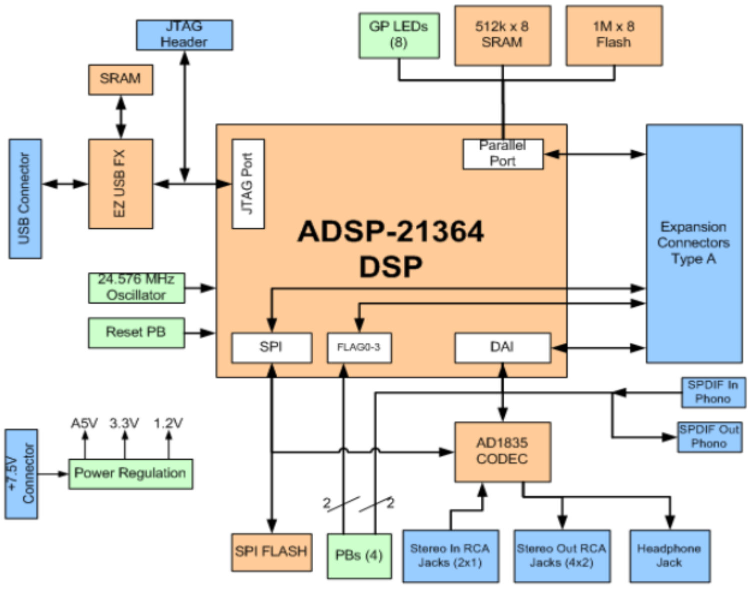


Figure 3.1: EZ-KIT LITE board schematics

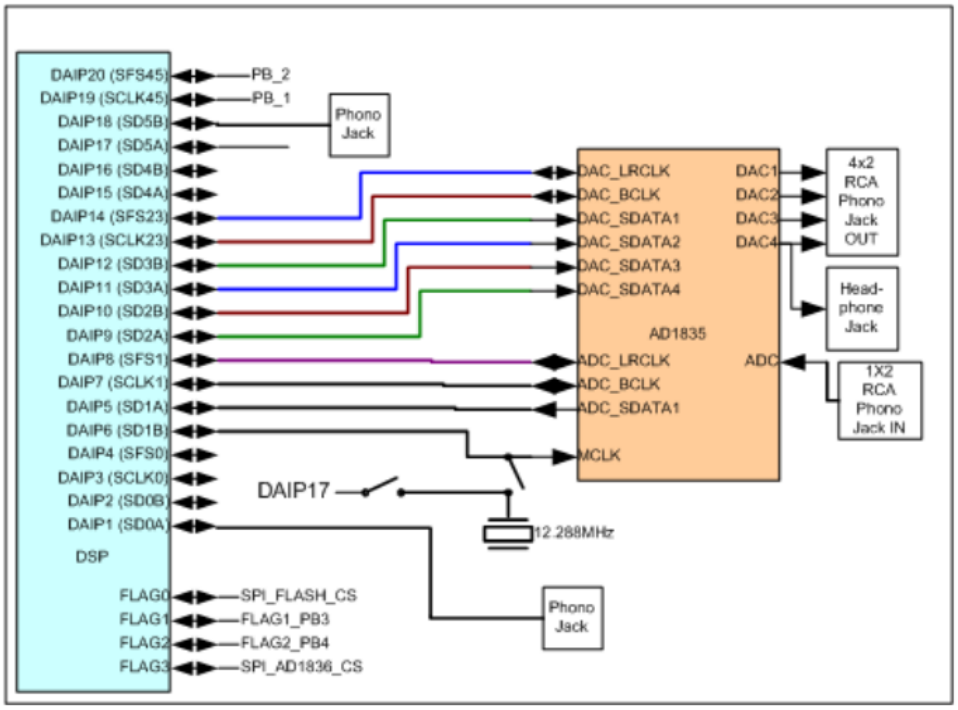


Figure 3.2: Digital Applications Interface (DAI) schematics

- Analog Audio Interface
    - AD1835A codec
    - 4x2 RCA phono jack for 4 channels of stereo output
    - 2x1 RCA phono jack for 1 channel of stereo input
    - Headphone jack for 1 channel stereo output
- Digital Audio Interface
    - RCA phono jack output
    - RCA phono jack input
- LEDs
    - 12 LEDs: 1 power (green), 1 board reset (red), 1 USB reset (red), 1 USB monitor (amber), and 8 general purpose (amber)
- Push Buttons
    - 5 push buttons: 1 reset, 2 connected to DAI, 2 connected to the FLAG pins of the processor
- Expansion Interface (Type A)
    - Parallel Port, FLAGs, DAI, SPI
- Other features
    - JTAG ICE 14-pin header
    - 0Ω resistors for processor current measurement
    - SPI header
    - DAI header

The EZ-KIT LITE board has a total of 1 MB of parallel flash memory and 512 Kbit of SPI flash memory. The flash memories can store user-specific boot code, allowing the board to run as a stand-alone unit. The board also has 512 kB of SRAM, which can be used at runtime. The DAI of the processor connects to the AD1835A audio codec and two connectors, which allow SPDIF input and output. The interface facilitates development of digital and analog audio signal-processing applications. Additionally, the EZ-KIT LITE board provides access to all of the processor's peripheral ports. Access is provided in the form of a three-connector expansion interface. We now proceed to explain in brief about the peripherals we are using in this project.

3.1.2 External memory

The EZ-KIT LITE contains three types of memory—parallel flash (1 MB), SPI flash (512 Kbit) and SRAM (512 Kbit). The parallel flash memory and the SRAM connect to the parallel port of the processor. The parallel port is a multiplexed address and data port. The port can connect to 8-bit and 16-bit memory devices. When configuring the parallel port, keep in mind that the memory devices on the board are 8 bits wide. To access the SRAM and flash memories, set up a Parallel Port DMA. The SPI flash memory connects to the SPI port of the processor and uses `FLAG0` as a chip select. In order for `FLAG0` to behave as a chip select, clear the `PPFLG` bit in the `SYSCLT` register.

3.1.3 DAI interface

The pins of the Digital Applications Interface (DAI) connect to the Signal Routing Unit (SRU). The SRU is a flexible routing system, providing a large system of signal flows within the processor. In general, the SRU allows to route the DAI pins to different internal peripherals in various combinations.

The DAI pins are connected to the AD1835A audio codec, a 26-pin header, 2 RCA connectors, the audio oscillator output, and two push buttons. Figure 3.2 illustrates the EZ-KIT LITE's connections to the DAI.

3.1.4 Analog audio

The AD1835A is a high-performance, single-chip codec featuring four stereo digital-to-analog converters (DAC) for audio output and one stereo analog-to-digital converters (ADC) for audio input. The codec can input and output data with a sample rate of up to 96 kHz on all channels. A 192 kHz sample rate can be used with the one of the DAC channels. The processor is interfaced with the AD1835A via the DAI port. The DAI interface pins can be configured to transfer serial data from the AD1835A codec in either Time Division Multiplexed (TDM) or Two-Wire Interface mode. The master input clock (MCLK) for the AD1835A can be generated by the on-board 12.288 MHz oscillator or can be supplied by one of the DAI pins of the processor. Using one of the pins to generate the MCLK, as opposed to the on-board oscillator, allows synchronization of multiple devices in the system. This is done on the EZ-KIT LITE when data is coming from the SPDIF receiver and being output through the audio codec. The SPDIF MCLK is routed to the AD1835A MCLK in the processor's SRU. It is also necessary to disable the on-board audio oscillator from driving the audio codec and the processor's input pin. The AD1835A codec can be configured as a master or as a slave, depending on DIP switch settings. In master mode, the AD1835A drives the serial port clock and frame sync signals to the processor. In slave mode, the processor must generate and drive all the serial port clock and frame sync signals.

