University of Stuttgart
M.Sc. Program INFOTECH

Examiner:	Prof. Dr. Hans-Joachim Wunderlich
Supervisor:	Dr. Rainer Dorsch
Begin:	01.01.2002
End:	15.07.2002
CR-Classification:	B.7.1 C.3 C.5

Design of an Audio Player as System-on-a-Chip

Luis Azuara
Pattara Kiatisevi

Division of Computer Architecture
Institute of Computer Science
Breitwiesenstr. 20-22
70565 Stuttgart

Abstract

An Ogg Vorbis [35] audio decoder based on Xiph's Vorbis reference library has been designed as System-on-a-Chip using hardware/software co-design techniques. A demonstrator was built on the XESS XSV-800 prototyping board [8]. The hardware architecture was built on the LEON SoC platform [10], which contained an open source SPARC-V8 architecture compatible processor, an AMBA bus, and the RTEMS embedded operating system. The audio interface hardware core from a previous project [2] was imported and reused. Vorbis stream decoding process was too computation-intensive for a real-time software-only decoder on the target platform. After an analysis of the Vorbis decoding algorithm, it was partitioned and the hardware part of the algorithm, MDCT, was designed as an AMBA compatible core, implemented and added to the system. The final Vorbis audio player decoded Vorbis streams with the help of this MDCT-core.

Acknowledgement

Special thanks for help and comments from Dr. Rainer Dorsch, Prof. Dr. Wunderlich, Jiri Gaisler, Joel Sherill, Aaron Grier, and other RTEMS folks, Daniel Bretz, Christopher Montgomery, Segher Boessenkool, Michael Smith, Dan Conti, and Dr. Van den Bout.

Pattara would like to thank Supavadee Monsathaporn for proof reading.

Contents

1 Ogg-on-a-chip **7**

1.1 Introduction . . . 7
1.2 The Work . . . 8
1.3 Project Elements . . . 9
1.4 Work Packages . . . 12

2 General Background **15**

2.1 Introduction to Compression . . . 15
2.2 Audio Compression . . . 17
2.3 Ogg Vorbis . . . 18
2.4 MDCT . . . 20
2.5 Overview of the AMBA specification . . . 22
2.5.1 Advanced High-performance Bus (AHB) . . . 22
2.5.2 Advanced System Bus (ASB) . . . 22
2.5.3 Advanced Peripheral Bus (APB) . . . 23
2.5.4 Objectives of the AMBA specification . . . 23

3 Embedded Software **24**

3.1 Introduction . . . 24
3.2 Preliminary Analysis . . . 28
3.2.1 Performance Observation on Linux PC . . . 28
3.2.2 Memory Usage Analysis . . . 29
3.2.3 Analysis . . . 31
3.2.4 Vorbis Library Profiling . . . 31
3.2.5 Summary of Preliminary Analysis . . . 34
3.3 Modeling of the System . . . 35
3.3.1 TSIM – LEON Simulator . . . 35
3.3.2 Audio Core as TSIM I/O Module . . . 35
3.3.3 Test of Audio Core TSIM I/O Module . . . 39
3.3.4 Summary . . . 42
3.4 Software Optimization . . . 42

3.4.1 Cross Compilation of Ogg/Vorbis Library to SPARC Platform . . . 42
3.4.2 Simpleplayer Test Program . . . 43
3.4.3 Vorbis Optimization . . . 45
3.4.4 Summary . . . 48
3.5 Hardware/Software Partitioning . . . 48
3.5.1 Hardware Limitations . . . 48
3.5.2 Hardware/Software Partitioning . . . 49
3.5.3 Summary . . . 53
3.6 Player Development . . . 53
3.6.1 RTEMS . . . 54
3.6.2 Device Driver for Audio-Core . . . 57
3.6.3 Final Player with Sound Output . . . 58
3.6.4 Summary . . . 60
3.7 Conclusion . . . 60

4 Underlying Hardware 62
4.1 Platform Exploration . . . 63
4.1.1 Upgrading to latest LEON Version . . . 63
4.1.2 Hardware Constraints . . . 65
4.1.3 Audio Core . . . 65
4.1.3.1 Audio Core Configuration . . . 65
4.1.3.2 Stereo Function for Audio Core . . . 66
4.1.3.3 Interrupt and Internal Stop Address . . . 67
4.1.4 Ogg-on-a-chip Hardware Configuration . . . 68
4.1.4.1 Extraction and Integration of Meiko FPU to LEON 69
4.1.4.2 DSU Integration . . . 70
4.1.4.3 Integer Unit Configuration . . . 72
4.1.4.4 Cache Configuration . . . 72
4.1.4.5 AMBA Configuration . . . 73
4.2 MDCT Core Design . . . 73
4.2.1 MDCT Algorithm in Ogg-Vorbis . . . 74
4.2.1.1 Twiddle Factors . . . 74
4.2.1.2 Mini-MDCT Calculation Process . . . 75
4.2.1.3 Pre-twiddling . . . 77
4.2.1.4 Butterflies calculations . . . 80
4.2.1.5 Remarks . . . 81
4.2.2 MDCT Core Architecture . . . 82
4.2.2.1 AMBA Interface . . . 84
4.2.2.2 Control Unit . . . 85
4.2.2.3 Arithmetic Unit . . . 87

4.3 Simulation, Synthesis and Test 89
4.3.1 Simulation Branch 91
4.3.1.1 Post-synthesis Simulation 92
4.3.2 Synthesis Branch 93
4.3.3 Hardware Test 94
4.4 Final System Test 95

5 Conclusion **96**

A CVS **97**

Bibliography **100**

List of Figures

1.1 Overview of the system 9
1.2 Work packages 14

2.1 Vorbis encoding and decoding 19
2.2 Typical succession of window shapes 21

3.1 Work packages for software part 25
3.2 Overview of software development components 27
3.3 Screen-shot of top command 29
3.4 Screen-shot of *purify* tool 30
3.5 Screen-shot of TSIM running on Linux PC 36
3.6 Audio core registers (translated from [2]) 37
3.7 Example of a write access to audio core TSIM I/O module 39
3.8 Graphical call graph starting from *vorbis_synthesis()* 46
3.9 Modified *mdct_backward*() function 52
3.10 Example of open() and write() calls to audio device 58
3.11 Communications between threads 59

4.1 Audio Core diagram. 67
4.2 Platform Configuration. 68
4.3 MicroSparcII FPU Block Diagram. 70
4.4 MDCT Block Diagram. 75
4.5 Twiddle factor LUT in memory. 76
4.6 MDCT callgraph. 77
4.7 Pre-twiddling process 78
4.8 Odd part process 79
4.9 Even part process 79
4.10 Butterflies for big block (2048 elements) 80
4.11 Basic butterfly 81
4.12 MDCT core architecture. 83
4.13 FSM of control unit. 86

4.14 Arithmetic Unit. 88
4.15 Hardware workflow 90
4.16 Screen-shot of Modelsim 92
4.17 Screen-shot of Synplify Pro 94

Chapter 1

Ogg-on-a-chip

In this chapter, brief introduction about embedded systems and System-on-a-Chip is given. Then the goals, elements and work packages of the project are discussed.

1.1 Introduction

Main technologies involved in this project are embedded systems and System-on-a-Chip. In this Section, both of them are shortly described.

Embedded Systems

Nowadays our life is full of interactions with embedded systems and processors. Each day we have contacts with 20 microprocessors in average, and most of these microprocessors are incorporated in embedded systems. An embedded system is a special-purpose computer built integratedly into a device. The embedded systems have varieties of types and sizes. It could range from a single microprocessor to a complex System-on-a-Chip system.

Embedded systems usually have a processor and memory hierarchy. In addition to that, there are a variety of interfaces that enable the system to measure, manipulate, and interact with the external environment. The human interface may be as simple as a flashing light or as complicated as real-time robotic vision. Embedded system usually provides functionality specific to its application. Its software often has a fixed function which is specific to the application. Instead of executing spreadsheet, word processing and engineering analysis applications, embedded systems typically execute control flows, finite state machines, and signal processing algorithms. They must often detect and react to faults in both the computing and surrounding electromechanical systems, and must manipulate application-specific user interface devices.

Embedded systems could be realized using special-purpose field programmable gates arrays (FPGA), application specific integrated circuits (ASIC), or even non-digital hardware to increase performance or safety.

System on a Chip (SoC)

The nearly boundless transistor capacity available for advanced integrated circuits (IC) gave birth to an electronic system design revolution. From the physical point of view, a large number of transistors allows the integration of very complex systems in one single die. Such systems can be hybrid analog-digital with different elements like processors, memories, sensors and application specific circuits. This technique presents many desirable features like low power consumption, small size and weight, and low cost for large volumes. Such a system is called *System-on-a-Chip* (SoC). Because of these advantages, SoCs became a popular way of embedded systems implementation.

The new techniques for SOC design make possible the combination of large, pre-designed complex blocks (or so-called *cores* or *IP blocks*) and embedded software. Additionally they reduce time-to-market for new products, which is very important taking into account that hi-tech products have an extremely short life-cycle. SoC design techniques are focused on the problems of evaluating, integrating, and verifying multiple pre-existing blocks and software components. This is characterized by more in-depth system-level design, concurrent hardware/software design and verification at all levels of the design process.

1.2 The Work

This project (also known as **Ogg-on-a-Chip**) aimed to demonstrate the use of System-on-a-Chip (SoC) technology by developing an audio decoder as SoC utilizing hardware/software co-design technique. The goals of the project were as follows:

- An Ogg Vorbis [35] audio decoder shall be implemented as embedded system based on open source LEON [10] SoC platform. The target systems are low CPU performance embedded devices like PDA or cell phone.
- A demonstrator shall be implemented on the FPGA-based XESS XSV-800 prototyping board [8].

In order to enable decoding of Vorbis data on systems with low CPU performance, part of the decoder was implemented in hardware in order to speed up the com-

putation. Hardware/software co-design was used to identify the most promising hardware/software partition.

The hardware part of the previous project, “Digital Dictation Machine as System-on-a-Chip” or DDM [2], the audio core as interface to the audio chip, was imported and reused in this project.

Embedded operating system was used for task management and hardware-resource abstraction and the open source RTEMS operating system [6] was chosen.

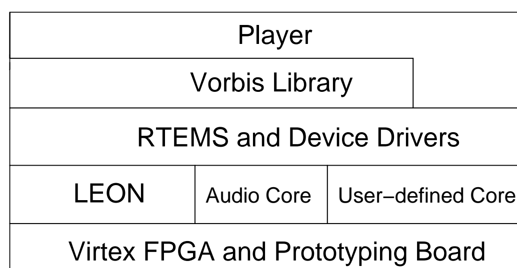

Figure 1.1: Overview of the system

In Figure 1.1, an overview of the system is shown. Components are classified into layers. The whole system was implemented on the FPGA and the prototyping board. The LEON SoC platform and additional cores, e.g. audio core and other user-defined core serve as hardware architecture. The RTEMS embedded operating system runs on top of LEON and contains appropriate device drivers for hardware cores. The Vorbis player makes use of Vorbis library (decoding part) and runs as a sole process in the system. It decoded compressed music data with the help of the user-defined core and delivered music output through audio core to audio device.

1.3 Project Elements

In this Section, useful elements that were involved in the project are shortly described.

LEON Platform

LEON is a 32-bit SPARC [29] compatible processor developed by the European Space Agency (ESA). It is available freely with full source code under LGPL

(GNU Lesser General Public License)[1] as ESA would like to promote the SPARC architecture and enable development of system-on-a-chip (SOC) devices using SPARC cores. It was initially designed and implemented by Jiri Gaisler while at ESA and is now maintained under contract by Gaisler Research.

The LEON processor version 2 (current version) has the following interesting features :

- SPARC V8 compatible integer unit with 5-stage pipeline
- Hardware multiply, divide and MAC units
- Separate, direct-mapped instruction and data caches
- Full implementation of AMBA-2.0 AHB and APB on-chip buses
- On-chip peripherals such as uarts, timers, interrupt controller and 16-bit I/O port,
- Interfaces for Meiko floating-point unit and user-defined co-processor

The LEON processor is extensively configurable and can be efficiently implemented on both FPGAs and ASIC technologies. Support for AMBA bus enables the easy integration of user-defined cores. The only technology-specific megacells needed are RAM cells for caches and register file. The latest release of LEON processor at the time of project was Leon-2 1.0.2a.

XESS XSV-800 Prototyping Board

The facts that SoC must be developed very fast in order to response to the market, and that the fabrication process of one single chip is extremely expensive, present the perfect scenario for rapid-prototyping boards, which are used to model SoC before proceeding to a silicon implementation.

The XSV-800 prototyping board is equipped with a Xilinx Virtex XCV800 FPGA with 800,000 gates, two independent banks of 512K x 16-bit SRAM (2 MB in total) for local buffering of signals and data and 1 MB of Flash. The XSV-800 offers a lot of peripheral interfaces e.g. audio, USB, PS2, and VGA. The audio chip AKM AK4520A can process stereo audio signals with up to 20 bits per sample and a bandwidth of 20 kHz.

[1] GNU Lesser General Public License, more information at `http://www.gnu.org/licenses/lgpl.html`.

Digital Dictation Machine

The project *Digital Dictation Machine (DDM) as System-on-a-Chip on an FPGA-based Prototyping Board* was done by Daniel Bretz [2] as his Diploma Thesis while at University of Stuttgart (February 2001). The final system performed the function of a digital sound player and recorder with internal memory buffers. It ran on the XSV-800 prototyping board with LEON-1.2.2 processor and user-defined audio core to communicate with audio device.

Audio Compression

Audio compression reduces storage consumption and enables transportation of high quality audio data at low data speed. Without any data reduction, a second of CD-quality music (2-channel stereo, 16 bits per sample, and 44.1 kHz sampling frequency) consumes more than 1.4 Mbit of data storage. Audio compression reduces the amount of music data depending on the algorithm and quality level needed. There are a lot of audio compression algorithms developed and used nowadays. One of the most popular algorithms is *MPEG audio compression layer III* (or MP3) with the data reduction rate of 1:10 to 1:12 while still maintaining the original CD sound quality. The uses of MP3 can be found from personal computers to portable audio devices and even household audio equipment. Although software for MP3 decoders and encoders are available freely on the market but the use of the MP3 algorithm is not completely free. MP3 algorithm is patented by *The Fraunhofer Institut Integrierte Schaltungen (Fraunhofer IIS-A)*, Erlangen, Germany and royalty fee must be paid when creating or using MP3 encoders[2].

Ogg Vorbis

Xiph.Org Foundation [35] developed in 1997 an audio compression algorithm called *Ogg Vorbis*. According to information from Xiph, Ogg Vorbis is a fully open, non-proprietary, patent-and-royalty-free general purpose compressed audio format for high quality audio (44.1-48.0kHz sampling frequency, 16 bits per sample or more, polyphonic) at fixed and variable bit-rates from 16 to 128 kbps/channel. Ogg Vorbis is categorized in the same class as MPEG-4 Audio (AAC and TwinVQ) and claims to have higher performance than MPEG-1/2 audio layer 3, MPEG-4 audio (TwinVQ), WMA and PAC.

[2]More information about legal aspect of MP3 can be found at `http://www.mp3licensing.com/`.