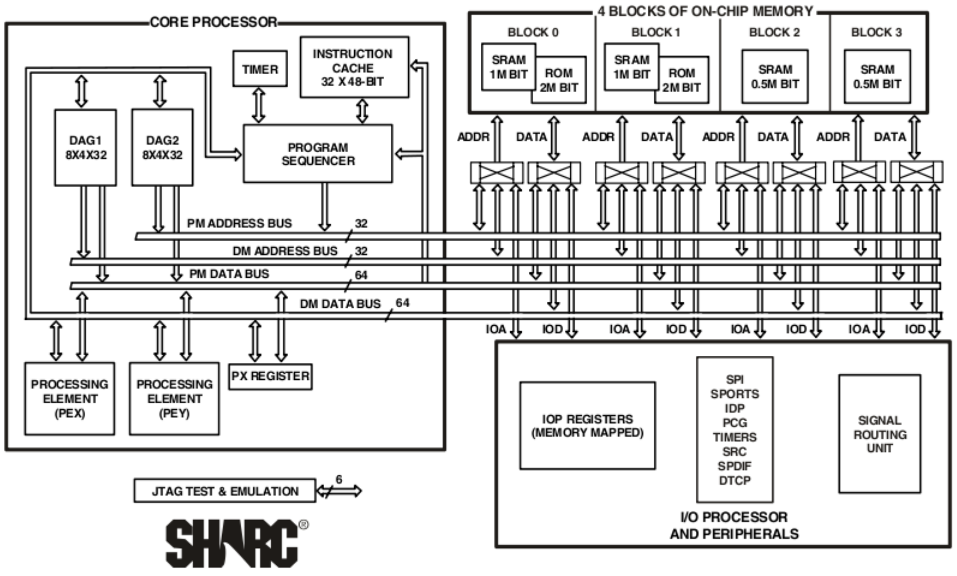


Figure 3.3: SHARC ADSP-21364 architecture

The AD1835A audio codec's internal configuration registers are configured using the SPI port of the processor. The `FLAG3` register is used as the select for the device. The RCA connector (J4) is used to input analog audio.

3.2 SHARC ADSP-21364 processor

3.2.1 Overview

The ADSP-21364 SHARC processor is a member of the SIMD SHARC family of DSPs that feature Analog Devices' Super Harvard Architecture. The ADSP-21364 is a 32-bit/40-bit floating-point processor optimized for professional audio applications with a large on-chip SRAM, multiple internal buses to eliminate I/O bottlenecks, and an innovative digital audio interface (DAI). The ADSP-21364 continues SHARC's industry-leading standards of integration for DSPs, combining a high performance 32-bit DSP core with integrated, on-chip system features.

ADSP-21364 uses two computational units to deliver a significant performance increase over previous SHARC processors on a range of signal processing algorithms. Fabricated in a state-of the-art, high speed, CMOS process, the ADSP-21364 processor achieves an instruction cycle time of 3.0 ns at 333 MHz. With its SIMD computational hardware, the ADSP-21364 can perform 2 GFLOPS running at 333 MHz.

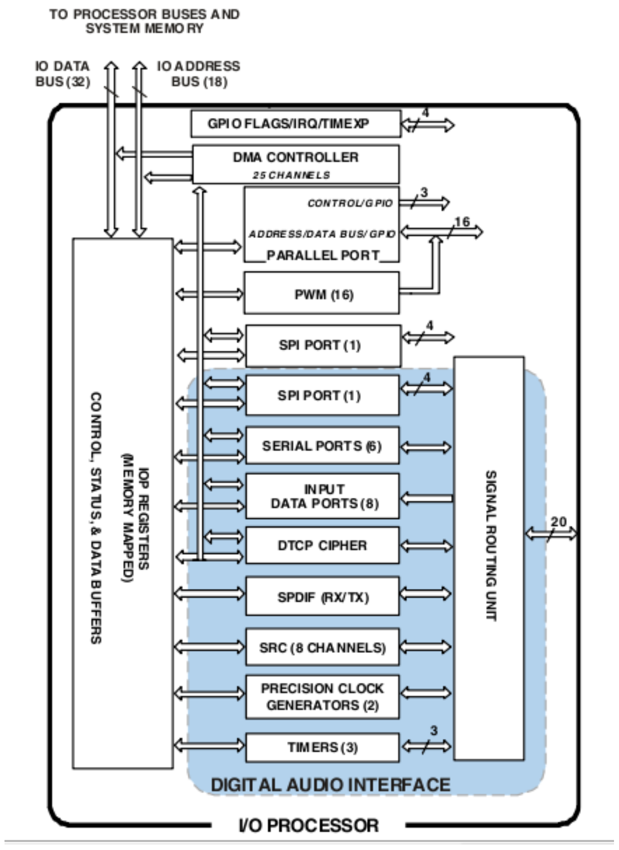


Figure 3.4: Internal I/O processor architecture

3.2.2 Architectural features

The block diagram of the ADSP-21364, from [4] and [5], shown in Figure 3.3, illustrates the following architectural features:

- Two processing elements, each of which comprises an ALU, multiplier, shifter and data register file
- Data address generators (DAG1, DAG2)
- Program sequencer with instruction cache
- PM (Program Memory) and DM (Data Memory) buses capable of supporting four 32-bit data transfers between memory and the core at every core processor cycle
- Three programmable interval timers with PWM generation, PWM capture/pulse width measurement, and external event counter capabilities
- On-chip SRAM (3M bit)
- On-chip mask-programmable ROM (4M bit)
- 8-bit or 16-bit parallel port that supports interfaces to off chip memory peripherals
- JTAG test access port

The block diagram of the ADSP-21364 shown in Figure 3.4, illustrates the following I/O processor and peripheral features:

- DMA controller
- Six full duplex serial ports
- Two SPI-compatible interface ports — primary on dedicated pins and secondary on DAI pins
- Digital audio interface that includes two precision clock generators (PCG), an input data port (IDP), an S/PDIF receiver/transmitter, eight channels asynchronous sample rate converters, six serial ports, eight serial interfaces, a 20- bit parallel input port, 10 interrupts, six flag outputs, six flag inputs, three timers, and a flexible signal routing unit (SRU)

3.2.3 ADSP-21364 memory and I/O interface features

The ADSP-21364 adds several architectural features to the SIMD SHARC family core. We hereby discuss the ones bearing relevance to this project.

On-chip memory

The ADSP-21364 contains three megabits of internal SRAM. Each block can be configured for different combinations of code and data storage. Each memory block supports single-cycle, independent accesses by the core processor and I/O processor. The ADSP-21364 memory architecture, in combination with its separate on-chip buses, allows two data transfers from the core and one from the I/O processor, in a single cycle. The ADSP-21364's SRAM can be configured as a maximum of 96K words of 32-bit data, 192K words of 16-bit data, 64K words of 48-bit instructions (or 40-bit data), or combinations of different word sizes up to three megabits. All of the memory can be accessed as 16-bit, 32-bit, 48-bit, or 64-bit words. A 16-bit floating- point storage format is supported that effectively doubles the amount of data that may be stored on-chip. Conversion between the 32-bit floating-point and 16-bit floating-point formats is performed in a single instruction. While each memory block can store combinations of code and data, accesses are most efficient when one block stores data using the DM bus for transfers, and the other block stores instructions and data using the PM bus for transfers. Using the DM bus and PM buses, with one dedicated to each memory block assures single-cycle execution with two data transfers. In this case, the instruction must be available in the cache.

DMA controller

The ADSP-21364's on-chip DMA controller allows data transfers without processor intervention. The DMA controller operates independently and invisibly to the processor core, allowing DMA operations to occur while the core is simultaneously executing its program instructions. DMA transfers can occur between the ADSP-21364's internal memory and its serial ports, the SPI-compatible (serial peripheral interface) ports, the IDP (input data port), the parallel data acquisition port (PDAP), or the parallel port. Twenty-five channels of DMA are available on the ADSP-21364?two for the SPI interface, two for memory- to-memory transfers, 12 via the serial ports, eight via the input data port, and one via the processor's parallel port. Programs can be downloaded to the ADSP-21364 using DMA transfers. Other DMA features include interrupt generation upon completion of DMA transfers, and DMA chaining for automatic linked DMA transfers.

General procedure for configuring a DMA

To configure the ADSP-2136x processor to use DMA, the following general procedure is used.

1. Determine which DMA options you want to use
    - IOP/Core interaction method—interrupt driven or status driven (polling)

- DMA transfer method—chained or non chained
- Channel priority scheme—fixed or rotating

2. Determine how you want the DMA to operate
    - Determine and set up the data's source and/or destination addresses (INDEX)
    - Set up the word COUNT (data buffer size)
    - Configure the MODIFY values (step size)
3. Configure the peripheral(s)
    - Serial ports (SPORTs)
    - Parallel port (PP)
    - Serial peripheral interface ports (SPI)
    - Input data port (IDP)
4. Enable DMA
    - Set the applicable bits in the appropriate registers. For example, the `PPDEN` bit in `PPCTL` register for the parallel port or the `SDENx` bit in `SPCTLx` register for the serial port.

Serial ports (SPORTs)

The ADSP-21364 features six synchronous serial ports that provide an inexpensive interface to a wide variety of digital and mixed-signal peripheral devices such as Analog Devices' AD183x family of audio codecs, ADCs, and DACs. The serial ports are made up of two data lines, a clock, and frame sync. The data lines can be programmed to either transmit or receive and each data line has a dedicated DMA channel. Serial ports are enabled via 12 programmable and simultaneous receive or transmit pins that support up to 24 transmit or 24 receive channels of audio data when all six SPORTS are enabled, or six full duplex TDM streams of 128 channels per frame. The serial ports operate at a maximum data rate of 50M bits/s. Serial port data can be automatically transferred to and from on-chip memory via dedicated DMA channels. Each of the serial ports can work in conjunction with another serial port to provide TDM support. One SPORT provides two transmit signals while the other SPORT provides the two receive signals. The frame sync and clock are shared. Serial ports operate in four modes

- Standard DSP serial mode
- Multi-channel (TDM) mode

- I^2S mode
- Left-justified sample pair mode

Parallel port

The parallel port provides interfaces to SRAM, flash memory and other peripheral devices. The multiplexed address and data pins (`AD15-0`) can access 8-bit devices with up to 24 bits of address, or 16-bit devices with up to 16 bits of address. In mode, 8- or 16- bit, the maximum data transfer rate is 55M bytes/sec. DMA transfers are used to move data to and from internal memory. Access to the core is also facilitated through the parallel port register read/write functions. The `RD`, `WR`, and `ALE` (address latch enable) pins are the control pins for the parallel port.

3.3 Development tools

The processor is supported with a complete set of CrossCore software and hardware development tools, including Analog Devices emulators and the VisualDSP++ development environment. The same emulator hardware that supports other SHARC Analog Devices products also fully emulates the ADSP-21364 processor family.

The VisualDSP++ project management environment lets programmers develop and debug an application. This environment includes an easy-to-use assembler that is based on an algebraic syntax, an archiver (librarian/library builder), a linker, a loader, a cycle-accurate instruction-level simulator, a C/C++ compiler, and a C/C++ runtime library that includes DSP and mathematical functions. A key point for these tools is C/C++ code efficiency. The compiler has been developed for efficient translation of C/C++ code to SHARC processor assembly. The SHARC processor has architectural features that improve the efficiency of compiled C/C++ code.

Debugging both C/C++ and assembly programs with the VisualDSP++ debugger, programmers can:

- View mixed C/C++ and assembly code (interleaved source and object information)
- Insert breakpoints
- Set conditional breakpoints on registers, memory, and stacks
- Trace instruction execution
- Perform linear or statistical profiling of program execution

- Fill, dump, and graphically plot the contents of memory
- Perform source level debugging
- Create custom debugger windows

The VisualDSP++ Integrated Development Environment (IDE) lets programmers define and manage software development. Its dialog boxes and property pages let programmers configure and manage all development tools, including Color Syntax Highlighting in the VisualDSP++ editor.

These capabilities permit programmers to control how the development tools process inputs and generate outputs. It allows for a one-to-one correspondence with the tool's command-line switches.

Analog Devices emulators use the IEEE 1149.1 JTAG test access port of the processor to monitor and control the target board processor during emulation. The emulator provides full speed emulation, allowing inspection and modification of memory, registers, and processor stacks. Non-intrusive in-circuit emulation is assured by the use of the processor's JTAG interface; the emulator does not affect target system loading or timing. In addition to the software and hardware development tools available from Analog Devices, third parties provide a wide range of tools supporting the SHARC processor family. Hardware tools include the ADSP-21364 EZ-KIT LITE standalone evaluation/development cards. Third party software tools include DSP libraries, real-time operating systems, and block diagram design tools.

Chapter 4

Implementation

4.1 Approaches to problem solving

Implementation involves the following steps

- A ramp up of the ADSP processor architecture, including implementation of simple codes to gain familiarity with the processor and with VDSP++ IDE.
- Understanding the entire flow of the decoder code with Ogg Vorbis source code as the reference.
- Apply the memory optimization schemes to the floating point code and fit it in the processor memory for the VDSP ADSP 21364 emulator.
- Profile the program with file I/O and optimize for speed.
- To understand interfacing with other components on the EZ-KIT, such as flash and output ports.
- Implementation of buffer management for efficient I/O handling. Removal of file I/O and implementation of DMA for on chip memory - SPORT and on chip memory - flash data transfers.

The tools used for implementation include the following:

- Microsoft Visual C++ (PC platform)
- Visual DSP++ by Analog Devices (SHARC processor platform)
- AEDTools Pro (for generating *.ogg file for debugging purposes)
- PCM2WAV.EXE for converting the PCM samples generated by the code into a .wav file so that it can be played on a PC-based media player

4.2 Implementation details

We hereby present the various stages through which the project was implemented.

4.2.1 Stage 1

This stage consisted of procurement of the reference implementation from xiph.org homepage. Also available—apart from the library 'libvorbis'—was an example decoder implementation. The downloaded code was compiled using the MSVC++ environment. The Tremor code was also looked at, but it was not of great consequence because it was a fixed point implementation. On running the decoder example in MSVC++ the Ogg Vorbis files were being decoded correctly to PCM files. Compilation in VDSP++ environment for the same decoder example was attempted. But it failed because of insufficient program memory size in the processor. The complier reported that all object files could not be mapped on to the program memory due to insufficient size of `seg_pmco` (program memory segment). This is because though in MSVC++, the entire memory management, including heap and stack sizes is managed entirely by the operating system. In VDSP++, this memory mapping has to be explicitly performed by means of a linker definition file (.ldf). This ldf contains explicit user-defined memory mapping and allocation for stack, heap, program and data memory sections. Using ldf files to define custom memory mappings are described in [6] and [7].

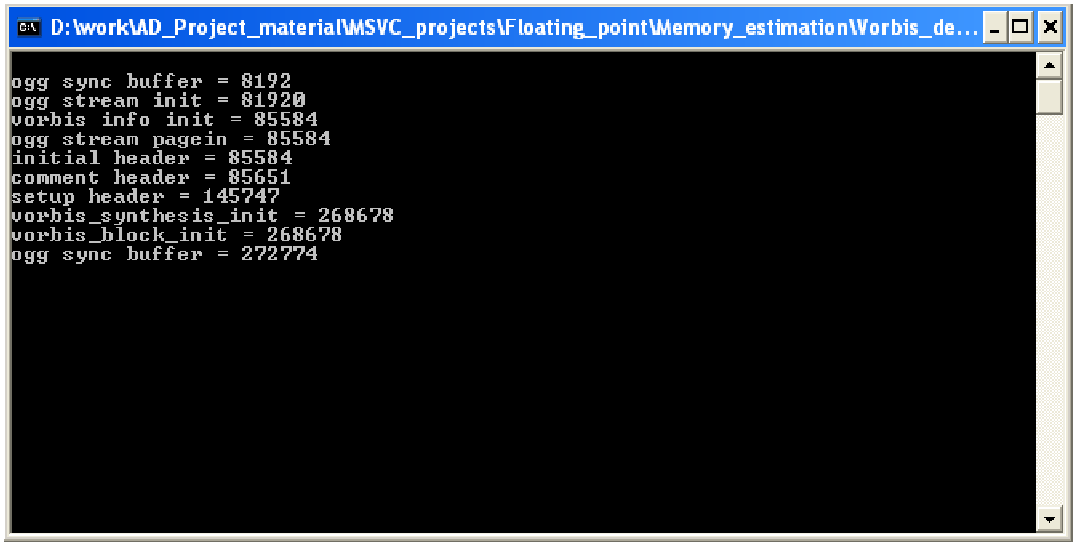
The Ogg Vorbis library that was downloaded from www.xiph.org was a very generic implementation written by many open source forum coders from around the world. They had also included many encoding and performance measuring functions which were not required for the decoder to decode an Ogg file. Hence to optimize `seg_pmco` usage by the code all those redundant functions and source files which were not required for the decoding functionality were removed.