Claiming to be fully-open (source code of Xiph's reference Vorbis library including decoder and encoder are distributed under BSD license[3] while other accompanied utilities are available under GPL[4]), patent-free[5], and license-free made Ogg Vorbis become prevalent nowadays. Ogg Vorbis is expected to be the most viable choice to replace proprietary MP3 in the near future. At the time of this project, the latest stable version of Vorbis library is 1.0 RC3. Ogg Vorbis software is available on almost all personal computer platforms.

1.4 Work Packages

Tasks in the project were divided into steps shown in Work Packages diagram in Figure 1.2. The *specification* phase defined goals and requirements of the system. The *feasibility study* was conducted to find out if the hardware platform, with future reasonable optimizations, would be powerful enough to decode Vorbis stream and if some parts of algorithm could be efficiently implemented in hardware.

Later, the main works were divided into *hardware part* and *software part* in order to have simultaneous developments of both software and hardware. In software part, the *hardware/software partitioning evaluation tools* were provided as preparation for *hardware/software partitioning* in the next step. In order to achieve fast software development, the TSIM (LEON simulator) was used in order to model the system on the software so that software development tasks could be done on the simulator instead of the real hardware. At this time, *hardware configuration* was done in the hardware part to study the capability and limitations of the platform.

Based on the information from *hardware configuration* phase of hardware part and tools from *hardware/software partitioning evaluation tools* phase of software part, *hardware/software partitioning* was done and the partition between hardware and software was proposed. The selected hardware part of algorithm (MDCT function) was designed, modeled and implemented in hardware part. Concurrently in software part, further development of the full version of Vorbis player was done.

The hardware part and software part met again in the *final test* phase. All components were tested together. The final player decoded the Vorbis stream

[3] More information about BSD type of license can be found at `http://www.opensource.org/licenses/bsd-license.html`

[4] GNU Public License, more information at `http://www.gnu.org/licenses/gpl.html`.

[5] It is noted that at the time of this project, the specification of Ogg Vorbis format was not yet available to the public. This implies that the patent-free claim could not yet be fully verified. There was a discussion regarding this also at `http://www.kuro5hin.org/story/2002/4/25/212840/001`. Despite this fact, Ogg Vorbis has already been widely accepted by users.

with the help of the user-defined core on the real hardware.

Report Organization

The following chapter (Chapter 2) covers theoretical information of audio compression and Ogg Vorbis. As works in the project were divided into two parts, Hardware part and Software part, later chapters describe work done in each part separately – Chapter 3 *Embedded Software* written by Pattara Kiatisevi and Chapter 4 *Underlying Hardware* by Luis Azuara. At the end, all the work is summarized in the Chapter 5.

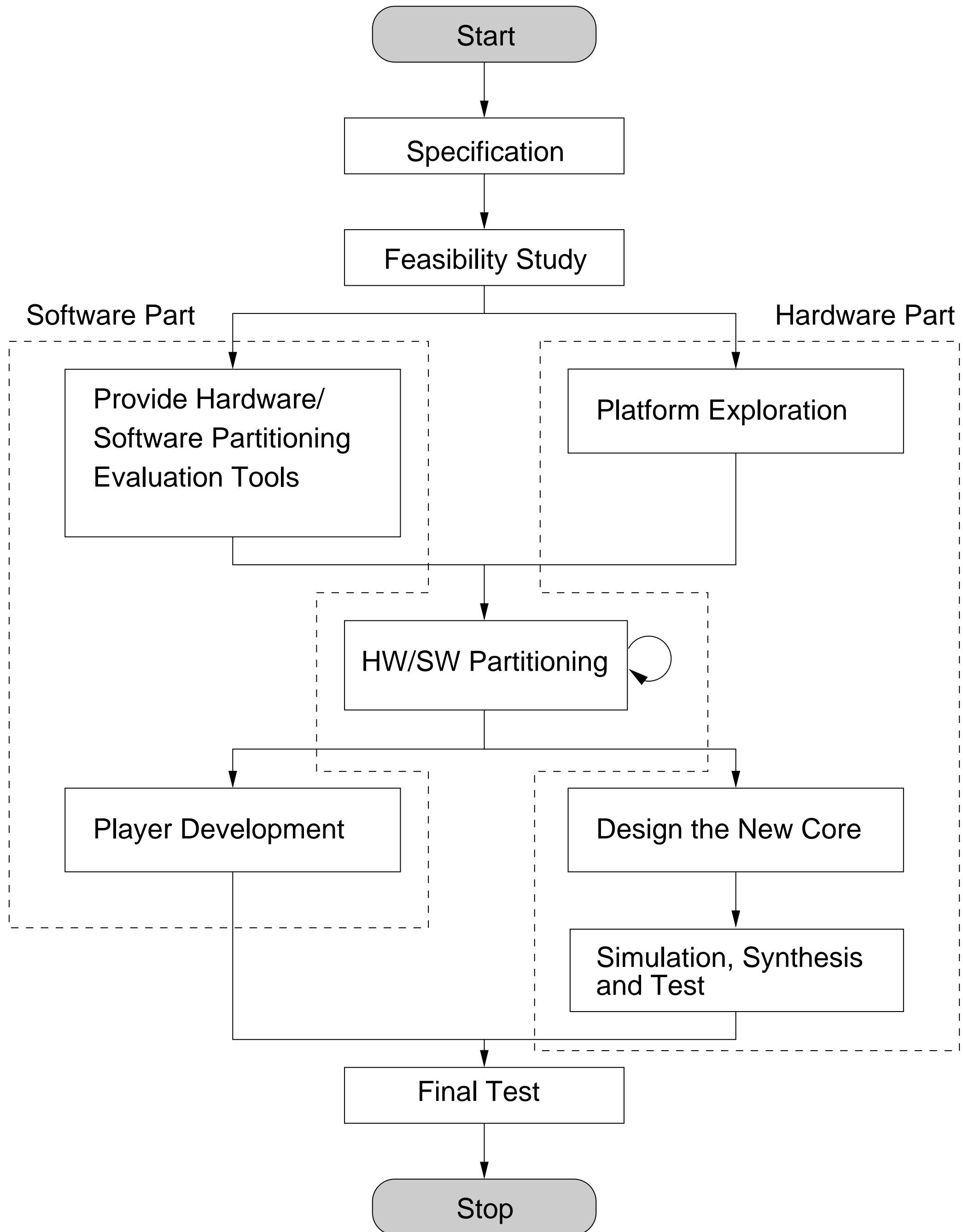

Figure 1.2: Work packages

Chapter 2

General Background

In this chapter, basic concepts of compression and audio compression are described. Brief explanation of Ogg Vorbis algorithm is given. The concept of MDCT transform which is an important transform used by Ogg Vorbis is discussed. Lastly, an overview of AMBA sepecification is presented.

2.1 Introduction to Compression

In this Section, general information about compression [30] is described. Useful terms and techniques are briefly mentioned.

Compression is a science of reducing the amount of data used to convey information. Information is usually not completely random. By reducing the redundant information, we could achieve the *lossless* compression which means that the decoding process will give exactly the same data as input data without losing any bit. Examples of lossless compression techniques are *run length encoding* and *entropy encoding*. Run length encoding replaces the repeated occurrences of the same symbol with shorter representation. Entropy encoding relies on the entropy of source information. *Huffman coding* is an entropy encoding which analyzes the probability of occurrences of each symbols and represents them with new codewords. Applications like zip, PNG image format are using lossless compression technology.

On the other hand, in *lossy* compression, some information that is irrelevant will be discarded and not reconstructed. Examples of this irrelevant information are information that can not be perceived anyway by the receiver and artifacts (features or elements that are not truly part of the information e.g. noise, quantization noise, filter ringing, film scratches). Lossy compression relies on a knowledge of how the information will be perceived by the recipient. The fidelity compared to the source information may be lost but in a way that is little or unperceivable by

recipient. JPEG, MPEG I, II (both video and audio) and Ogg Vorbis are examples of lossy compression techniques.

According to symmetrical aspect, compression algorithms can be *symmetric* or *asymmetric*. In symmetric systems, the same amount of effort is needed for encoding and decoding. Example of application is video conferencing system, there is no need for encoder to work faster than decoder and reciprocal. However, for video or TV broadcasting, once encoded, the video will be decoded by thousands of viewers. In this scheme, having a very fast decoder is beneficial and a slow encoder is acceptable as it will be used only once. MPEG and also Ogg Vorbis are examples of asymmetrical algorithms.

In the case that information source is not discrete-memoryless-source, which means the probability for a current symbol depends on previous symbol(s), e.g. photographic images (intensity/color of a pixel depends on that of those pixels around it), several techniques could be applied here to compress the information by trying to predict the next value and store only the different of the prediction and real value. This technique is called *predictive coding*.

Signals that changes its amplitude over time (including audio signal) or image that each pixel changes its value over the x and y axes are said to be in *time-domain*. However, in many cases, it is much easier to analyze and manipulate these signals in other domain e.g. frequency domain, and after finished, convert back to time domain. This conversion forward and backward is called *transformation*. Transform will convert set of values from one set to a different set. A transform used extensively in the engineering and sciences fields is Fourier transform. Fourier Fourier assumes however infinite time domain signal and continuous function in time, and gives complex values in frequency domain. In image and audio/video compression, Discrete Cosine Transform (DCT) is more generally used. DCT works with discrete signal, which is usually the case for image/video/audio compression, instead of continuous signal and gives only real part in the result, no complex component. In Ogg Vorbis, a modified version of DCT called MDCT is used.

In order to obtain discrete signal, continuous signal (e.g. audio signal) will be sampled at a certain frequency called *sampling frequency* and then stored in a representation format of a defined amount of storage (e.g. 8-bit, 16-bit, 32-bit). The process to fit the sampled values (which can be arbitrary or limited by some ranges) to a fixed set of representative values is called *quantization*. Quantization is a lossy process because the selected representative value is merely the nearest value to that real value, but not exactly the same. Error caused by this is called *quantization noise*. We can avoid it by selecting the appropriate representative setting so that this quantization noise is not significant compared to other errors or noises. Quantization can be scalar or vector. In scalar case, individual value will be represented by a value in the fixed possible set of representative values.

In vector case, not only one individual value but it can be an array of values to be represented by the representative values. Codebook is the mapping between representative values and represented values. Example application that uses vector quantization is GIF image compression algorithm.

2.2 Audio Compression

Similar to other kinds of compression, the main idea of audio compression is to eliminate redundancy. Redundancy can be *absolute*, which means removal of it will not cause any data lost (lossless), or *perceptual*, indicating that removal will cause the data lost (lossy) but it will not be significant or hardly perceivable by human observers. This latter part contributes largely to high compression rate obtained in the modern audio compression algorithm including Ogg Vorbis.

Examples of perceptual redundancies are part of information that is intrinsically insensitive to perceiver e.g. higher frequency range than that human can hear or the receiver device can produce, masking effect in human hearing, and correlation between various audio channels. Study about human perceptual of sound and how hearing works is called *psychoacoustics*.

Masking in human hearing is the effect of the natural behavior of basilar membrane of the inner ear. It is frequency sensitive and can even vibrate actively providing positive feedback to low amplitude vibrations. Basilar membrane is divided to 24 or more finite regions [30], each can vibrate over a small range of frequencies, but only one at a time determined by the strongest stimulus within that range and it is unaffected by any smaller stimuli. This causes the frequency masking: any frequency in the signal in the same band, but lower in amplitude need not to be encoded as it will not be perceived anyway because of the effect from the prime stimulus. Thus the audio band can be split (linearly or non-linearly) into regions as small as or smaller than the regions of the basilar and then analyzed using this psychoacoustics knowledge to identify unperceivable information.

Positive feedback of the basilar also contributes to *temporal masking* because the vibration responds slowly to changes in the amplitude of the stimulus. It will be less sensitive to other sound short time before, and longer time after a strong stimulus.

Dividing input data stream into short frames can also improve compression ratio because the amplitude of contiguous samples tend to be in similar range, thus the samples can be stored in the way that the number of bits per sample can be reduced.

Apart from these general techniques, there are other advanced and complex techniques available for audio compression specially for each algorithm.

2.3 Ogg Vorbis

Ogg Vorbis is an audio compression format developed by the Xiph.Org Foundation [35], a non-profit organization working in the area of Internet multimedia technology. Ogg itself is a big framework for several multimedia projects including Vorbis (audio) and Tarkin (video). Vorbis is the first project in the Ogg family aimed at the audio compression/decompression. The discussion about Vorbis development took place on the *vorbis-dev@xiph.org* mailing list and has been visible to public since August 1999.

Ogg Vorbis is lossy, asymmetrical algorithm and utilizes several techniques mentioned before, e.g. dividing of input into short blocks, MDCT (modified version of DCT), psychoacoustics, vector quantization, predictive, and many other advanced techniques. Ogg defines generally the format of data to be packed into streams and transported regardless of data content in the stream which can be Vorbis or future Ogg codecs. Ogg bit-streams are streams of octets which can compose of several logical streams inside one physical stream using multiplexing or chaining techniques.

Audio data will usually be encoded by the Vorbis encoder, packed into an Ogg bit-stream and then transported to decoder. Vorbis decoder opens the Ogg bit-stream, unpack the Vorbis stream data out, decode and give the result as uncompressed audio data.

Ogg Vorbis has no official specification at the time of writing. Study of Vorbis algorithm has been done through its source code, some documents available on xiph.org web-site, Vorbis Illuminated document [3] and discussion with Vorbis developers.

Encoding and decoding of Ogg Vorbis can be classified in 6 big stages. Vorbis encoder takes the raw audio data as overlapped but contiguous short-time segments and analyses the audio data to find the optimal small representation. This stage is called *analysis*. After that it encodes the audio data into a much smaller data representation as determined in the previous step. This stage is called *coding*. Then the raw packets will be packed into streams, called *streaming*. At the other end, the decoder extracts the sequence of raw packets from the stream, the stage is called *streaming-decomposition*. It then tries to reconstruct the sound signal representation from these packets, called *decoding*. Lastly, the audio signal will be regenerated from the decoded representation in the *synthesis* stage. The stages are shown in Figure 2.1.