After cleaning up the entire code by removing all the redundant functions, the code was ported on to the processor using VDSP++. It was now able to map the entire into the program memory section successfully.

4.2.2 Stage 2

Although the code was now accomodated in the processor's program memory, it didn't execute properly i.e. it produced many errors at runtime. These errors were determined to be due heap overflow which was due to insufficient heap size on the processor. So the first task was to estimate the heap requirements of the code. Figure 4.1 shows the results of the memory estimation in MSVC++ environment. It turns out to be nearly 2.8 MBits! This was way larger than the allocated 0.5 MBits of heaps.

As the MDCT part of the code generates lookup tables, and vector tables were


D:\work\AD_Project_material\MSVC_projects\Floating_point\Memory_estimation\Vorbis_de...

```
ogg sync buffer = 8192
ogg stream init = 81920
vorbis info init = 85584
ogg stream pagein = 85584
initial header = 85584
comment header = 85651
setup header = 145747
vorbis_synthesis_init = 268678
vorbis_block_init = 268678
ogg sync buffer = 272774
```

Figure 4.1: Heap requirement estimation in MSVC++

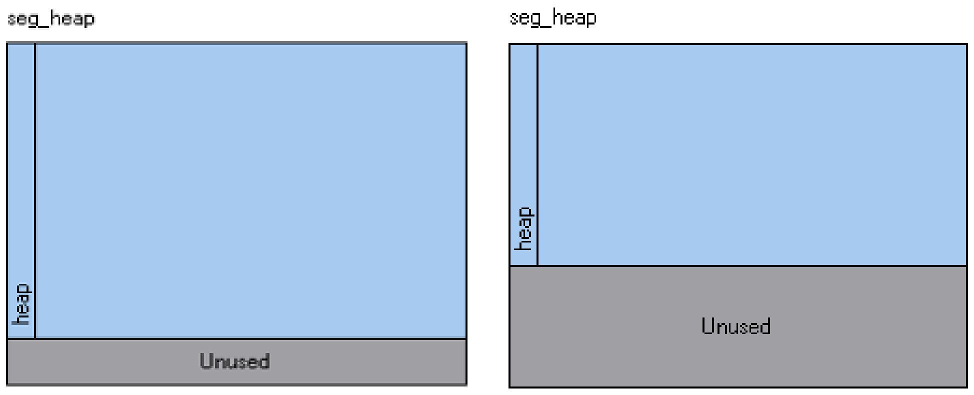


Figure 4.2: Heap reduction after byte packing at synchronization and streaming layers

computed during setup and stored in dynamically allocated memory, heavy usage of the heap was expected. This led to heap overflow, thus causing a malfunction in VDSP++. Increase in available heap size cannot be achieved without resorting to multiple heaps. Each heap can be increased to a maximum of the memory bank size in which the heap is located. If the heap requirement is more than the size of one memory bank, then it requires the usage of 'multiple heaps'. This requires an alteration of the ldf.

Another option would be to expand heap into the external memory. Though the evaluation board contains an external SRAM chip, the processor does not support running the code from the external memory. Also, this memory cannot be accessed directly—a DMA transfer had to be done each time ([5]). Moreover, the memory is 8-bit, and hence some sort of bit packing had to be done. All these issues made the use of external memory for code space unfeasible. Each access to the external memory is through a DMA access involves tedious bit packing, thus rendering it unsuitable for real-time applications.

It was however suggested that alternate heaps could be made 16 bits wide (which would double the available heap size). An attempt was made towards making heap memory 16 bits wide instead of the default 32. This was expected to double the amount of available heap. But according to AD engineers (processor.support@analog.com) this conversion of 32 bit memory to 16 bit memory is not advisable. The program exhibited faulty behavior even though there was no heap overflow. This was expected because compiler generated code assumes 32 bit memory. Hence, the standard library functions like `printf()`, `malloc()` etc. do not work properly.

In the SHARC processor the basic data types `char`, `int`, `float` and `long` are all 32 bits wide. This means that each byte stored on a SHARC processor will occupy the least significant 8 bits of a 32 bit wide memory location. Hence the 24 most significant bits are not being used. This resulted in wastage of memory as bytes were written to word locations. So the solution was to do byte packing in the heap area.

Byte packing was done by creating a data structure called as `DATA_PTR` and defining a few functions for byte read/write functionality in word addressed memory environments—basically a set wrapper functions to obtain byte addressability in word addressed memory. This data structure contained 2 fields—one that pointed to the word address and the other was a modulus value that pointed to the corresponding byte in the word. Byte packing was done at the synchronization layer and a reduction in heap usage of about 4000 memory words was observed. However this reduction was not sufficient enough for meeting the memory requirements of the code. Hence the packing has to be extended to other layers, most essentially the streaming layer. This resulted in an additional reduction in the heap usage by about 6000 words. Figure 4.2 shows the relative heap requirements until partial decode of the setup header.

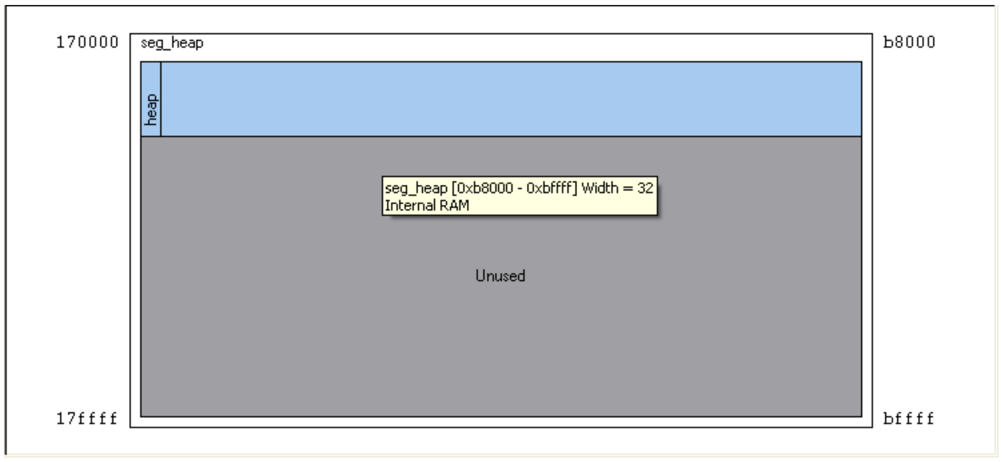


Figure 4.3: Total heap usage without vector table generation

4.2.3 Stage 3

Although byte packing brought about a reduction in heap usage, the heap requirement of the modified reference implementation could not be met. Even with an alternate heap the maximum heap available on the ADSP 21364 processor was about 1.4 Mbits. Hence it was decided to rewrite the entire decoder implementation in full compliance with the Ogg Vorbis I specification document. By rewriting the implementation most minimal usage of heap could be ensured by freeing heap data that would not be required for further processing.

It was observed that the major portion of the heap usage was due to the vector tables that were decoded during header decode. Ogg Vorbis uses a codebook read in vector context. This means that the decoded Huffman codeword is used as an offset into the vector table giving a vector as output. In the reference implementation, the vector tables were built and saved in the heap during codec setup phase. This resulted in excessive heap usage. On disabling the vector table generation altogether (for the purpose of estimating what percentage of heap it occupied), it was found that more than 70% of the heap usage was due to the vector tables. The heap usage status after the codec setup in on disabling vector table genertion is shown in Figure 4.3.

This problem was overcome by building the vector table as and when it was required during audio decode phase. This resulted in a reduction in the memory requirements (to about 0.7 Mbits of heap) of the code enabling it to be implemented on the processor even without the necessity of multiple heaps. This was the crux behind the reduction in heap usage and successful runtime implementation of the decoder on the processor.

The floating point version (the one that we wrote as per Vorbis I specification document) of the Ogg Vorbis decoder with file I/O was implemented in VDSP++. The code was tested for bit-exactness with the reference implementation.

4.2.4 Stage 4

The next and the final stage in the project was to make this implementation decode Ogg Vorbis audio in real time. To decode the audio in real time the rate at which the processor decodes the audio samples has to be faster than the rate at which audio is played. Since the maximum frequency at which the core runs is 333 MHz it was determined that the average time to decode a block of PCM samples of size 2 KB is about 0.004 seconds (was determined by counting the number of clock cycles taken in VDSP++) and since the rate at which these audio samples would be played is 48 KHz the time to play this would be approximately 0.04 seconds. Since the real time constraint was satisfied it was possible to make the current implementation real time.

To make the file I/O implementation real time the following steps were followed. The song was to be burnt to the external flash memory which was of size 1 MB. This song would be then read from the flash memory by the parallel port DMA. This eliminates file input. As the audio samples are decoded they are to be routed to the DAC to be converted into analog audio. To route the audio samples to the DAC serial ports (SPORTs) were used. At the output side a ping pong buffer system was implemented for efficient IO handling.

Flash programming

The first step in this stage involved programming the flash through the parallel port. Flash can be programmed either directly from the 'Program Flash' options under 'Tools' drop down menu in VDSP++ IDE, or it can be programmed by writing C programs using high-level and low-level Flash Programmer routines defined in `adi_am29lv081b.c`. However, the 'Program Flash' option can be used only to program .ldr files. Thus for programming the ogg files onto the flash, a C code which explicitly programmed the flash using the parallel port had to be used.

The main C program had the buffer declarations and functions such as `GetSectorMap()` , `GetFlashInfo()`, `FillData()`, `ReadData()`, `WriteData()` to carry out filling, reading and writing tasks for the flash. The following steps were performed:

- Allocate `AFP_Buffer` using `AllocateAFPBuffer()`
- Get sector map using `GetSectorMap()`
- Setup the device so the DSP can access it using `SetupForFlash()`
- Get flash manufacturer and device codes, title and description using `GetFlashInfo()`
- Erase the entire flash

- Reset the Flash
- Read 8kB from the file into a buffer. Each 32-bit word in this buffer is then split and each byte from this word is masked out and each byte is then stored into a separate word of the ldr_source. Thus we require 8 kB x 4 = 32 kB of memory space in the ldr_source.
- The data from the ldr_source is then burnt onto the flash using `WriteData()`.

File input removal

The next step involved performing a DMA to read the contents of the flash memory into the internal memory using a DMA. The parallel port has a two stage data FIFO for receiving data (RXPP). In the first stage, a 32-bit register (PPSI) provides an interface to the external data pins and packs the 8 or 16-bit input data into 32 bits. Once the 32-bit data is received in PPSI, the data is transferred into the second 32-bit register (RXPP). Once the receive FIFO is full, the chip cannot initiate any more external data transfers. The RXPP register acts as the interface to the core or I/O processor (for DMA).

The `PPTRAN` bit must be zero in order to perform an external read. To use the parallel port for DMA programs, start by setting up values in the DMA parameter registers. The program then writes to the `PPCTL` register to enable `PPDEN` with all of the necessary settings like cycle duration value, transfer direction, and so on. While a parallel port DMA is active, the DMA parameter registers are not writeable. Furthermore, only the `PPEN` and `DMAEN bits` (in the `PPCTL` register) can be changed. If any other bit is changed, the parallel port malfunctions. It is recommended that both the `PPDEN` and `PPEN` bits be set and reset together to ensure proper DMA operation.

Use the following steps to configure the parallel port for a standard DMA read:

- Reset the `PPTRAN` bit in the `PPCTL` register. Ensure that FIFO is empty and the external interface is idle by reading the status of the `PPS` and `PPBS` bits respectively.
- Initialize the `IIPP`, `IMPP`, `ICPP`, `EIPP`, `EMPP` and `ECPP` registers with the appropriate values, keeping `PPDEN` and `PPEN` disabled.
- With all other controls set in the `PPCTL` register, enable the `PPEN` and `PPDEN` bits.

`PPCP` is used only in the case of a chained DMA transfer, which is not so in this implementation. Data was read from the parallel port as and when it was necessary. One problem here was the data stabilizing and polling issue. It took some time for the first word read from the parallel port to be written into the internal memory buffer. Since polling the parallel port to see whether the

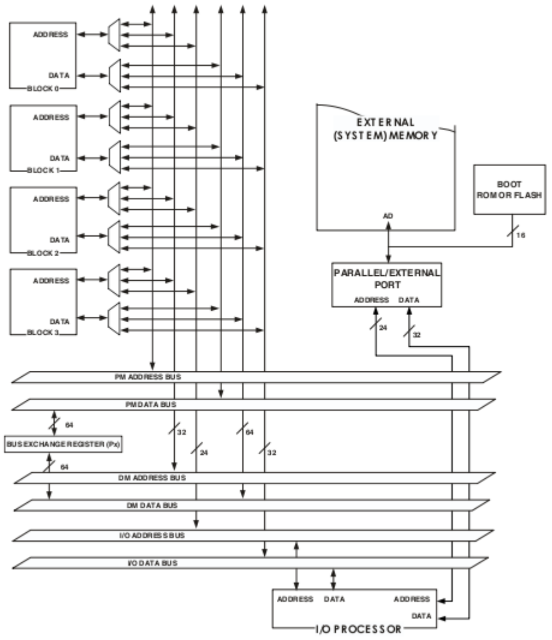


Figure 4.4: Internal memory bus architecture of ADSP-21364

entire block of data had been read would result in unnecessary cycle count it was decided to include sufficient delay between the first word read from parallel port flash and first word read from internal memory buffer (to which the parallel port is writing data fetched from the flash). This delay was brought about by re-scheduling the code. This ensured data stabilization and removed the need for polling. Implementation of parallel port DMA at the input side removed file input.

The ping-pong buffer system

Essentially this involved having 2 buffers A and B. When the SPORT was reading from buffer A the decode engine would write into buffer B and vice versa. If the reading of the audio samples was faster than the writing of the audio samples to the buffer, then real time decoding would not be possible as the decoding is very slow and the SPORT would route out old data repeatedly and it would appear as if the song was being played with bursts of repetition. If the writing of the audio samples to the output buffer, say output buffer A, was faster than the reading of the samples of the other output buffer B, then the decode engine would be asked to 'wait' (i.e execute `NOP`) until the reading of that buffer (buffer B) is complete. Once the reading is complete and SPORT starts readin buffer A the decode engine writes the audio samples to the buffer B. The sizes of the buffers decide the waiting time of the decode engine.

The two output buffers have to be accessed continuously by the SPORT. If these buffers are located in the same heap as the rest of the data, then core access and IOP access to the same memory bank causes bus contention and results in unwanted stalls. To remove these stalls and to use the bus architecture, shown in Figure 4.4, effectively, these output buffers are located in a separate heap in the space above `seg_stak`. As desribed in [8], every memory bank can be accessed using any of the buses. There are three buses totally—DM, PM and IO (for address and data separately). SPORT uses the IO bus for DMA transfer between internal memory and the AD1835A chip. The core uses the DM bus to write into the output buffers or to access some other variables in `seg_heap`. If the data were in the Program Memory then it could also be accessed using PM bus (the essence of Super Harvard Architecture). Thus, there is no bus contention.

File output removal

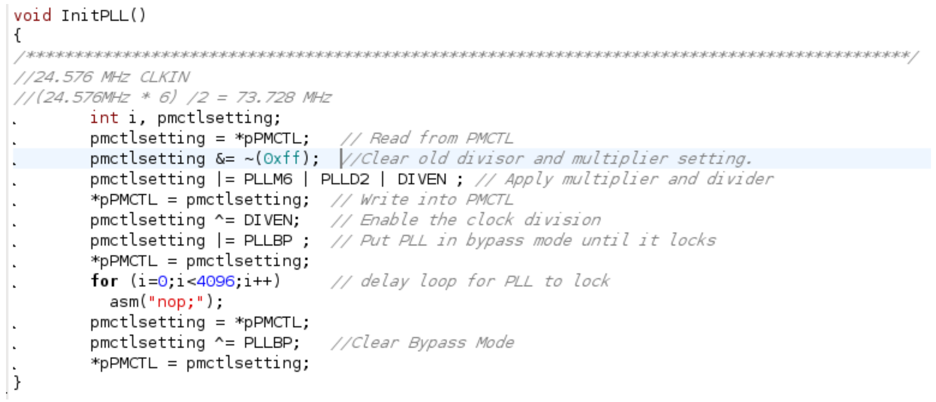
Next step was to initialize 2 buffers for ping pong buffering at the output so as to route the audio samples to DAC-4 in AD1835A. Serial ports (SPORTs) were used for DMA transfers between the internal memory (output ping-pong buffers) and codec chip. SPORT DMA provides a mechanism for receiving or transmitting an entire block of serial data before the interrupt is generated. When SPORT DMA is not enabled, the SPORT generates an interrupt every time it receives or starts to transmit a data word. The processor's on-chip DMA controller handles the DMA transfer, allowing the processor core to continue running until the entire block of data is transmitted or received. Service routines can then operate on the block of data rather than on single words, significantly reducing overhead.