In Analysis stage, audio data will be divided into overlapping blocks of 2 sizes: short (256 samples) or long (2048 samples). Normally long window will be used except when sudden attacks or explosive sounds occur because short window can prevent temporal spreading artifacts which might be produced by MDCT in case of long window. The step is called *Block Switching*. Each block will be

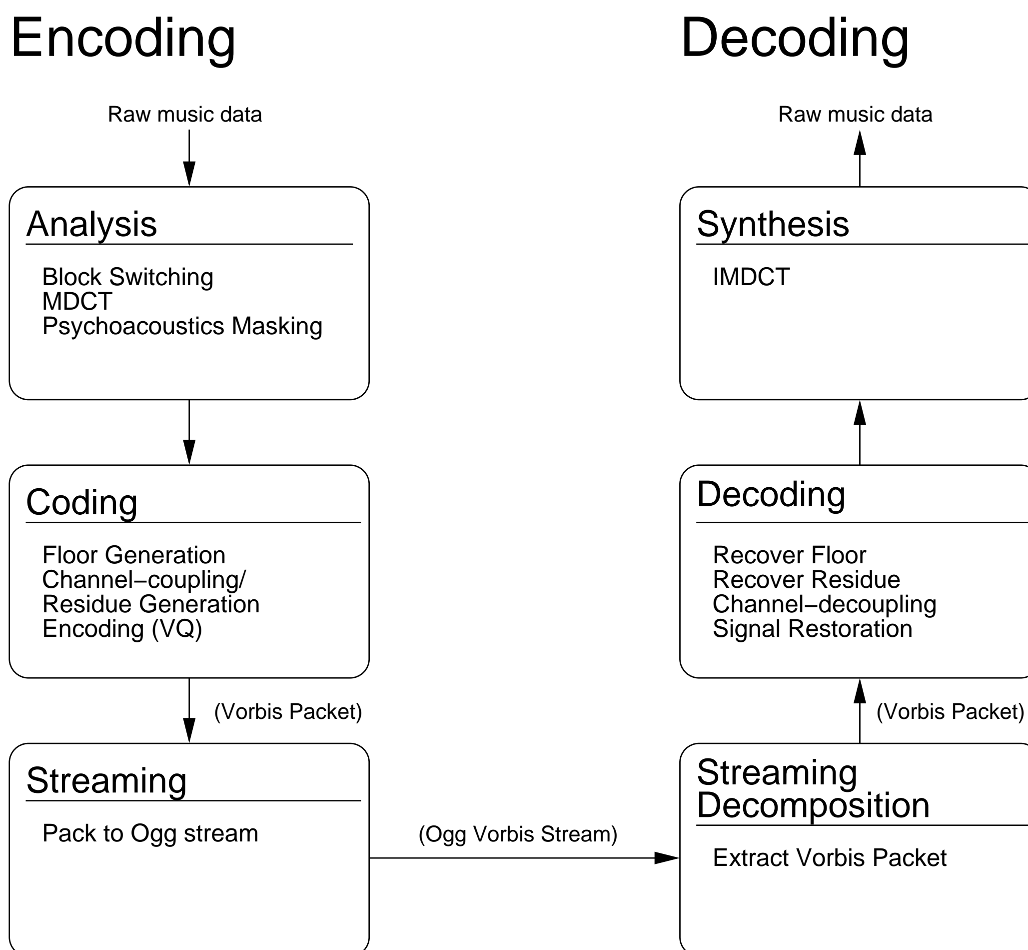

Figure 2.1: Vorbis encoding and decoding

MDCT-transformed to frequency domain and then analyzed in the psychoacoustics masking step.

In Coding stage, information received from psychoacoustics masking process will be used to create the spectral envelope of the signal and floor function. Channel coupling and residue generation will be generated afterwards. Small representation of the audio data (floor and residue) will be encoded using VQ (Vector Quantization) to form a Vorbis packet which will be later, with the VQ codebook, packed into Ogg bit-stream.

Decoding is straight forward and less complex than encoding. Decoder decomposes the Vorbis packet out of the Ogg stream. Vorbis packet is then processed in the Decoding stage to extract floor, residue and do channel-decoupling. The audio signal in frequency domain is recovered, inverse-MDCT-transformed

back to time domain and de-overlapped to form the output audio signal.

In our project, only decoder part was of interest.

2.4 MDCT

The step calculating the inverse MDCT (Modified Discrete Cosine Transform) became an important issue, because was found it is the most calculation expensive process during decoding with Ogg-Vorbis. In this Section is presented a briefly introduction with the mathematical background.

MDCT is widely used in state of the art audio codecs such as MPEG 1 Layer III, Dolby AC/3, or MPEG AAC and of course in Ogg Vorbis. [34]

The MDCT is a linear orthogonal lapped transform, based on the idea of time domain aliasing cancellation (TDAC). It was first introduced in [25], and further developed in [26].

MDCT is critically sampled, which means that though it is 50% overlapped, a sequence data after MDCT has the same number of coefficients as samples before the transform (after overlap-and-add). This means, that a single block of IMDCT data does not correspond to the original block on which the MDCT was performed. When subsequent blocks of inverse transformed data are added (still using 50% overlap), the errors introduced by the transform cancels out TDAC. Thanks to the overlapping feature, the MDCT is very useful for quantization. It effectively removes the otherwise easily detectable blocking artifact between transform blocks.

Be $x(k)$ the samples in the time domain and n the size of the block. $x_t(k)$, $k = 0..n-1$ are the samples used to calculate the frequency domain samples $X_t(k)$, $k = 0..\frac{n}{2} - 1$of the block number t.

According [19] the equation of the direct MDCT is:

$$X_t(m) = \sum_{k=0}^{n-1} f(k)x_t(k)cos(\frac{\pi}{2n}(2k+1+\frac{n}{2})(2m+1))$$

for $m = 0..\frac{n}{2} - 1$

For the Inverse MDCT:

$$y_t(p) = f(p)\frac{n}{4}\sum_{m=0}^{\frac{n}{2}-1} X_t(m)cos(\frac{\pi}{2n}(2p+1+\frac{n}{2})(2m+1))$$

for $p = 0..n-1$

Cancellation of time domain alias terms is done by an overlap add operation:

$$\tilde{x}_t(q) = y_{t-1}(q + \frac{n}{2}) + y_t(q)$$

$$\text{for } q = 0..\tfrac{n}{2} - 1$$

To cancel alias terms the shape of each window must keep the following conditions: The shapes of the windows in succeeding blocks must fit to each other only in the overlapping part. It is possible to split each long block into shorter blocks. Overlapping this shorter blocks must result in the same window shape as used by the overlapping part of a long block. This fact is presented in Figure 2.2.

$$f_{t-1}(\frac{n}{2} + k)^2 + f_t(k)^2 = 1$$

$$\text{for } k = 0..\tfrac{n}{2} - 1$$

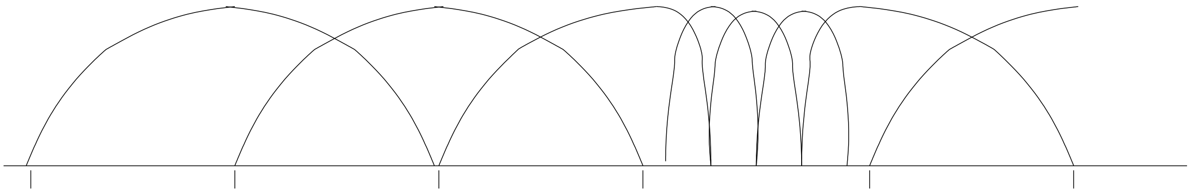

Figure 2.2: Typical succession of window shapes

There must be a symmetry in each half of a window:

$$f_t(k)^2 + f_t(\frac{n}{2} - k)^2 = 1$$

$$\text{for } k = 0..\tfrac{n}{2} - 1$$

and

$$f_t(\frac{n}{2} + k)^2 + f_t(n - 1 - k)^2 = 1$$

$$\text{for } k = 0..\tfrac{n}{2} - 1$$

Several fast algorithms are known. Ogg-Vorbis uses a highly optimized algorithm described in [19], which has been used in real-time implementation of high quality coders for several years. It has the following advantages:

- Low numbers of operations (additions, multiplications, storage operations)
- Minimum size of storage needed (in-place algorithm)
- High robustness against rounding errors
- Simple implementation on general purpose DSP

In chapter 4 the process to implement a part of this algorithm in hardware will be presented.

2.5 Overview of the AMBA specification

The Advanced Microcontroller Bus Architecture (AMBA) specification [20] defines an on-chip communications standard for designing high-performance embedded microcontrollers. Three distinct buses are defined within the AMBA specification:

- the Advanced High-performance Bus (AHB)
- the Advanced System Bus (ASB)
- the Advanced Peripheral Bus (APB).

A test methodology is included with the AMBA specification which provides an infrastructure for modular macrocell test and diagnostic access.

2.5.1 Advanced High-performance Bus (AHB)

The AMBA AHB is for high-performance, high clock frequency system modules. The AHB acts as the high-performance system backbone bus. AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral macrocell functions. AHB is also specified to ensure ease of use in an efficient design flow using synthesis and automated test techniques.

2.5.2 Advanced System Bus (ASB)[1]

The AMBA ASB is for high-performance system modules. AMBA ASB is an alternative system bus suitable for use where the high-performance features of

[1] ASB is not implemented on LEON platform, and therefore is not used in Ogg-on-a-chip project.

AHB are not required. ASB also supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral macrocell functions.

2.5.3 Advanced Peripheral Bus (APB)

The AMBA APB is for low-power peripherals. AMBA APB is optimized for minimal power consumption and reduced interface complexity to support peripheral functions. APB can be used in conjunction with either version of the system bus.

2.5.4 Objectives of the AMBA specification

The AMBA specification has been derived to satisfy four key requirements:

- to facilitate the right-first-time development of embedded microcontroller products with one or more CPUs or signal processors
- to be technology-independent and ensure that highly reusable peripheral and system macrocells can be migrated across a diverse range of IC processes and be appropriate for full-custom, standard cell and gate array technologies
- to encourage modular system design to improve processor independence, providing a development road-map for advanced cached CPU cores and the development of peripheral libraries
- to minimize the silicon infrastructure required to support efficient on-chip and off-chip communication for both operation and manufacturing test.

Chapter 3

Embedded Software

Works in project were divided into hardware part and software part. In this chapter, work in the software part is discussed.

3.1 Introduction

From the full work packages of the project illustrated in Figure 1.2, software part related tasks were taken and elaborated in Figure 3.1. The *provide hardware/software partitioning evaluation tools* phase is now expanded into two smaller sub-phases: *Modeling of the System* and *Software Optimization*. Other phases remain the same. The arrows at the right side of each phase box means this phase has a cooperation with hardware part.

In the first phase, *Feasibility Study*, preliminary analysis and performance estimation of Vorbis decoding process are concerned. The required computing resource of Vorbis decoding process was observed by running the decoder on a Linux PC and measuring the resource usage. As it was planned that a part of algorithm should be implemented in the hardware, the Vorbis algorithm was analyzed based on profiling information and the computation-intensive part was further investigated if it was possible to be implemented as hardware core. The decision to proceed the project was done based on the information gathered in this phase.

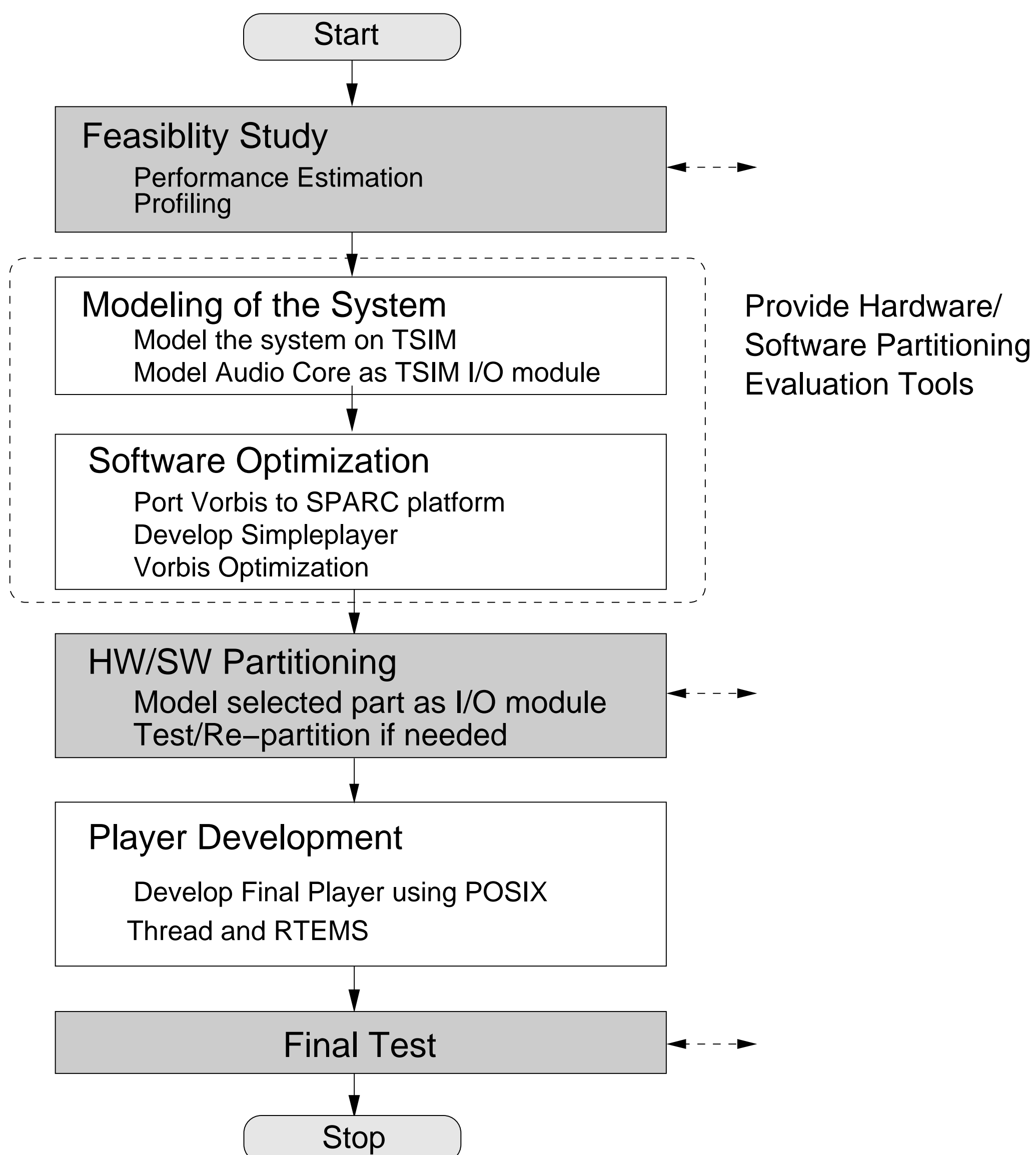

Figure 3.1: Work packages for software part

As mentioned in Chapter 1, in order to achieve simultaneous software and hardware developments, TSIM, the simulator of LEON processor, was introduced to the project in the *Modeling of the system* phase to simulate the hardware architecture on the Linux PC so that software development could be started without having to wait for the development in the hardware part. The audio core imported from [2] could be also simulated using the TSIM's feature of I/O emulation. A C-language program was written to simulate the hardware audio core and plugged

in to TSIM as a *TSIM I/O module*. With the TSIM and its I/O module, the required functionalities of hardware architecture (LEON and audio core) were completely simulated on the software. The test program with raw music output (without any compression) was written and successfully tested on both TSIM (with its I/O module) and the real hardware with real audio-core.

In *software optimization* phase, Ogg Vorbis library was cross-compiled to SPARC platform. A simple version of Vorbis decoder (or *simpleplayer*) without any audio output was written and successfully tested on TSIM. Possible software optimization was applied. Vorbis decoding process was changed to use integer calculations instead of floating-point ones in order not to have an FPU in the design and to boost up the performance.

After the preparation for hardware/software partitioning evaluation tools had been done, the *hardware/software partitioning* was started. TSIM I/O module feature was used again in this phase. A computation-intensive part was selected based on profile information done in the feasibility phase, implemented as TSIM I/O module (to simulate as if it was implemented as a user-defined hardware core) and tested for performance gain. Various partitions were tested and compared easily as everything could be done in software (TSIM) without having to design and test on the real hardware. The inverse MDCT function *mdct_backward* was selected as partition candidate and tested. Finally two partitions (MDCT and Mini-MDCT) were delivered. MDCT (the whole inverse MDCT function) was proposed as a preferred solution. Mini-MDCT, which was a subset of MDCT, was proposed as the secondary solution in case that implementing the whole MDCT function was not feasible.

Last stage in software part is *player development*. The RTEMS [6] operating system was introduced. RTEMS device driver for audio core as abstraction layer of hardware was written. The final Vorbis player which extended the simpleplayer and glued all mentioned elements together was developed. It made use of Vorbis library for decoding Vorbis stream and sent music output to audio device via audio device driver. The final player utilized the POSIX thread (PThread) in order to have achieve the multi-threading capability (the decoder task continued decoding the data while another music-playing task wrote data to audio hardware via audio device driver concurrently). By sticking to POSIX standards, the result player code is portable, the same code ran on both TSIM (SPARC target) and Linux-x86 PC.

Overview of software development components is shown in Figure 3.2. Notice the two lower layers compared to those in Figure 1.1. Instead of LEON and hardware cores, TSIM and I/O modules emulate their functions. TSIM software was running on Linux x86 workstation.

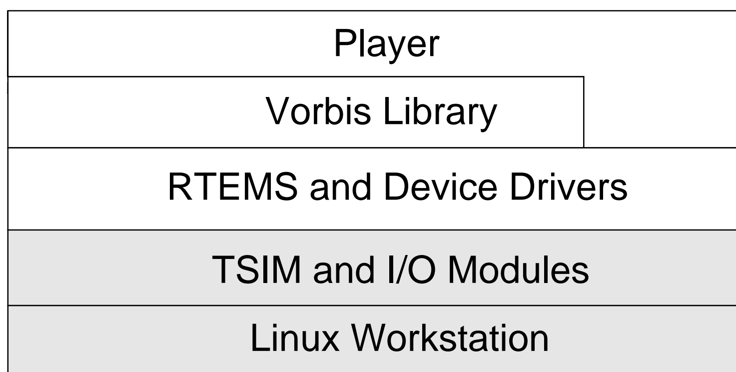

Figure 3.2: Overview of software development components

Development Environment

These below tools were involved in the development processes:

Open-Source Software

- GNU/Linux operating system as development platform
- Ogg Vorbis reference library
- RTEMS operating system
- GNU GCC compiler
- Cygnus newlib C library
- LECCS package from Gaisler Research
- XEmacs for source code editing
- GNU gdb/DDD for debugging
- gprof, GNU profiling tool
- CVS for source code concurrent versioning system
- Ogg Vorbis integer version from Dan Conti [4]
- Ogg Vorbis integer version from Nicolas Petre [23]
- LyX/LaTeX/TeX for editing/typesetting report

Commercial Software:

- TSIM LEON Simulator Professional Edition

Organization of the Software chapter

In this chapter the content is arranged as follows:

- Section 3.2: **Preliminary Analysis**, describes the feasibility study and preliminary analysis of Vorbis library.
- Section 3.3: **Modeling of the System**, contains information about the use of TSIM LEON simulator in the project, TSIM I/O module and emulation the Audio-Core as I/O module.
- Section 3.4: **Software Optimization**, describes the cross compilation process to SPARC platform of Ogg Vorbis library, creation of simpleplayer and the optimization and integerization of Vorbis library decoder part.
- Section 3.5: **Hardware/Software Partitioning**, covers the hardware/software partitioning process and the proposed partitions. At the end the statistics of simpleplayer with proposed partitions are shown.
- Section 3.6: **Development of Player**, discusses about the RTEMS real time operating systems, development of the audio device driver and the concept and implementation final player.
- Section 3.7: **Conclusion**, concludes all the work in software part and lists some problems found. Possible further improvement for the software part is also suggested.

Short summaries are provided at the end of each Section as brief description of tasks done in each phase.

3.2 Preliminary Analysis

3.2.1 Performance Observation on Linux PC

A simple performance observation of the reference Vorbis decoder library and the client music player *Ogg123* on a Linux PC gave preliminary performance information of Vorbis decoding process. Ogg123 is a multi-threaded feature-rich text-based Ogg Vorbis player. The complexity of Ogg123 is higher than what we expected to have in our Vorbis player so this estimation was slightly pessimistic.

The rough resource observation of Ogg123 was done through UNIX's *top* command while ogg123 was running (decoding the Ogg Vorbis music). The screen-shot of this is shown in Figure 3.3. The machine under test was equipped with Intel CPU Pentium-III 600 MHz, 256 MB RAM and Linux kernel 2.4.18. The information got from *top* (notice the line that contains ogg123) shows CPU usage of *ogg123* process around **4.6%** (%CPU field) and memory usage around **1.67 MB** (SIZE field).

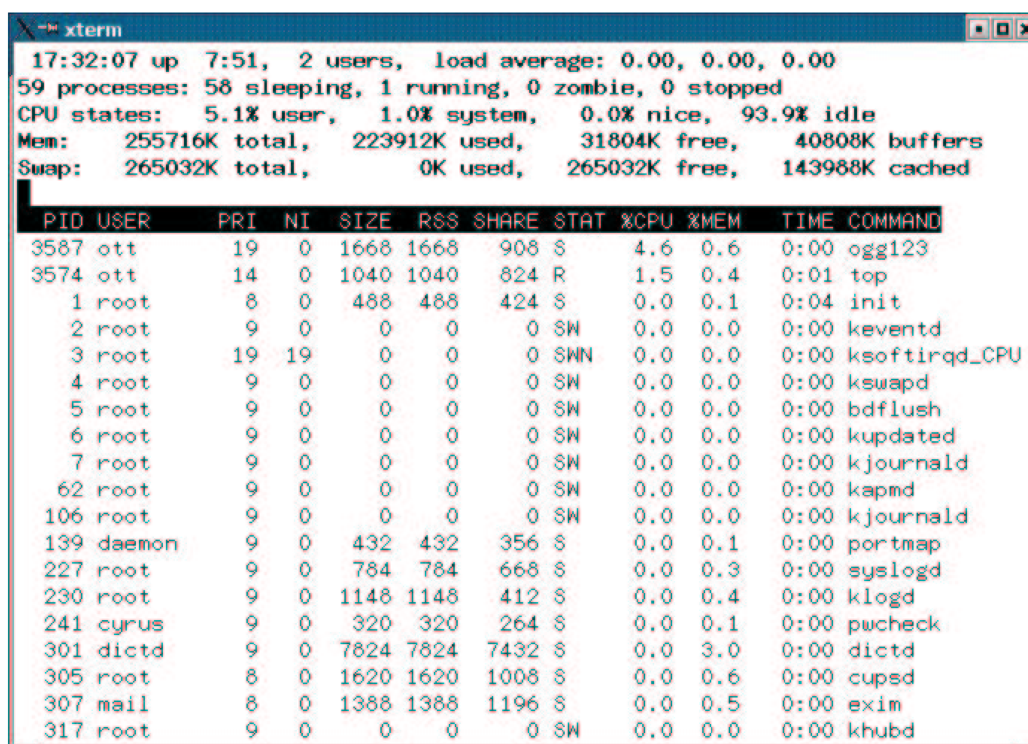

Figure 3.3: Screen-shot of top command

3.2.2 Memory Usage Analysis

With the *purify* tool from Rational Software Corporation [15], one can do real-time analysis of the software in detail e.g. identify execution errors or memory leaks. In our case, it was used for memory usage analysis. The simple Vorbis decoder (*decoder_example.c*) that came with the reference Vorbis library was tested with purify. No memory leak was found. Heap memory peak usage was around **880 kBytes** and stack size was 2,304 bytes. This information, however, included

the purify overhead. Here is the example got from purify running with the *decoder_example.c* code and example 15-second snapshot of music.

```
Basic memory usage (including Purify overhead):

819185 code
356772 data/bss 8
884736 heap (peak use)
2304 stack
```

Example screen-shot of purify tool is shown in Figure 3.4.

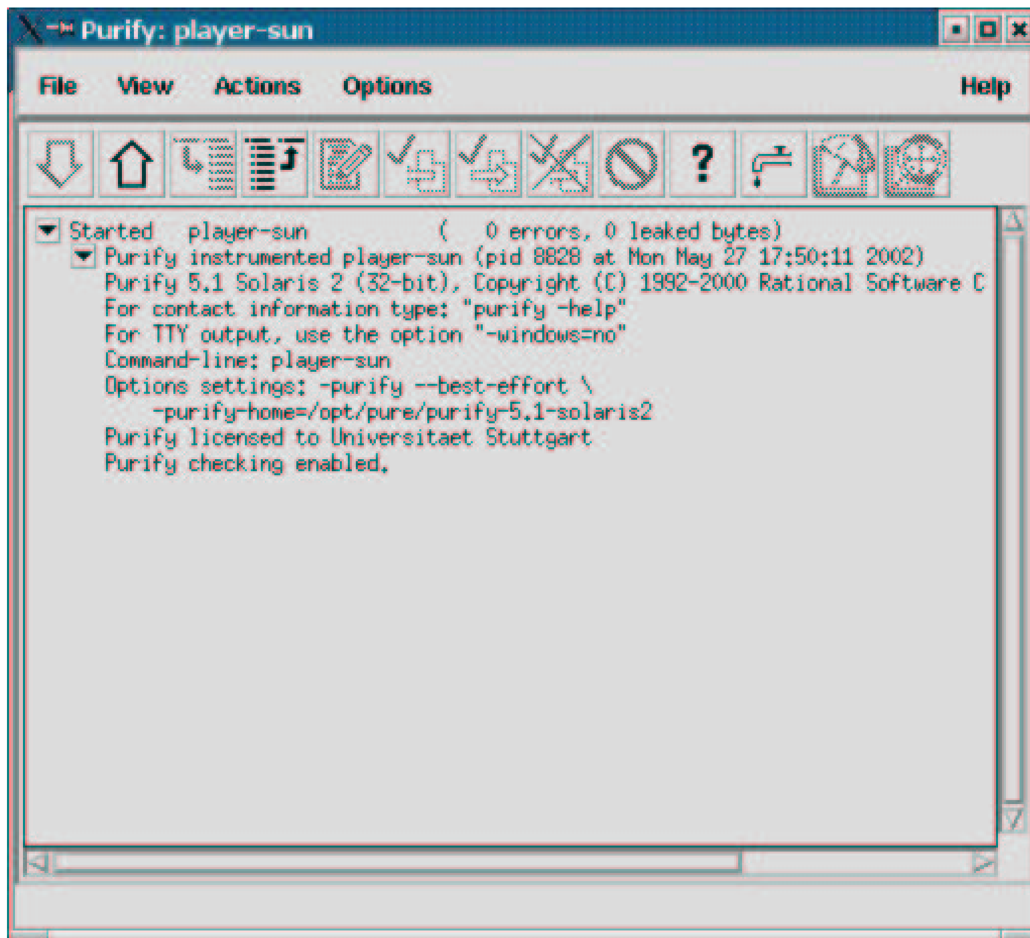

Figure 3.4: Screen-shot of *purify* tool

3.2.3 Analysis

For memory usage, *top* and *purify* gave similar result. *top* showed usage of about **1.67 MB** of memory while *purify* said almost 880k of heap was used, to which if we add the size of the application program (around 700 kB for *ogg123* static-linked version) then the result would be **1.58MB** (overhead of purify included). Though the player to be used in the project would be smaller than *ogg123* because the unnecessary parts would be removed and we would over-estimate this memory usage, however, we need to take also the memory usage of embedded operating system (RTEMS) and place for Ogg Vorbis music data to be played into account (as we might not have external storage for music data). So this could compensate the overhead of *ogg123* and *purify*. Taken **1.6 MB** as the estimated value of memory usage, the player should have no memory problem running on the real board as our target board has **2 MB** of RAM.

CPU analysis is more difficult. From *top* it was shown that the player took roughly **5%** of Intel Pentium-III 600 MHz CPU. This could be calculated to 5% * 600 = **30 MHz**. However, LEON processor has different architecture than Intel Pentium. Thus it is incorrect to compare directly using this MHz number. Better approximation could be obtained from information of the Vorbis player running on other embedded processors. Dan Conti reported in his e-mail sent to *vorbis-dev@xiph.org* mailing list[1] that Vorbis player ran successfully on Cirrus 7312-74 MHz processor (with ARM7TDMI core). Taken the worst case of 100% CPU usage, this could be roughly estimated to **74 MHz** for LEON CPU. Our target FPGA could run however at the highest speed of 25 MHz due to hardware limitations (more information in Chapter 4). That means roughly **60-70%** performance improvement [(74-25)/74 = 66] was needed.

3.2.4 Vorbis Library Profiling

As **60-70%** performance improvement was needed in order to have Vorbis stream decoded and played in real-time. This improvement could be done through software optimization (algorithm optimization, code optimization, changing compiler flag) and hardware optimization (tuning of LEON parameters, creating a new application-specific core for computation-intensive part). For hardware optimization, in order to create a new core, we need to locate first which part of the software is taking most of CPU time and deserves to be implemented in hardware. This could be done by profiling the software code and the run-time data could then be analyzed. The *gprof* tool from GNU project was chosen for this task.

Vorbis reference library and simple client music player program (*vorbisfile_example.c* that came with Vorbis library) were recompiled with the *-pg* flag for profiling pur-

[1] `http://www.xiph.org/archives/vorbis-dev/200202/0125.html`

pose. Profiling was done on Sun SPARC workstation as to have the same CPU architecture as LEON. Following are parts of the profile information of the simple Ogg Vorbis decoder. The flat profile shows how much time the program spent in each function and how many times the function was called. The call graph shows, for each function, which functions called it, which other functions it called, and how many times [14]. There is also an estimation of how much time spent in the subroutines of each function which is very useful to analyze further which function consumes a lot of computing power and should be implemented in hardware.