Each SPORT DMA channel has an enable bit (`SDEN_A` and `SDEN_B`) in its `SPCTLx` register. When DMA is disabled for a particular channel, the SPORT generates an interrupt every time it receives a data word or whenever there is a vacancy in the transmit buffer. Each channel also has a DMA chaining enable bit (`SCHEN_A` and `SCHEN_B`) in its `SPCTLx` control register.

To set up a serial port DMA channel, write a set of memory buffer parameters to the SPORT DMA parameter registers. Load the II, IM, and C registers with a starting address for the buffer, an address modifier, and a word count, respectively. The SPORT transmit mode is set to transmit the data in I^2S mode because the DAC receives PCM stereo data in I^2S mode. Once serial port DMA is enabled, the processor's DMA controller automatically transfers the data from one of the output buffers to the DAC 4 (which is connected to the stereo headphone jack) which converts the stereo PCM data into analog stereo audio and can be listened to on the stereo jack by using a stereo headphone.

Estimation of MIPS

The decoder was now running in real-time. The next logical task was to find out the computational intensity of the decoder. To estimate the Million Instructions



```
void InitPLL()
{
/*****************************************************************************************/
//24.576 MHz CLKIN
//(24.576MHz * 6) /2 = 73.728 MHz
        int i, pmctlsetting;
        pmctlsetting = *pPMCTL;    // Read from PMCTL
        pmctlsetting &= ~(0xff);  //Clear old divisor and multiplier setting.
        pmctlsetting |= PLLM6 | PLLD2 | DIVEN ; // Apply multiplier and divider
        *pPMCTL = pmctlsetting;  // Write into PMCTL
        pmctlsetting ^= DIVEN;   // Enable the clock division
        pmctlsetting |= PLLBP ;  // Put PLL in bypass mode until it locks
        *pPMCTL = pmctlsetting;
        for (i=0;i<4096;i++)     // delay loop for PLL to lock
          asm("nop;");
        pmctlsetting = *pPMCTL;
        pmctlsetting ^= PLLBP;   //Clear Bypass Mode
        *pPMCTL = pmctlsetting;
}
```

Figure 4.5: Changing core clock frequency using PMCTL

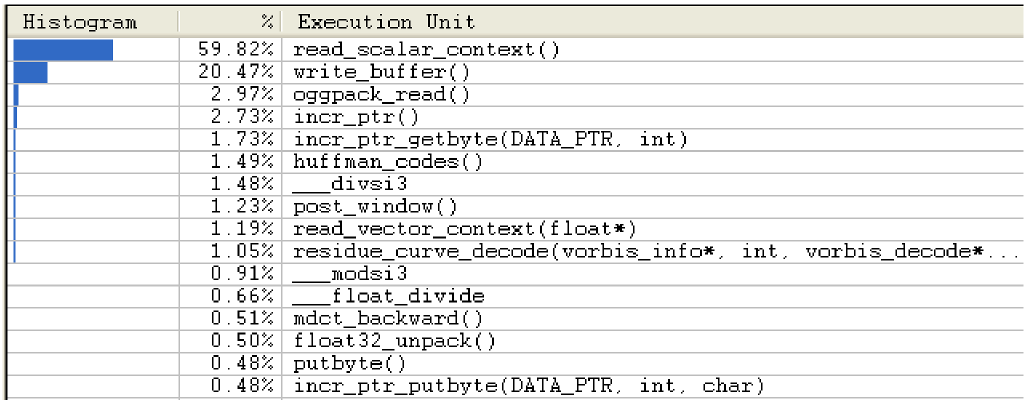
Per Second (MIPS) requirement of the decoder, the DSP's capability of variable operating frequencies was used.

ADSP-21364 has a feature that allows changing core clock and peripheral clock speeds of the processor in software—even during runtime. This uses a memory-mapped register called `PMCTL` which is the Power Management Control register. By using the code snippet shown in Figure 4.5 the operation frequencies can be changed to suit the required computational intensity.

Fields `PPLM` and `PPLD` in `PMCTL` provide the PLL multiplying and dividing factors. The external clock frequency, which is 24.576 MHz, is multiplied by `PPLM` and divided by `PPLD` and the result is the core clock frequency. By consistently reducing the core clock, and finding that point where noise gets introduced into the output audio due to insufficient operating frequency of the DSP, the MIPS requirement of the floating point decoder could be determined. It required about 290 MIPS. Anything lesser would result in bursts of repetition in the output audio. Bursts of repetition occur because the SPORT is driven by an external clock from the ADC in AD1835A and not from the core clock. Thus decoding happens at a lower rate but playback happens at the same rate. So frames of audio are played back repeatedly.

Statistical profiling

The decoder was now running in real-time on the DSP. But on reduction of the operating frequency of the DSP below 290 MHz, the audio output would be noisy. So the code required 290 MIPS for noiseless decoding. The code was profiled using the statistical profiling tool in VDSP++, the result of which is shown in Figure 4.6. This was done to identify the critical section of the code. It was apparent that the function `read_scalar_context()` consumed majority of the computing time. Measures had to be taken to optimize the critical section of the code



Histogram	%	Execution Unit
	59.82%	read_scalar_context()
	20.47%	write_buffer()
	2.97%	oggpack_read()
	2.73%	incr_ptr()
	1.73%	incr_ptr_getbyte(DATA_PTR, int)
	1.49%	huffman_codes()
	1.48%	___divsi3
	1.23%	post_window()
	1.19%	read_vector_context(float*)
	1.05%	residue_curve_decode(vorbis_info*, int, vorbis_decode*...
	0.91%	___modsi3
	0.66%	___float_divide
	0.51%	mdct_backward()
	0.50%	float32_unpack()
	0.48%	putbyte()
	0.48%	incr_ptr_putbyte(DATA_PTR, int, char)

Figure 4.6: Statistical profiling results before CAR optimization (incomplete list)