Flat profile

```
granularity:  each sample hit covers 4 byte(s) for 0.29% of 3.43 seconds

  %   cumulative   self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 16.0       0.55     0.55      944     0.58     2.94  mapping0_inverse [3]
 11.7       0.95     0.40    44448     0.01     0.02  vorbis_book_decodevv_add[7]
 10.2       1.30     0.35    38272     0.01     0.01  mdct_butterfly_generic [10]
  9.9       1.64     0.34     1888     0.18     0.56  mdct_backward [5]
  7.3       1.89     0.25      944     0.26     0.26  vorbis_synthesis_blockin[11]
  7.3       2.14     0.25                            internal_mcount [12]
  6.1       2.35     0.21    19671     0.01     0.01  render_line [14]
  5.0       2.52     0.17   595437     0.00     0.00  oggpack_adv [15]
  4.7       2.68     0.16     1888     0.08     0.08  mdct_bitreverse [16]
  4.4       2.83     0.15   597050     0.00     0.00  oggpack_look [17]
  3.5       2.95     0.12        1   120.00  3170.00  main [1]
  2.9       3.05     0.10     1888     0.05     0.05  mdct_butterfly_first [19]
  2.6       3.14     0.09   597050     0.00     0.00  vorbis_book_decode [9]
  2.0       3.21     0.07   168192     0.00     0.00  mdct_butterfly_8 [22]
  1.7       3.27     0.06     1888     0.03     0.05  floor1_inverse1 [21]
  1.2       3.31     0.04   126782     0.00     0.00  oggpack_read1 [23]
  0.9       3.34     0.03     1888     0.02     0.13  floor1_inverse2 [13]
  0.9       3.37     0.03      944     0.03     0.91  res2_inverse [6]
  0.6       3.39     0.02    84096     0.00     0.00  mdct_butterfly_16 [20]
  0.3       3.40     0.01    42048     0.00     0.00  mdct_butterfly_32 [18]
  0.3       3.41     0.01     1003     0.01     0.01  _packetout [27]
  0.3       3.42     0.01       48     0.21     0.21  _make_words [29]
```

Call Graph Profile [top 10 only]

```
                                  called/total       parents
index  %time    self descendents  called+self    name           index
                                  called/total       children

                0.12        3.05       1/1           _start [2]
[1]     92.7    0.12        3.05       1         main [1]
                0.00        2.78     944/944         vorbis_synthesis [4]
                0.25        0.00     944/944         vorbis_synthesis_blockin [11]
                0.00        0.01    1003/1003        ogg_stream_packetout [24]
                0.00        0.01       1/1           vorbis_synthesis_init [26]
```

```
                0.00    0.00  1887/1887      vorbis_synthesis_pcmout [34]
                0.00    0.00   943/943       vorbis_synthesis_read [36]
                0.00    0.00   118/118       ogg_sync_pageout [37]
                0.00    0.00    62/62        ogg_sync_buffer [40]
                0.00    0.00    62/62        readAudioData [42]
                0.00    0.00    62/62        ogg_sync_wrote [41]
                0.00    0.00    58/58        ogg_stream_pagein [50]
                0.00    0.00    56/114       ogg_page_eos [39]
                0.00    0.00     3/3         vorbis_synthesis_headerin
[58]
                0.00    0.00     1/2         ogg_sync_init [70]
                0.00    0.00     1/59        ogg_page_serialno [43]
                0.00    0.00     1/1         ogg_stream_init [79]
                0.00    0.00     1/1         vorbis_info_init [90]
                0.00    0.00     1/1         vorbis_comment_init [87]
                0.00    0.00     1/1         vorbis_block_init [85]
                0.00    0.00     1/1         ogg_stream_clear [78]
                0.00    0.00     1/1         vorbis_block_clear [84]
                0.00    0.00     1/1         vorbis_dsp_clear [88]
                0.00    0.00     1/1         vorbis_comment_clear [86]
                0.00    0.00     1/1         vorbis_info_clear [89]
                0.00    0.00     1/1         ogg_sync_clear [80]

  -----------------------------------------------

                                             <spontaneous>
  [2]    92.7   0.00    3.17               _start [2]
                0.12    3.05     1/1         main [1]

  -----------------------------------------------

                0.55    2.23   944/944       vorbis_synthesis [4]
  [3]    81.3   0.55    2.23   944         mapping0_inverse [3]
                0.34    0.71  1888/1888      mdct_backward [5]
                0.03    0.82   944/944       res2_inverse [6]
                0.03    0.21  1888/1888      floor1_inverse2 [13]
                0.06    0.03  1888/1888      floor1_inverse1 [21]

  -----------------------------------------------

                0.00    2.78   944/944       main [1]
  [4]    81.3   0.00    2.78   944         vorbis_synthesis [4]
                0.55    2.23   944/944       mapping0_inverse [3]
                0.00    0.00  3120/16560     oggpack_read [32]
                0.00    0.00  2832/5628      _vorbis_block_alloc [283]
                0.00    0.00   944/945       _vorbis_block_ripcord [284]
                0.00    0.00   944/947       oggpack_readinit [35]

  -----------------------------------------------

                0.34    0.71  1888/1888      mapping0_inverse [3]
  [5]    30.7   0.34    0.71  1888         mdct_backward [5]
                0.00    0.55  1888/1888      mdct_butterflies [8]
                0.16    0.00  1888/1888      mdct_bitreverse [16]

  -----------------------------------------------

                0.03    0.82   944/944       mapping0_inverse [3]
  [6]    25.0   0.03    0.82   944         res2_inverse [6]
                0.40    0.41 44448/44448     vorbis_book_decodevv_add [7]
                0.00    0.01 20586/597050    vorbis_book_decode [9]
                0.00    0.00   944/5628      _vorbis_block_alloc [283]
```

```
-----------------------------------------------

                0.40        0.41   44448/44448       res2_inverse [6]
[7]     23.7    0.40        0.41   44448         vorbis_book_decodevv_add [7]
                0.08        0.33  542525/597050      vorbis_book_decode [9]

-----------------------------------------------

                0.00        0.55    1888/1888        mdct_backward [5]
[8]     16.1    0.00        0.55    1888         mdct_butterflies [8]
                0.35        0.00   38272/38272       mdct_butterfly_generic [10]
                0.01        0.09   42048/42048       mdct_butterfly_32 [18]
                0.10        0.00    1888/1888        mdct_butterfly_first [19]

-----------------------------------------------

                0.00        0.01   20586/597050      res2_inverse [6]
                0.01        0.02   33939/597050      floor1_inverse1 [21]
                0.08        0.33  542525/597050      vorbis_book_decodevv_add [7]
[9]     13.2    0.09        0.36  597050         vorbis_book_decode [9]
                0.17        0.00  595437/595437      oggpack_adv [15]
                0.15        0.00  597050/597050      oggpack_look [17]
                0.04        0.00  126782/126782      oggpack_read1 [23]

-----------------------------------------------

                0.35        0.00   38272/38272       mdct_butterflies [8]
[10]    10.2    0.35        0.00   38272         mdct_butterfly_generic [10]
```

From the above profile information, the high computation intensive parts of Vorbis decoder are:

- *inverse MDCT* (*mdct_backward()* function and subroutines) with **30.7%** of total computation time
- residue and codebook operation (starting from *res2_inverse()* and subroutines) with **25%** of computation time
- windowing and other multiplications in *mapping0_inverse()* which is the main decode loop (81.3 - 30.7 - 25.0 - 7 - 2.5 = **16.1%**)

We focused first at the *mdct_backward()* function. This piece of code (in *mdct.c*) was found to be possible to be implemented in hardware. Other parts of code were not so independent and hardware-friendly as *mdct_backward()*.

3.2.5 Summary of Preliminary Analysis

60-70% speed improvement was expected in order to have the player played the Ogg Vorbis stream at real-time speed on the target hardware. By considering the possibility to implement a part of software as hardware to achieve about **30-40%** improvement (*mdct_backward()* function was targeted) and apply possible

software optimizations for other part of code to get also **30-40%** improvement, we considered the project as feasible and the project was proceeded.

3.3 Modeling of the System

In this Section, work in the *modeling of the system* phase is described. TSIM simulator is introduced and its use in the project is shown. Modeling of hardware audio core as TSIM I/O module is discussed and illustrated.

3.3.1 TSIM – LEON Simulator

As discussed in chapter 3.1, testing every software pieces directly on the real hardware board is not efficient because the process to test the code on the real hardware (cross-compiling and downloading of the software to the hardware board) consumes a lot of time. Gaisler research developed a simulator of SPARC architecture capable of simulating LEON called *TSIM*. TSIM runs on Linux-x86, SPARC, and Windows/Cygwin platforms. Developers could then develop, test and debug their programs on TSIM running on their workstations without having to touch any real hardware. This speeded up dramatically the development process. Figure 3.5 shows a screen-shot of TSIM running on a Linux PC. TSIM can be connected with GNU Debugger *gdb* to enable debugging.

Gaisler Research also provided LECCS (LEON/ERC32 Cross Compilation System), a complete set of software tools for development on LEON platform based on GNU tools e.g. GNU C/C++ compiler, , DDD debugger, binutils, newlib C-library, RTEMS operating system, and boot-prom utility, to let developers use the tools immediately without having to face the cumbersome compiling process of each tool. Tools in LECCS were used in our project most of the time except some rare cases that recompilation of some tools were needed.

Apart from LEON CPU, the user-defined I/O hardware-core can also be written as I/O module to emulate the behavior of that hardware. In the project, firstly the audio core hardware module (or so called "DDM") from [2] was written as a TSIM I/O module. In the hardware/software partitioning process, the selected software piece that was planned to be developed as hardware was first written also as TSIM I/O module and tested for performance gain before we really designed and implemented on the real hardware.

3.3.2 Audio Core as TSIM I/O Module

TSIM and I/O module work cooperatively in a way that if there is a read or write access to a certain address which falls in the I/O address range, it will be for-

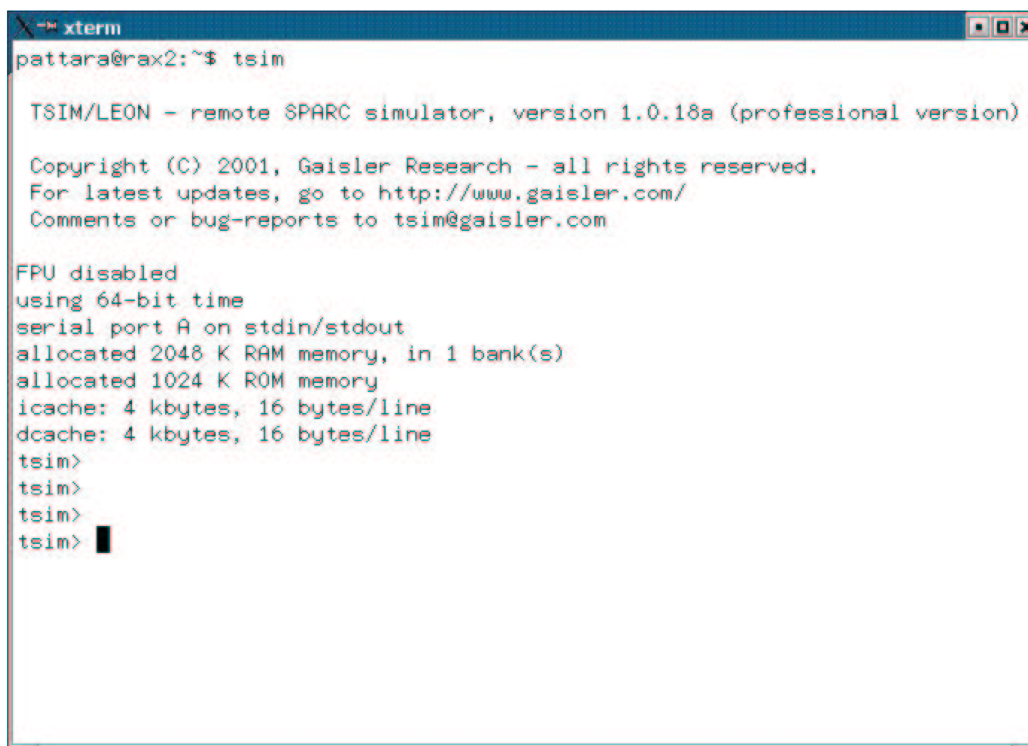

Figure 3.5: Screen-shot of TSIM running on Linux PC

warded to I/O module (i.e. functions in I/O module will be called), otherwise TSIM processes that request without interacting with the module. I/O address ranges (for TSIM 1.0.18) are:

```
0x10000000 - 0x3fffffff
0x80000100 - 0xffffffff
```

I/O module can also generate interrupts and DMA requests.

Before we can write I/O module such that it emulates the behavior of audio core, the functions and characteristics of audio core itself need to be studied. Structure and functions of audio hardware core was described in [2]. Usage of audio core can be done through 7 registers as access point. Figure 3.6 illustrates these registers.

These register interface was mapped to C-language structure *audio_core_regs* in TSIM I/O module as follows:

```
struct audio_core_regs {
  volatile unsigned int controlreg; /* 0x00 */
  volatile unsigned int startaddr;  /* 0x04 */
```

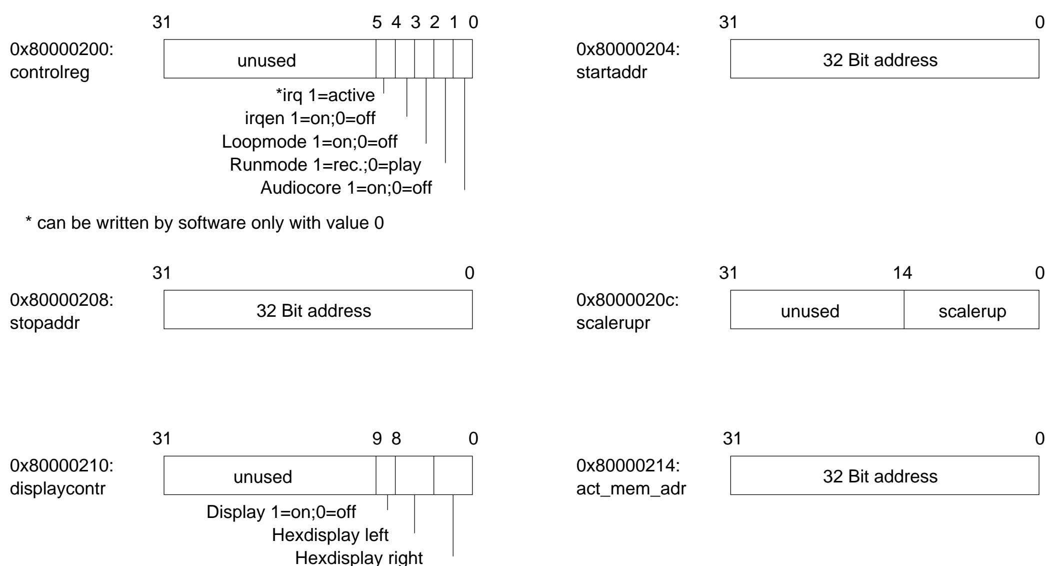

Figure 3.6: Audio core registers (translated from [2])

```
    volatile unsigned int stopaddrr;  /* 0x08 */
    volatile unsigned int scalerupr;  /* 0x0C */
    volatile unsigned int displcontr; /* 0x10 */
    volatile unsigned int buttonreg;  /* 0x14 */
    volatile unsigned int act_mem_adr;/* 0x18 */
};

#define AUDIO_CORE_START    0x80000200
#define AUDIO_CORE_END      0x8000021C

#define AUDIO_CORE_SIZE     0x1C

#define AUDIO_CORE_CONTROLREG  0x00
#define AUDIO_CORE_STARTADDR   0x04
#define AUDIO_CORE_STOPADDRR   0x08
#define AUDIO_CORE_SCALERUPR   0x0C
#define AUDIO_CORE_DISPLCONTR  0x10
#define AUDIO_CORE_BUTTONREG   0x14
#define AUDIO_CORE_ACT_MEM_ADR 0x18
```

The audio core resides at address `0x80000200`, corresponding to that on the real hardware. This address is in the I/O address range for I/O mentioned before. There are 7 registers, each of size 4 bytes (*unsigned int*). Therefore, audio core occupies address range from `0x80000200` - `0x8000021C`. Other hardware cores (e.g. MDCT core in the later chapter) utilize other ranges of addresses.

Audio core TSIM I/O module behaves similarly to audio core hardware as described in [2]. Application can make use of the core by storing the audio data in the memory, writing start and stop addresses to *startaddr* and *stopaddrr* registers accordingly, and writing last bit of control register *controlreg* to '1'. These will trigger the core to do the DMA transfer of music data from memory starting from *startaddr* to *stopaddrr*, start to play that music content and send an interrupt when finished.

Upon playing the music content, the core (as emulated by TSIM I/O module) first checks whether the audio device of host PC which TSIM is running on can be opened, if yes, the music will be played directly through audio device of host computer, otherwise, the content will be saved to a file *audioout.raw* on the current directory. OSS (Open Sound System) audio application programming interface [32] was used for communicating with host PC's audio device. Example of a write access (write value of 1 to *controlreg* register) from application to audio core TSIM I/O module is illustrated in Figure 3.7.

The audio I/O module worked with 44 kHz-sampling rate, 16-bit unsigned, 2-channel stereo audio format as default. Changing of sampling frequency can be done by writing values to *scalerupr* register (0, 1, 2 for 44 kHz, 22 kHz, 11 kHz frequency respectively). Changing number of channels or bit per sample was not yet possible in the version. There are some functions of the real hardware core that were not implemented in I/O module as they were not required in the project, e.g. looping feature, updating of the *act_mem_adr* register upon playing the audio data.

Care must be taken when running TSIM on the x86-compatible PC as TSIM emulates SPARC and thus conforms to the big endian byte ordering (most significant bit at the lowest address), while the x86 PC is little endian (least significant bit at the lowest address). In our project, the song file stored on harddisk of the PC needed to be converted to big-endian format before it could be fed to the player program that ran on TSIM. Another difference between x86 and SPARC architecture is that memory accesses in SPARC architecture will be always in a full 32-bit word only, no 16-bit or 8-bit access.

The implementation of TSIM I/O module was done modularly. The main *io.c* file acts as task distributor, after checking the address ranges of accesses, it will call other appropriate files, e.g. *audio_core.c*, *mdct_core.c* (as later in the project there were MDCT module also, not only audio core module).

Compilation of TSIM modules is as follows:

```
<CVS>/software/tsim_io$ gcc -Wall -W -fPIC -c -O2 io.c -o io.o
<CVS>/software/tsim_io$ gcc -Wall -W -shared io.o -o io.so
```

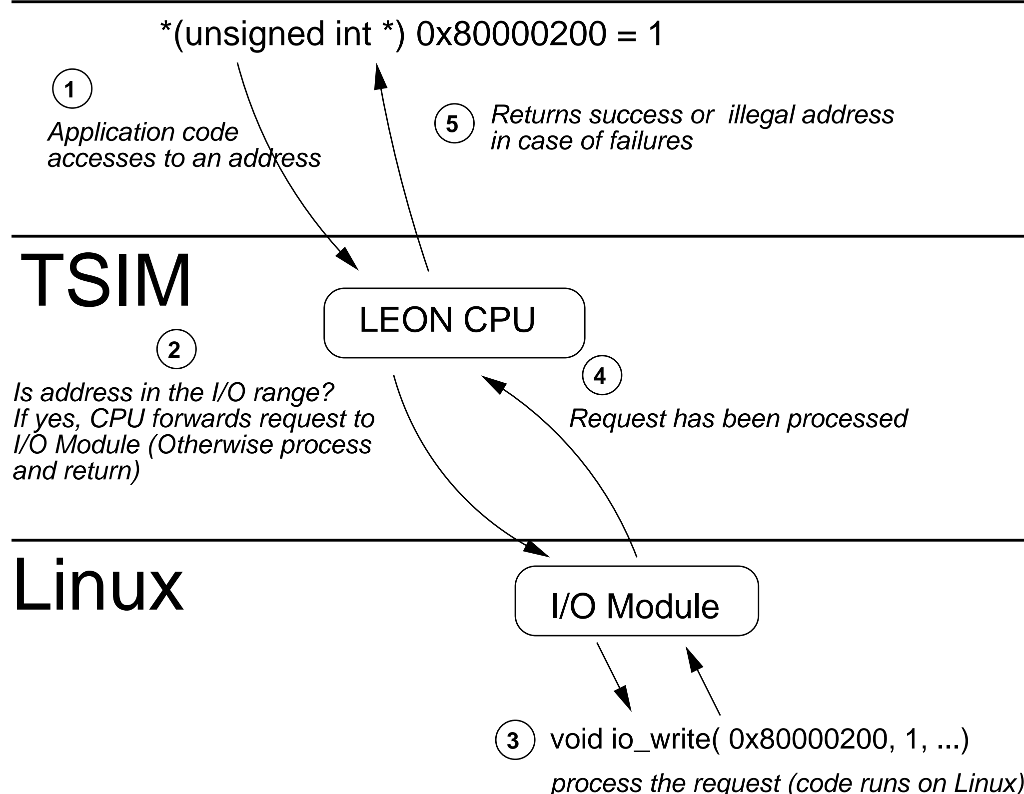

Figure 3.7: Example of a write access to audio core TSIM I/O module

The module needs to be compiled as a shared library named *io.so* and placed under the current working directory of TSIM. Upon executing TSIM, the *io.so* will be automatically loaded.

TSIM I/O module feature largely aided our development process. Huge amount of software bugs were discovered and solved by testing on TSIM without bothering the real hardware.

3.3.3 Test of Audio Core TSIM I/O Module

After the audio core has been written as I/O module, a test program to output raw music via this module was written in order to test the functionality of this audio core module. Test suite contains static raw music data with various sampling frequencies (11kHz, 22kHz or 44 kHz). The music was played via audio core I/O module correctly.

After that we tested the same program on the real hardware board. This was at the same time that the audio hardware core (initially imported from [2]) has been modified to support 2-channel stereo and 16-bit sample size. The test program has been cross-compiled to SPARC platform using GNU C cross-compiler (sparc-rtems-gcc) and converted the proper format to be loaded to the hardware development board using *mkprom* from LECCS package and *objcopy*, *clean_srec* utilities. These steps of compiling and converting are shown below. The test program name is *test.c* and the final output to be loaded to hardware is *test-soft.exo*.

Compiling the test program

```
<CVS>/software/audio_core$ sparc-rtems-gcc -static -O2 -W -Wall test.c -o test
```

Convert to appropriate format to be loaded to hardware

```
<CVS>/software/audio_core$ ../utils/genexo.sh test

+ mkprom -nocomp -v -o test.rom -ramsize 2048 -ramcs 1 -rmw -ramrws 0
-ramwws 0 -freq 25 -baud 38400 -romws 2 -romsize 1024 test

MKPROM boot-prom builder v1.3.5
Copyright Gaisler Research 2001, all rights reserved.

loading test:
section:  .text at 0x40000000, size 32080 bytes
section:  .data at 0x40007d50, size 681552 bytes

creating LEON boot prom:  test.rom

sparc-rtems-gcc -O2 -g -N -qprom -qzero -nostartfiles -Xlinker -Ttext
-Xlinker 0 /opt/rtems/sparc-rtems/lib/promcorel.o dump.s -lmkproml -o
test.rom
+ sparc-rtems-objcopy --adjust-vma=0x100000 -O srec test.rom test.exo

+ clean_srec
+ set +x
Clean up ...
Done...
```

After that the test program was tested on TSIM.

```
$ tsim-leon -nfp -freq 25 test.rom

 TSIM/LEON - remote SPARC simulator, version 1.0.18a (professional version)

 Copyright (C) 2001, Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to tsim@gaisler.com

FPU disabled
using 64-bit time
serial port A on stdin/stdout
allocated 2048 K RAM memory, in 1 bank(s)
```

```
allocated 1024 K ROM memory
icache:  4 kbytes, 16 bytes/line
dcache:  4 kbytes, 16 bytes/line
Enter IO Module:  time = 0
io:  AUDIO_CORE: Audio-core initialized at 0x80000200
io:  AUDIO_CORE: time = 0
io:  AUDIO_CORE: Open /dev/dsp failed, output to file audioout.raw instead...
section:  .text at 0x0, size 716416 bytes
tsim> go
resuming at 0x00000000


  MkProm LEON boot loader v1.2
  Copyright Gaisler Research - all right reserved

  system clock   :  25.0 MHz
  baud rate      :  38580 baud
  prom           :  1024 K, (2/2) ws (r/w)
  ram            :  2048 K, 1 bank(s), 0/0 ws (r/w)
  edac           :  disabled

  loading .text
  loading .data

  starting test
Sound test program.  Sampling rate = 22050, Bit-per-sample = 16, 2-channel
Number of sound samples found in music data:  339826
Allocated memory of 1359304 bytes for sound buffer
Loading music data to memory ...
Done...
Now start to play the sound ...
io:AUDIO_CORE:  Hex displays '55'.

Program exited normally.
```

Note that from the above example, the I/O module could not open audio device (*/dev/dsp*) of host machine, therefore it wrote the output to the file *audioout.raw* instead. The music content in the file can be played by using the *play* tool from Sox sound utility [1] as below:

```
$ play -r 22050 -s w -f s -c 2 audioout.raw
```

The description [12] for each option is: *-r 22050* sets sampling frequency to 22050 Hz (as this test case used 22.05 kHz as sampling frequency), *-s w* means each sound sample will be 16-bit large (word), *-f s* means the value of each sound sample is stored in unsigned format and *-c 2* indicates 2-channel stereo music.

The output *test-soft.exo* file was uploaded to the development board and tested. The result was the same as on TSIM. Thus we have modeled and implemented this audio core as TSIM I/O module to simulate all required functionalities of the real audio hardware core. This module was used further throughout the project.

All further tests with TSIM in the project used the default setting: 2 MB RAM, 1 MB ROM, 4kB for both data and instruction cache, and clock frequency of 25 MHz which corresponds to those on the real hardware.

3.3.4 Summary

In this phase, the system was modeled on the software using TSIM LEON simulator. The audio core imported from DDM [2] was successfully written as TSIM I/O module to simulate the function of hardware audio core. Test has been conducted to make sure that TSIM and its I/O module correctly simulated the required functionalities presented by the real hardware.

3.4 Software Optimization

After we had modeled the system using TSIM simulator, testing of Vorbis decoder was done by developing and running a *simpleplayer* test program on this simulated platform. Vorbis library was further optimized. This Section describes the work phase of *software optimization* according to Figure 3.1.

3.4.1 Cross Compilation of Ogg/Vorbis Library to SPARC Platform

Ogg Vorbis reference library used the GNU Autoconf tool [13]. Cross-compiling to SPARC platform was trivial. Theoretically passing the appropriate CC environment variable to `./configure` script and adding `--disable-shared` option (because the sparc-rtems-gcc that came with LECCS did not support shared library) should accomplish the task. Practically however we found that Ogg library's `configure` script failed to find the size of each integer variable type so extra lines in `configure.in` file were added as follows:

```
AC_CHECK_SIZEOF(short,2)
AC_CHECK_SIZEOF(int,4)
AC_CHECK_SIZEOF(long,4)
AC_CHECK_SIZEOF(long long,8)
```

And then the `./configure` script was called:

```
CC=sparc-rtems-gcc ./configure --disable-shared && make
```

This configure line worked for Vorbis library too. However, in the project this GNU Autoconf tool was not used, the Makefiles were customly created as to test

with various compile options to match our environment. See the software source code for more details.

After compiling both Ogg and Vorbis, we got three main libraries to be used in the project: *libogg.a, libvorbis.a* and *libvorbisfile.a*.

3.4.2 Simpleplayer Test Program

Test program *simpleplayer* has been written to test Vorbis library. Sample program that came with Vorbis library *decoder_example.c* has been used as starting-point. Simpleplayer decodes the sample Vorbis stream data and writes PCM output to standard-out or optionally discards the output. It can verify checksum of audio output in order to check the accuracy of decoding process.

Sample Vorbis stream data used throughout this project was 15.40 seconds length with 44 kHz sampling frequency, 16-bit per sample, 2-channel stereo and was encoded with Ogg Vorbis encoder from reference Vorbis library version 1.0RC3 with quality level 3 (default level).

Simpleplayer was cross-compiled to SPARC platform using *sparc-rtems-gcc* compiler from LECCS package. It ran without operating systems on LEON. On TSIM at 25 MHz clock speed, simpleplayer ran and completed its decoding job of sample Vorbis stream in **32.90** seconds. Simpleplayer prints out the activities e.g. percentage of input data to standard output.

Here is the example of compilation and running of simpleplayer on TSIM. Notice that *per* command at the end shows total simulated time used, processor utilization and other performance-related data.

```
<CVS>/software/simpleplayer$ make player-leon
sparc-rtems-gcc -static -mv8 -O2 -o player-leon player.c
-I../vorbis/include/ -L../vorbis/rtems-fpu-normal/ -lvorbis -logg -lm

$ ../utils/genexo.sh player-leon
+ mkprom -nocomp -v -o player-leon.rom -ramsize 2048 -ramcs 1 -rmw
-ramrws 0 -ramwws 0 -freq 25 -baud 38400 -romws 2 -romsize 1024
player-leon

MKPROM boot-prom builder v1.3.5
Copyright Gaisler Research 2001, all rights reserved.

loading player-leon:
section:  .text at 0x40000000, size 163792 bytes
section:  .data at 0x40027fd0, size 256512 bytes

creating LEON boot prom:  player-leon.rom

sparc-rtems-gcc -O2 -g -N -qprom -qzero -nostartfiles -Xlinker -Ttext
-Xlinker 0 /opt/rtems/sparc-rtems/lib/promcorel.o dump.s -lmkproml -o
player-leon.rom

+ sparc-rtems-objcopy --adjust-vma=0x100000 -O srec player-leon.rom
player-leon.exo
```

```
+ clean_srec
+ set +x
Clean up ...
Done...

$ tsim-leon -ram 2048 -rom 2048 -freq 25 player-leon.rom
 TSIM/LEON - remote SPARC simulator, version 1.0.18a (professional version)

 Copyright (C) 2001, Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to tsim@gaisler.com

using 64-bit time
serial port A on stdin/stdout
allocated 2048 K RAM memory, in 1 bank(s)
allocated 2048 K ROM memory
icache:  4 kbytes, 16 bytes/line
dcache:  4 kbytes, 16 bytes/line
Enter IO Module:  time = 0
io:  AUDIO_CORE: Audio-core initialized at 0x80000200
io:  AUDIO_CORE: time = 0
io:  AUDIO_CORE: Open /dev/dsp failed, output to file audioout.raw instead...
section:  .text at 0x0, size 423072 bytes
tsim> go
resuming at 0x00000000


  MkProm LEON boot loader v1.2
  Copyright Gaisler Research - all right reserved

  system clock   :  25.0 MHz
  baud rate      :  38580 baud
  prom           :  1024 K, (2/2) ws (r/w)
  ram            :  2048 K, 1 bank(s), 0/0 ws (r/w)
  edac           :  disabled

  loading .text
  loading .data

  starting player-leon
Input data read:   1%    Time:    0        Diff:    0
Input data read:   3%    Time:    0        Diff:    0

Bitstream is 2 channel, 44100Hz
Encoded by:  Xiphophorus libVorbis I 20011231

Input data read:   5%    Time:    0        Diff:    0
Input data read:   6%    Time:    1        Diff:    1
Input data read:   8%    Time:    2        Diff:    1
Input data read:  10%    Time:    2        Diff:    0
Input data read:  11%    Time:    3        Diff:    1
Input data read:  13%    Time:    4        Diff:    1
Input data read:  15%    Time:    4        Diff:    0
Input data read:  16%    Time:    5        Diff:    1
Input data read:  18%    Time:    5        Diff:    0
Input data read:  20%    Time:    6        Diff:    1
Input data read:  21%    Time:    7        Diff:    1
Input data read:  23%    Time:    7        Diff:    0
Input data read:  25%    Time:    8        Diff:    1
Input data read:  26%    Time:    8        Diff:    0
Input data read:  28%    Time:    9        Diff:    1
```

```
Input data read:  30%    Time:   10        Diff:    1
Input data read:  31%    Time:   10        Diff:    0
Input data read:  33%    Time:   11        Diff:    1
Input data read:  35%    Time:   11        Diff:    0
[...]
Input data read:  97%    Time:   16        Diff:    1
Input data read:  98%    Time:    0        Diff:  -16
Input data read:  100%   Time:    0        Diff:    0
Input data read:  100%   Time:    0        Diff:    0
Input data read:  100%   Time:    0        Diff:    0
Done.
Program exited normally.
tsim> per
CPU performance (25.0 MHz) :  8.18 MOPS ( 6.66 MIPS, 1.53 MFLOPS)
Cache hit rate (inst/data) :  99.3 / 36.7 %
Simulated time : 32.90 s
Processor utilisation :  100.00 %
Real-time performance :  27.14 %
Simulator performance :  2220.68 KIPS
Used time (sys + user) :  121.24 s
```

3.4.3 Vorbis Optimization

From the result of simpleplayer test, it took 32.90 seconds to play the sample Ogg Vorbis stream music which means that (32.90 - 15.40)/32.90 = **53.19 percent** speed improvement from both software and hardware optimizations was needed in order to have music played in real-time. In this sub-Section, Vorbis optimization is discussed. In order to do the software optimization, one needs to understand how Vorbis decoding works. Because Xiph did not yet publish the specification of Vorbis, study of Vorbis algorithm was done with the help of the information from the Internet and from the software code. Graphical call graph was one of a useful tool we used to analyze and understand Vorbis source code.