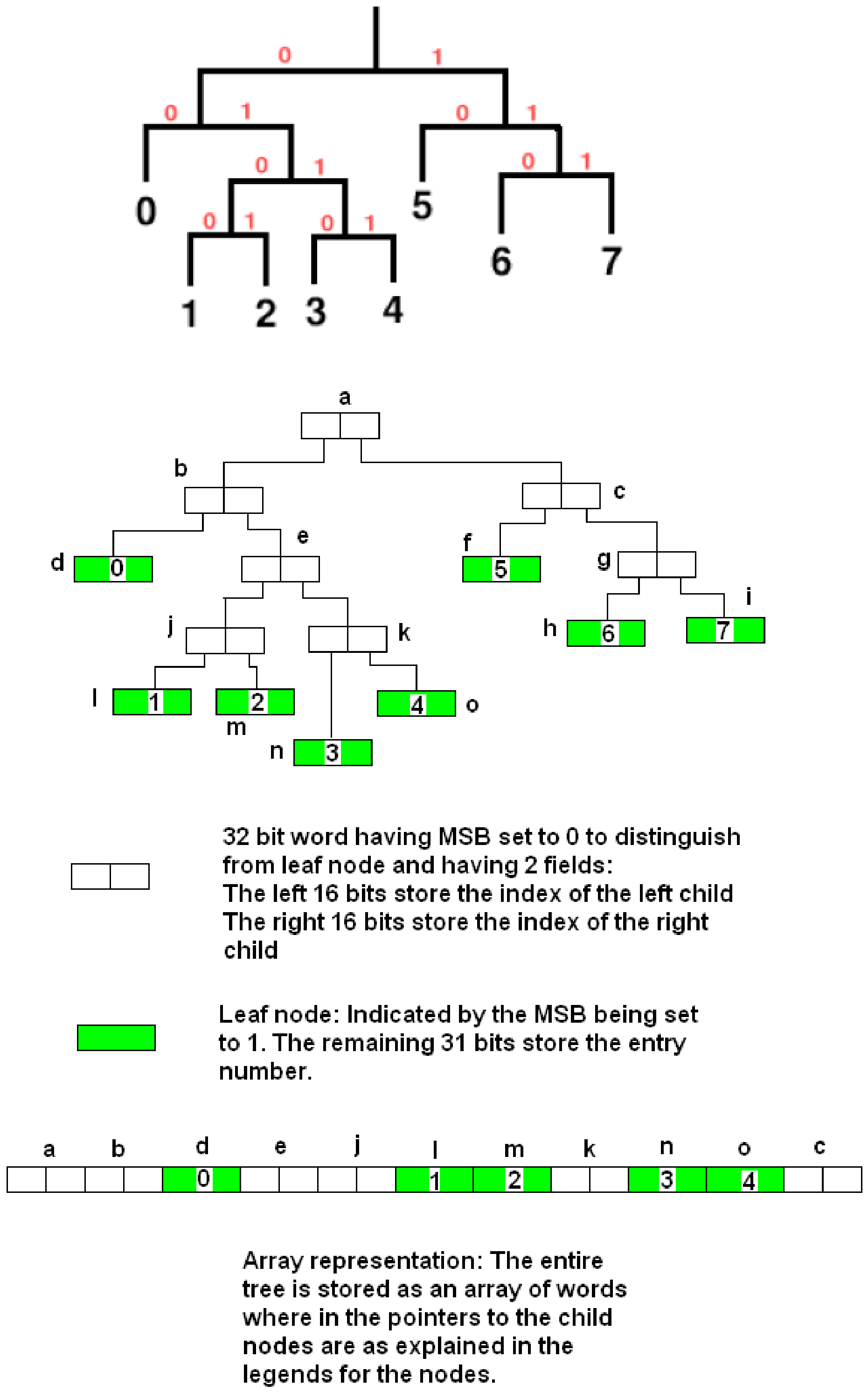
Efficient Huffman tree representation

The `read_scalar_context()` routine reads bit-by-bit from the ogg bitstream, walks the huffman tree. Whenever it hits a leaf node, it outputs the entry number of that particular leaf node. The function `read_scalar_context()` did not "walk" the Huffman tree. It read bits from the ogg bitstream and compared those bits with a list of codewords until there was a match. This linear-comparision method would not be an overhead if the list of codewords were small. But when it is large, as is the case usually, it becomes very intensive. And for decoding each frame of data, calls to `read_scalar_context()` are made very frequently.

For the purpose of optimization, instead of using the list of codewords, if the Huffman tree were built then `read_scalar_context()` would be very fast. But Huffman tree building is an offline process and cannot be included in the real-time decode loop. So Huffman trees for each of the codebooks would have to be stored in the heap. This would mean that a minimum of three 32-bit words had to be allocated for each node of a tree (value, left child address, right child address). There would be tens of such trees with each tree having hundreds of nodes on an average. This would bloat the dynamic memory requirements. So to achieve a fast execution of `read_scalar_context()` and to *reduce* the heap requirements, a different kind of data structure was used.

What was sought was a data structure that provided all the properties of the 'tree', and yet consumed considerably less space. A Compressed-Array-Representation (CAR) of the tree was used.

In CAR every tree is an array of integers. Every node in a tree is a 32-bit integer. The higher 16 bits of data contain the index of the array element which is the left child of that node in the tree. The lower 16 bits contain the index of the right child. Leaf nodes are marked by setting MSB of that word to 1. Leaf nodes contain the entry number in the lower 31 bits. This representation is suitable because normally the trees are never so big so as to overflow the 16 bits index

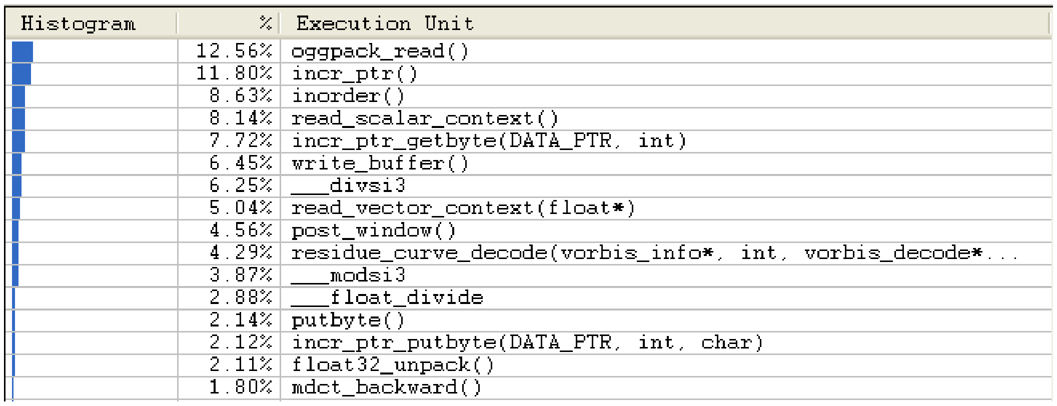


32 bit word having MSB set to 0 to distinguish from leaf node and having 2 fields:
The left 16 bits store the index of the left child
The right 16 bits store the index of the right child

🟩 Leaf node: Indicated by the MSB being set to 1. The remaining 31 bits store the entry number.

Array representation: The entire tree is stored as an array of words where in the pointers to the child nodes are as explained in the legends for the nodes.

Figure 4.7: Huffman tree translation to CAR



Histogram	%	Execution Unit
	12.56%	oggpack_read()
	11.80%	incr_ptr()
	8.63%	inorder()
	8.14%	read_scalar_context()
	7.72%	incr_ptr_getbyte(DATA_PTR, int)
	6.45%	write_buffer()
	6.25%	___divsi3
	5.04%	read_vector_context(float*)
	4.56%	post_window()
	4.29%	residue_curve_decode(vorbis_info*, int, vorbis_decode*...
	3.87%	___modsi3
	2.88%	___float_divide
	2.14%	putbyte()
	2.12%	incr_ptr_putbyte(DATA_PTR, int, char)
	2.11%	float32_unpack()
	1.80%	mdct_backward()

Figure 4.8: Statistical profiling results after CAR optimization (incomplete list)

space. This representation can handle trees with less than 2^{15} nodes. In the case of this decoder, it is safely within range. Figure 4.7 shows the translation from a traditional Huffman tree to CAR.

After the implementation of CAR trees, the decoder was subjected to statistical profiling. The results are shown in Figure 4.8. It shows a marked improvement in performance of `read_scalar_context()`. It can also be seen that the function `oggpack_read()` is the critical section. This function could not be optimized further because of the child function `incr_ptr()` it calls. This `incr_ptr()` function is necesary for incrementing array pointers (DATA_PTR structures) for byte-packing in the synchronization and streaming layers. For significant optimization, the entire byte-packing structure has to be changed.

An estimation of required MIPS was done after the CAR optimization using the same procedure described earlier. The observed MIPS count now was around 80.

Memory requirements

The memory requirements of the decoder after the above mentioned optimizations were reduced drastically. The memory map of ADSP-21364 is shown in Figure 4.9 for reference. It can be seen where each of the sections are placed in memory and how much of each section is used. This is flexible due to the existence of the linker description file.