Graphical Call Graph Generation

A graphical representation of function calls or *graphical call graph* aids the process of software code investigation because it shows clearly how functions call other functions and how they are called. Profile information generated by gprof mentioned in Section 3.2 was processed further by *kprof [24]* and *VCG tool [27]* to generate graphical call graph. Example of a graphical call graph of Vorbis decoder is shown in Figure 3.8. This graph only contains nodes starting from *vorbis_synthesis()* function which is the main decoding tree (consumes 81.3% of total computation time).

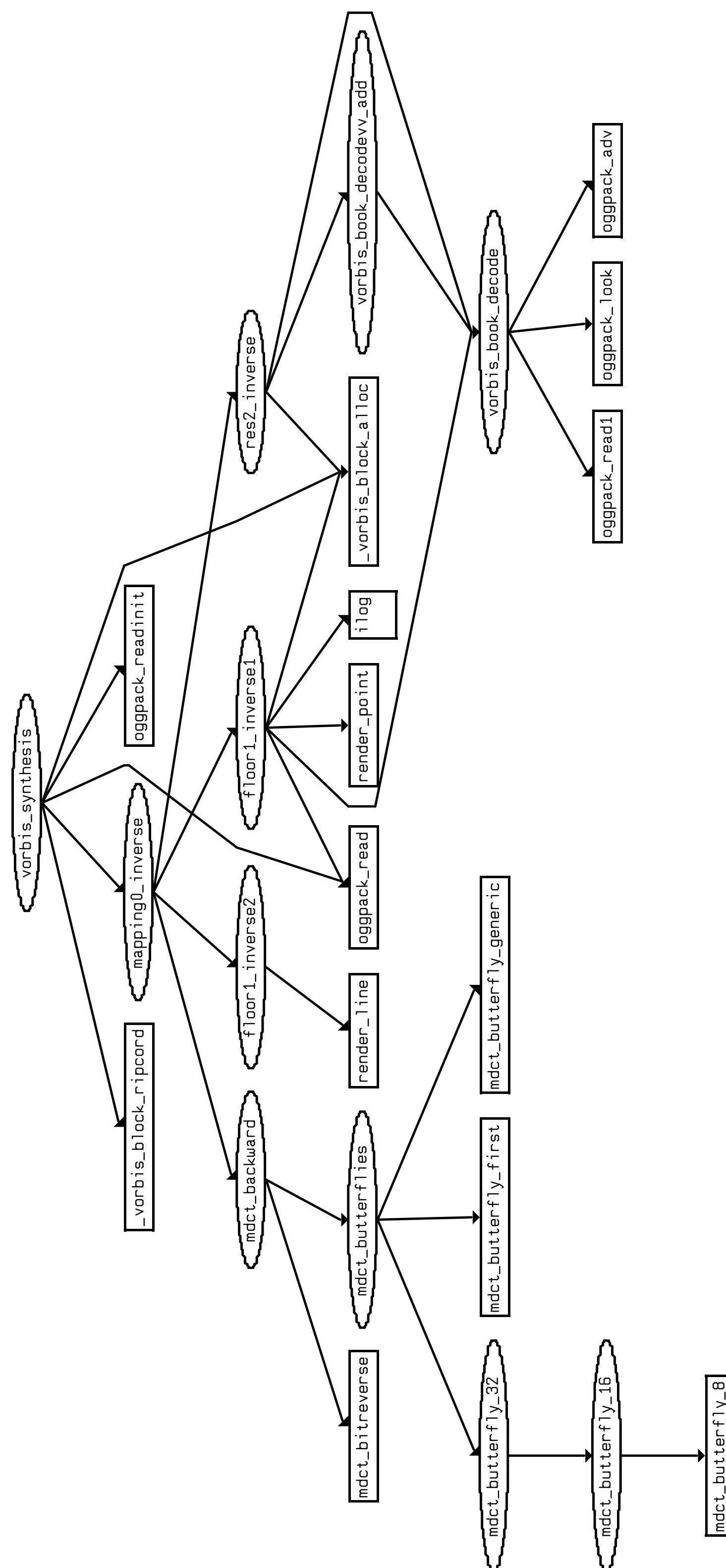

Figure 3.8: Graphical call graph starting from *vorbis_synthesis()*

With call graphs and profile information from the feasibility study phase, the structure of Vorbis algorithm was studied and the result is summarized in Chapter 2. During this study phase, it was noticed that this reference Vorbis library has a lot of calculations done in floating point. This triggered the need of an FPU (floating point unit) in the project. LEON itself came with primitive floating point unit (FPU) but it had the external FPU interface which could be connected to other FPU design. However, having an FPU was not desired in the project because it consumed a lot of space on the FPGA and increased the complexity of the design (more information in Chapter 4). To avoid having the FPU, floating point operations in Vorbis library must be converted to integer. The rest floating point usage must be kept to minimum as without FPU, all floating point instructions will be emulated using integer instruction and executed on Integer unit which would be much slower than running natively on FPU.

Integer Vorbis Decoder Library

At the time of the project, there was only one freely available integer implementation of Vorbis library done by Dan Conti [4][2]. This version was rather old and could not play Ogg Vorbis files encoded by newer (more recent than beta4) Vorbis encoder. Therefore, we aimed to create our own integer version of Vorbis decoder library based on the reference Vorbis library version 1.0 RC3 and Dan Conti's library with the following goals:

- **Most major calculations must be converted to integer.** The rest floating-point operations must be very few.
- **Speed improvement**. It should run 20-30% faster than the floating-point implementation.
- **Some degraded quality is tolerable.** Sound quality can be lower than floating-point version but the difference should NOT be easily perceivable by human with moderate audio output equipment (small speaker, headphone).

The integerizing work has been started and successfully accomplished. Affected parts are de-windowing, de-overlapping and main decode loop in *mapping0_inverse()*. There were only few uses of floating point left e.g. codebook extraction.

Our integerized version of Vorbis library was tested and compared with the floating-point version on TSIM. The result is shown below:

[2]Later in April/May 2002, there were two more integer implementations of Vorbis library available. Xiph itself offered the high-quality commercial integer version and Nicolas Petre [23] published the GPL version based on reference Vorbis decoder.

Table 3.1: Result of simpleplayer tests on TSIM at 25 MHz

Program	Result (s)
simpleplayer + floating point Vorbis library	32.90
simpleplayer + integerized Vorbis library	19.42

According to the result from TSIM, the integer version ran faster than the original floating point version (32.90 - 19.42) / 32.90 = **40.97%**. Because this integer version stores intermediate data using small number of bits (15 bits), the output result was not numerically the same as that from the original floating point decoder, although the difference was not easily perceivable by observers. Higher accuracy from Integer decoder can be obtained by using more bits in calculation. However, it would consume more running time and would not run in real-time in our environment.

More slightly software optimizations have been done through compiler flags adjustment and inline-ing of functions. As this integer version of Vorbis decoder matched our goals defined above completely, it was used further in the project.

3.4.4 Summary

In this phase, Ogg Vorbis decoder library has been cross-compiled to SPARC platform and the *simpleplayer* client program was created to test the decoding of Vorbis stream data on TSIM target. It decoded 15-second encoded Vorbis stream in 32.90 seconds. Vorbis library has been optimized and integerized so that no hardware floating-point unit was needed. The result integerized version of Vorbis library gave **40.97%** better performance than the original floating-point version. With this integerized version, the simpleplayer decoded 15-second music in **19.42** seconds.

3.5 Hardware/Software Partitioning

3.5.1 Hardware Limitations

Most work in software part has been done on TSIM simulator which we can configure it to have many settings (clock frequency, cache size, etc.) but not all of them are possible on the real hardware. Here we list the hardware limitations (more information in Chapter 4):

- **FPGA limitation:** due to the speed of our FPGA (Virtex XCV-800-4) chip, LEON can run at the maximum clock frequency of 28 MHz.

- **Development board limitation:** The internal oscillator on the board can generate only some certain frequencies, i.e. 10, 11.1, 12.5, 14.29, 16.67, 20, 25, 33, and 50 MHz. Running at the frequency in between (e.g. 28 MHz) requires external oscillators.
- **Meiko FPU limitation:** The design of FPU from Sun's Meiko SPARC implementation can run with maximum frequency of 25 MHz. And FPU consumes a large amount of space on the FPGA.

Due to these limitations, the conclusion was drawn:

- **No FPU:** As we already had the integerized version of Vorbis decoder library (which is also much faster than the floating point version), the need for an FPU was eliminated.
- **No external oscillator:** To avoid possible problems and complexity from external oscillator, we targeted to run the board with its internal oscillator at **25 MHz**.

From previous Section, with the integer version of Vorbis library, the simple-player took **19.42 seconds** to play 15-second (15.40 seconds to be exact) music. In order to have this player run in real time, we would need to run TSIM at (19.42/15.40)*25 = **32 MHz**. But according to the hardware limitation above, we could run the board at only **25 MHz**, therefore extra hardware optimization for at least (19.42-15.40)/19.42 = **20.7%** was needed.

3.5.2 Hardware/Software Partitioning

From the profile information in chapter 3.2, we have seen that inverse MDCT function starting from *mdct_backward()* (or in short – MDCT) consumes **30.7%** of computation time and is also hardware-friendly. Ogg Vorbis uses currently either small (256 samples) or large (2048 samples) data block sizes. From the sample Ogg Vorbis stream data of 15-second music, there are total 1,888 calls to *mdct_ward()* function, 1,232 of which are the calls with large block size, the rest are with small block size. Table 3.2 shows time needed for *mdct_backward()* function for both small and large block sizes.

Table 3.2: Statistics for *mdct_backward()* function

mdct_backward() version	Clock Cycles needed	Time (milliseconds)
software, small block size	14,600	0.584
software, big block size	180,250	7.21

From table 3.2, each call to *mdct_backward()* for big block size consumes 7.21 ms, this leads to total time spent in *mdct_backward* function of 1,232*7.21 ms = 8.88 seconds. Therefore this function was a good candidate to be implemented as hardware to improve the calculation speed. Two choices of partitions have been focused:

- **MDCT:** the whole *mdct_backward()* function tree
- **Mini-MDCT:** major part of *mdct_backward()* function tree including the code in *mdct_backward()* from the beginning to *mdct_butterfly_generic()* function. This does not include *mdct_butterfly_32()*, *mdct_bitreverse()* and afterwards. The Mini-MDCT covers roughly 60-70% of the whole MDCT.

Both partitions were tested by implementing them as a TSIM I/O module and observing the performance gain. Modeling these MDCT and Mini-MDCT cores as TSIM I/O module was done similarly to that of audio core in Section 3.3. Firstly the interface between hardware and software needs to be designed by looking at the information required to be transfered between them. The interface of *mdct_backward()* in Vorbis library is:

```
void mdct_backward(mdct_lookup *init, DATA_TYPE *in, DATA_TYPE *out);
```

mdct_backward() does not return a value. But instead the input and output array of data are passed by reference as *in* and *out* variables respectively. No change will be made to *in* (input vector) but the result transformed data will be saved to *out* (output vector). *DATA_TYPE* is *int* for integer version of Vorbis library. Structure *init* is pre-initialized static data of type *mdct_lookup* structure whose size is dependent on the size of input and output vectors. Structure of *mdct_lookup* is shown below:

```
typedef struct {
  int n;
  int log2n;
  DATA_TYPE *trig;
  int *bitrev;
  DATA_TYPE scale;
} mdct_lookup;
```

This *mdct_lookup* structure is initialized in the *mdct_init()* function. Variable *n* is the size of input and output vectors to be fed to MDCT function which corresponds to Vorbis internal block sizes (either 256 or 2,048). log2n and scale variable will be calculated by *mdct_init()*. The important note here is the two big arrays *trig* and *bitrev*, which after initialization consume space around *6*n* bytes (4,608 and 12,288 bytes for small and large block sizes respectively). This amount

is too big to be stored in the hardware therefore they will be initialized in software and at every call to hardware, the hardware core will do the DMA-transfer of this data for the calculation. Therefore the register interface of MDCT (and also apply to Mini-MDCT core) was defined similarly to that of audio core in chapter 3.3 as follows:

```
struct mdct_core_regs {
  volatile unsigned int controlreg;        /* 0x00 */
  volatile unsigned int arraysize;         /* 0x04 */
  volatile unsigned int bitrevaddr;        /* 0x08 */
  volatile unsigned int trigaddr;          /* 0x0C */
  volatile unsigned int startreadaddr;     /* 0x10 */
  volatile unsigned int startwriteaddr;    /* 0x14 */
  volatile unsigned int status;            /* 0x18 */
  volatile unsigned int actmemaddr;        /* 0x1C */
};
```

MDCT-core interface has total 8 registers. *arraysize* contains the appropriate size of input and output vectors (value 0 for 256 and 1 for 2048). *bitrevaddr* and *trigaddr* are addresses for *bitrev* and *trig* arrays (part of *mdct_lookup* structure as described above) to be DMA-transfered consecutively. *startreadaddr* is start read address for input vector and *startwriteaddr* is start write address for output vector. *controlreg* is the main control register. The core will start the calculation upon writing 1 to bit 0 of *controlreg*. Status register will be 1 during the calculation and 0 otherwise. *actmemaddr* reflects the current processing memory address. Complete information about MDCT-core can be found in the Chapter 4.

After defining the interface of MDCT (and Mini-MDCT) core, it has been implemented as TSIM I/O module. Vorbis library (the integerized version) was then modified to make use of the core. Usual use of this MDCT-core takes place in *mdct_backward()* function and is illustrated in Figure 3.9. *mdct_backward()* function of Vorbis library was modified to utilize MDCT-core instead of calculation by its own. When *mdct_backward()* is called, it will write the start read and write register, addresses of bitrev and trig registers, and set the appropriate values to control register. After that the hardware core will start the inverse MDCT calculation. *mdct_backward()* will monitor the status register value, wait until it becomes 0 (which means that hardware-core has finished its calculation), and then return the output. For the case of Mini-MDCT, the extra calculation which was missing (because Mini-MDCT was a subset of MDCT) is added after getting result from hardware core.