1. 73% of 1 Mbits of heap space in Block-1 RAM is used
2. 33% of 0.25 Mbits stack space in Block-3 RAM is used
3. 55% of 0.67 Mbits of program memory in Block-0 RAM is used
4. 60% of 0.5 Mbits of data memory in Block-2 RAM is used

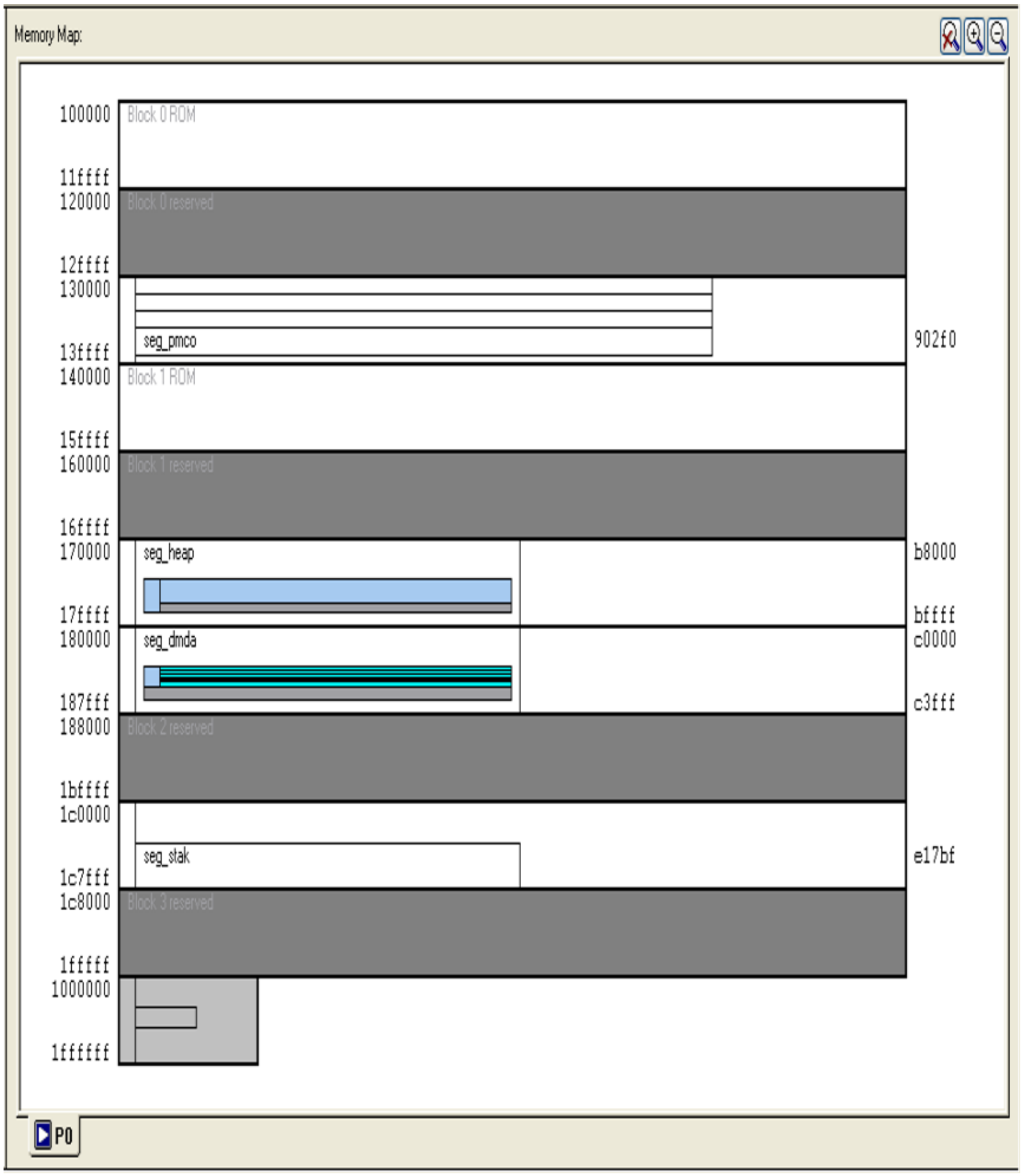


Figure 4.9: Memory map of ADSP-21364

Chapter 5

Summary of results

The work done by us and the subsequent results obtained are being hereby summarized and stated below:

1. The SHARC ADSP-21364 hardware architecture was studied.

2. A study of the Ogg Vorbis audio codec encoding and decoding was carried out and a comprehensive documentation was prepared for the same.

3. The floating point version of the Ogg Vorbis decoder reference code was implemented on the MSVC++ platform. It was found that the floating point code consumes approximately 2.8 Mbits of heap memory and a safely small amount of stack memory.

4. Optimization and implementation of the floating point code for the MSVC++ platform. This had a direct bearing upon reduction in `seg_pmco` or the program code section of the ADSP-21364 memory.

5. Creation of a .ldf file with description for multiple heaps to port the code on to the processor.

6. Initial, comment and setup header decoding done on the emulator.

7. Modified code with packing of bytes done in the synchronization and streaming layers.

8. Flash Programming small audio samples of various sample frequencies and playing them back via SPORT to gain familiarity with the board.

9. The Ogg Vorbis library downloaded from www.xiph.org had a lot of redundant structures and variables. In many occasions, the dynamically allocated memory was not freed. Hence the code was rewritten in full compliance with the Vorbis I specification document, but with simple global structures. Only information (fields, structures etc) required for further processing was saved.

10. The problem of heap overflow was overcome by building the vector table as and when it was required during audio decode phase. This resulted in a reduction in the memory requirements (to about 0.7 Mbits) of the code enabling it to be implemented on the processor *without* the implementation of multiple heaps.

11. The floating point version of the Ogg Vorbis codec with file I/O was implemented in VDSP++. The code was tested for bit exactness with the reference implementation.

12. Implementation of flash writing and DMA driven parallel port flash reading.

13. Implementation of a ping-pong buffer system to read from internal memory, route it through SPORT using SRU to the on-board AD1835A codec chip for playback. The system was able to handle different playback speeds. This resulted in complete removal of file I/O.

14. After file I/O removal the decoder was taking about 290 MIPS. A reduction in the processor core frequency resulted in a deterioration in the audio quality of the song. Hence to reduce the MIPS requirement of the decoder a statistical profiling was done to determine the functions that were taking up the most of the time. It was seen that in the `read_scalar_context()` function i.e. during the Huffman tree decode the processor was spending about 60% of its active time.

15. Instead of storing codewords in memory, the CAR tree optimization was used to reduce the time complexity of the `read_scalar_context()` function. This change resulted in a drastic reduction in the MIPS requirement of the decoder, which now fell to about 80 MIPS.

16. The memory utilization, as described earlier, is given below

    - 73% of 1 Mbits of heap space in Block-1 RAM is used
    - 33% of 0.25 Mbits stack space in Block-3 RAM is used
    - 55% of 0.67 Mbits of program memory in Block-0 RAM is used
    - 60% of 0.5 Mbits of data memory in Block-2 RAM is used

Implementation of real time decoding of Ogg Vorbis audio on Analog Devices ADSP-21364 has been achieved with the decoder taking about 80 MIPS.

Chapter 6

Conclusions and scope for future work

6.1 Conclusions

This project, being an industry sponsored project provided us an insight on the type of current research being carried out in the industry. It has enhanced our knowledge of the working of a DSP.

We also observe the fact that the Ogg Vorbis is a highly competitive codec and due to its extremely flexible nature, this codec is on the verge of becoming one of the leading audio codecs in the near future. This project gave us the opportunity to implement this codec on the SHARC ADSP 21364 architecture for the first time.

The following areas were explored and the respective skills acquired.

1. Programming and optimization
2. Familiarization with a generic DSP architecture
3. Interfacing of the DSP with external peripherals
4. Familiarization with use of industry-standard tools such as VDSP++
5. Also, the codec used DSP concepts such as MDCT and vector quantization, and the implementation of this codec provided us with a practical application of DSP to develop embedded systems

Thus, on the whole, this project provided us with a complete picture of the product development cycle.

6.2 Scope for future work

Future work that could be done in this direction includes the following:

- Booting from the Flash memory so that the system can be made a stand-alone device
- Code optimization of the floating point code for lower memory usage.
- Assembly level optimizations for reduction of MIPS.
- Integration with other standard codecs for a complete audio player system in embedded environments

Bibliography

[1] Ogg documentation from http://www.xiph.org.

[2] Xiph.org. *Vorbis I Specification.*

[3] Analog Devices Inc. *ADSP-21364 EZ-KIT Lite Evaluation System Manual*, January 2005. Rev 2.0.

[4] Analog Devices Inc. *ADSP-21364 SHARC Processor Data Sheet.* Rev 0.0.

[5] Analog Devices Inc. *ADSP-2136x SHARC Processor Hardware Reference*, October 2005. Rev 1.0.

[6] DSP Applications group. *Understanding and Using Linker Description Files.* Analog Devices Inc. Engineer-to-Engineer Note EE-69.

[7] Analog Devices Inc. *VDSP++ 4.5 C/C++ Compiler and Library Manual for SHARC Processors*, April 2006. Rev 6.0.

[8] Analog Devices Inc. *ADSP-2136x SHARC Processor Programming Reference*, November 2005. Rev 1.0.

[9] Analog devices inc., website http://www.analog.com.