Both partition candidates (MDCT and Mini-MDCT) were tested with simple-player on TSIM at 25 MHz clock frequency. The result is shown in table 3.3. The solution with MDCT core as I/O module gave result faster than real-time with margin time = 15.40 - 11.70 = 3.7 seconds. The Mini-MDCT solution gave result time almost right at the real-time limit. Note that by testing with TSIM, the

mdct_backward(init, in, out)

```
volatile struct mdct_core_regs *regs;
regs = (struct mdct_core_regs *) MDCT_CORE_START;

reg->startreadaddr = in;                    }
reg->startwriteaddr = out;                  }  Write to MDCT-core registers
regs->bitrevaddr = init->bitrev;            }
regs->trigaddr = init->trig;                }

controlreg = 0x1;           <----  MDCT-core starts the calculation here

while (regs->status != 0 );  <----  Wait until MDCT-core finishes the calculation

...
```

Figure 3.9: Modified *mdct_backward*() function

processing time of hardware core (which is implemented now as TSIM module) is not taken into account, one must add the appropriate time needed by hardware core to the total simulated time obtained from TSIM in order to estimate the real elapsed time. Also this test was done with the simpleplayer, more overhead from RTEMS operating system in case of the full player must be also added.

Table 3.3: Result of simpleplayer tests on TSIM at 25 MHz

Program	**Result (seconds)**
Final Player	19.42
Final Player + Mini-MDCT core as I/O module	15.41
Final Player + MDCT core as I/O module	11.70

From the statistics above, it could be concluded that the suitable partition was the MDCT case. In addition, it must be calculated further that the designed hardware core could complete its work (1,888 calls of *mdct_backward()*) within 3.7 seconds margin time, which is highly probable. For Mini-MDCT, however, it was expected to be too marginal as the partition is right at the limit. However, in the case that designing the whole MDCT core was not possible, Mini-MDCT could be the choice and by reducing the quality or sampling frequency of encoded music data, the final player could decode the Ogg Vorbis stream in real-time for demonstration purpose.

3.5.3 Summary

In this phase the main task of the project, hardware/software partitioning, was done. TSIM and its I/O module were used as the main tool. Two partitions, MDCT and Mini-MDCT were selected and tested. The statistics of simpleplayer running on TSIM with the help of these MDCT and Mini-MDCT cores are shown in table 3.3. MDCT was proposed as the preferred partition in order to have the music decoded and played in real-time. Mini-MDCT was proposed as the second solution in case that the implementation of the whole MDCT core was not feasible.

3.6 Player Development

On top of the LEON processor, there are choices of either running the application code directly or using an operating system. Simpleplayer from the previous Section is an example of running the application code without the operating systems. Running directly the application without OS (and accessing directly to hardware address) is free of OS overhead, but by using an OS, one can get benefits from services offered by OS e.g. task management, memory management, and abstraction of hardware devices. Also the application code will be easily portable to other operating system in the future because it can use the abstraction interface of hardware provided by OS instead of having direct hardware accesses. For these reasons, it was decided to have an OS for the final version of player.

Viable choices of operating systems at the time of this project were the RTEMS from OAR Corporation [6] and uCLinux [33] (LEON-port by LEOX team [31])[3]. We preferred at the beginning to try uCLinux as Linux was open-source and became more and more popular in the PC market. However, we found that uCLinux port on LEON were not yet mature and development of user applications was at that time not trivial. C library was not yet available.

RTEMS, on the other hand, offered good and stable support for LEON. It worked well with Newlib C library [16] and also supported POSIX standard. RTEMS is open source as well (copyrighted under a modified version of GPL). Therefore RTEMS was chosen to be used in the project.

In this chapter, the development of final layer as extension to simpleplayer is described. RTEMS was used and the player now could output the audio music to audio device via the developed audio device driver.

[3] Later in March 2002, eCos operating system from Red Hat Inc. was successfully ported to LEON.

3.6.1 RTEMS

RTEMS [6] is an acronym for the *Real-Time Executive for Multiprocessor Systems* developed by *On-Line Applications Research Corporation (OAR)* for the U.S.Army. The goal of RTEMS was to provide a portable, standards-based real-time executive for which source code was available. RTEMS is licensed under a modified version of the GNU General Public License (GPL).

The word *executive* in this sense means the (usually small) operating systems kernel. Although former developed primarily for military applications, it was now used also in general embedded applications. The original *classic* RTEMS application programming interface (API) is based on the *Real-Time Executive Interface Definition (RTEID)* and the *Open Real-Time Kernel Interface Definition (ORKID)*. RTEMS claims to support about 70% of POSIX 1003.1b-1996 API standard[4]. This POSIX standard defines the programming interfaces of standard UNIX. This means that much source code that works on UNIX should also work on RTEMS. It includes support for POSIX threads and real-time extensions.

RTEMS is available on various processor families e.g. Motorola MC68xxx, Motorola MC683xx, Motorola ColdFire, Hitachi SH, Intel i386, Intel i960, MIPS, PowerPC, and SPARC. RTEMS works well with LEON processor and was included in the LECCS software development kit from Gaisler research.

RTEMS Compilation

RTEMS was included as pre-compiled library in the LECCS package from Gaisler research. Re-compilation is generally not necessary as all the tool works right out of the box for software development on LEON SPARC platform. However, there was a case in our project that recompilation was necessary. As the POSIX Thread API was needed for the final player and POSIX Thread support was not enabled in the precompiled RTEMS version that came with LECCS, recompilation has been done as shown below:

```
$ cd <RTEMS extracted source path>
$ mkdir build && cd build
$ ../configure --target=sparc-rtems --enable-posix \
    --prefix=/usr/local/my-rtems --enable-tests
$ make RTEMS_BSP="leon2"
$ make install RTEMS_BSP="leon2"
```

According to the above commands, the newly compiled RTEMS will be installed under /usr/local/my-rtems directory. With *–enable-tests* option, many example programs will be also compiled and installed.

[4]This was mentioned in `http://www.oarcorp.com/rtemsdoc-4.5.0/rtemsdoc/html/FAQ/FAQ00004.html`

sparc-rtems-gcc compiler came with LECCS has default location of RTEMS library under /opt/rtems/sparc-rtems/lib/. In order to have this new libraries worked with sparc-rtems-gcc from LECCS, the newly compiled RTEMS library must be copied to that path as follows:

```
# cd /opt/rtems/sparc-rtems/lib/
# cp -pdvR /usr/local/my-rtems/leon2/lib .
# ln -s libbsp.a libleon.a
```

Developing RTEMS Applications

A typical RTEMS program has choices to use POSIX standard, RTEMS classic API or ITRON 3.0 application programming interfaces (APIs). In our project only POSIX and classic RTEMS APIs were used. Simple test programs used classic RTEMS API. The final player used POSIX API as the multi-threading feature was needed and we preferred the POSIX Thread API to classic RTEMS task API because POSIX Thread is available on many platforms, the software can be developed and tested on Linux workstation and only at the end run on RTEMS and target hardware.

Developing RTEMS application is similar to writing normal C program except that appropriate definition and includes statements are needed to enable required features of RTEMS. To get the idea how programming with RTEMS works, simple hello-world program with RTEMS classic API is shown below:

```
#include <rtems.h>
/* configuration information */
#define CONFIGURE_INIT
#include <bsp.h>

rtems_task Init( rtems_task_argument argument);

/* configuration information */
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_MAXIMUM_TASKS             1
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_EXTRA_TASK_STACKS  (3 * RTEMS_MINIMUM_STACK_SIZE)

#include <confdefs.h>

rtems_task Init(
  rtems_task_argument ignored
)
{
  printf( "Hello World\n" );
  exit( 0 );
}
```

The program can be compiled and tested on various hardware target. Here we compile and test the code on unix/posix target which we can run the code natively

on Linux-x86 PC:

```
$ gcc -B/usr/local/my-rtems/posix/lib/ -specs bsp_specs \
      -qrtems -o hello-rtems hello-rtems.c
$ ./hello-rtems
Hello World
```

And then the program is cross-compiled for sparc-rtems target and tested under TSIM:

```
$ sparc-rtems-gcc -o hello-rtems hello-rtems.c -rtems
$ tsim-leon  -freq 25 hello-rtems

 TSIM/LEON - remote SPARC simulator, version 1.0.18a (professional version)

 Copyright (C) 2001, Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to tsim@gaisler.com

using 64-bit time
serial port A on stdin/stdout
allocated 4096 K RAM memory, in 1 bank(s)
allocated 2048 K ROM memory
icache:  4 kbytes, 16 bytes/line
dcache:  4 kbytes, 16 bytes/line
Enter IO Module:  time = 0
section:  .text at 0x40000000, size 95184 bytes
section:  .data at 0x400173d0, size 2208 bytes
tsim> go
resuming at 0x40000000
Hello World

Program exited normally.
```

Note: In order to run directly the binary executable from sparc-rtems-gcc on TSIM, one needs to run TSIM with 4 MB RAM setting because as default, binary output from compiler will work with 4 MB RAM target only. Otherwise one can use *mkprom* tool to generate the executable image for other RAM, ROM, frequency, and cache size settings.

By default, RTEMS application will not be configured to use the floating point unit. If the FPU hardware exists and the use of it from an RTEMS application is desired, one must add this line in the program header:

```
#define CONFIGURE_INIT_TASK_ATTRIBUTES \
        (RTEMS_FLOATING_POINT | RTEMS_DEFAULT_ATTRIBUTES)
```

RTEMS is aimed for embedded application target. Most features are not enabled by default. Enabling and disabling of features are done via these define statements on the program header. More information about this could be found in the RTEMS RTEMS C User's Guide [5].

3.6.2 Device Driver for Audio-Core

Recalled from previous sub-Section, we had the simpleplayer program which could decode the Vorbis stream but could not yet send the stream out the sound hardware. The missing jigsaw here was something that provides an access point for application in order to access audio device, accepts audio data from application via that access point, and sends that data to audio hardware. This is the task of audio device driver.

Device driver in the operating system provides an abstraction layer of underlying hardware for software applications. Instead of accessing each register of the audio core directly in order to send music data to be played, device driver provides a file representing the hardware as access point. This device file interface can be accessed in the similar way as a normal file on UNIX/POSIX-compatible operating systems. In our project, the standard programming interface for audio from OSS (Open Sound Systems) was used as it was prevalently used in the Linux and Open Source/Free software. OSS provides a complete API with extensive documentation [32]. OSS uses */dev/dsp* device file for audio hardware. Various sound parameters e.g. sampling frequency, number of channels and sample size could be set via the UNIX standard *ioctl()* system call. Typical usage of audio playing function via */dev/dsp* used in our project is illustrated in Figure 3.10. Application first opens the audio device via *open()* call and then writes music data via *write()* call. Notice that the call to device driver blocks until result is returned.

An RTEMS device driver for audio core has been written. It was implemented according to the OSS standard. It contains 6 main functions *sound_initialize*, *sound_open*, *sound_close*, *sound_read*, *sound_write*, and *sound_control* to handle the initialization, open-, close-, read-, write- and ioctl-operations of the */dev/dsp* device file. In the header file of the device driver they are declared as follows:

```
#define SOUND_DRIVER_TABLE_ENTRY \
{ sound_initialize, sound_open, sound_close, \
  sound_read, sound_write, sound_control }
```

Function *sound_initialize()* will register itself to RTEMS via *rtems_io_register_name()* function so that other operations to */dev/dsp* will be forwarded to this device driver code.

The device driver can be used by adding it to RTEMS Driver Address Table via appropriate define statements in the header of RTEMS application as follows:

```
#define CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE
#define CONFIGURE_MAXIMUM_DRIVERS 3
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS 5

rtems_driver_address_table Device_drivers[] = {
        CONSOLE_DRIVER_TABLE_ENTRY,
        CLOCK_DRIVER_TABLE_ENTRY,
```

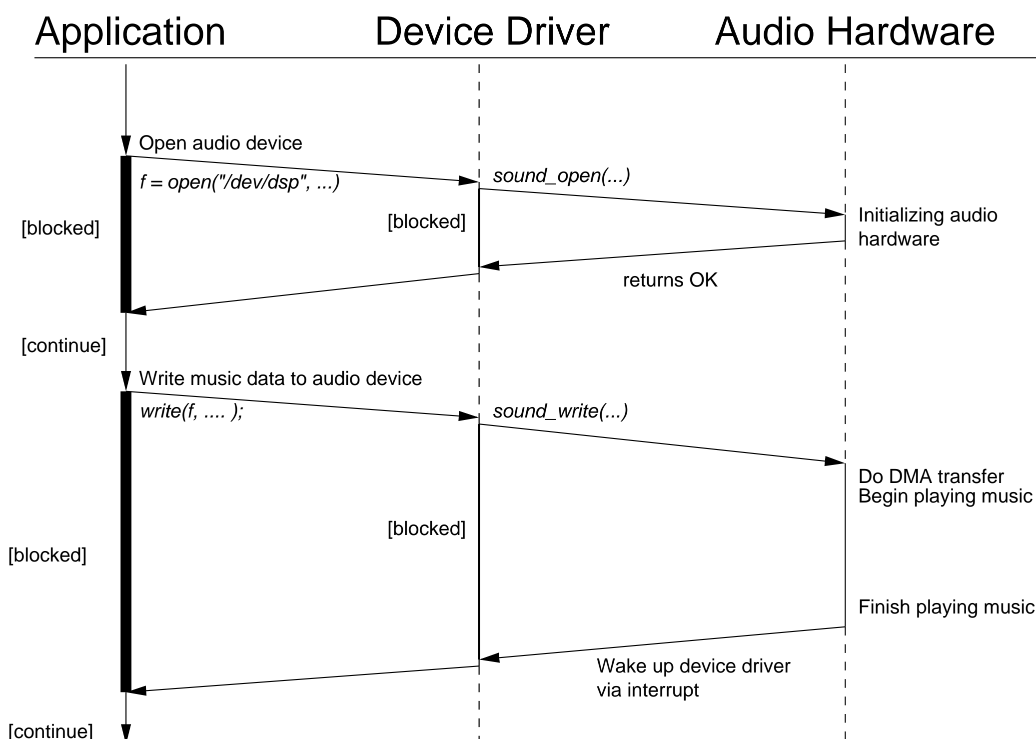

Figure 3.10: Example of open() and write() calls to audio device

```
        SOUND_DRIVER_TABLE_ENTRY,
        { NULL, NULL, NULL, NULL, NULL, NULL }
};
```

With appropriate header to add the audio device driver shown above, application code can later access the audio device via */dev/dsp*.

3.6.3 Final Player with Sound Output

At this point we had the simpleplayer which could decode the Vorbis stream and sound device driver which provided */dev/dsp* as abstraction layer for audio-core, the final player that could really output the music to audio device was written.

As we have seen in Figure 3.10, writing music data to */dev/dsp* will block until all the music data has been played. While waiting for this blocking music playing process, it is more efficient to continue decoding the next block of Vorbis stream data instead of doing nothing. The final player accomplished this with the help of multi-threading feature of POSIX Thread (or PThread). By having one thread for decoding Vorbis stream and another thread for receiving decoded data from the first thread and then playing it through audio device, the decoding thread could

always work without having to wait for the music playing process. The decoding task needs to communicate with the audio-playing task with the appropriate mechanism so that the audio-playing task will play the current already-decoded block while the decoding continues decoding next block. This communications between threads are illustrated in Figure 3.11.

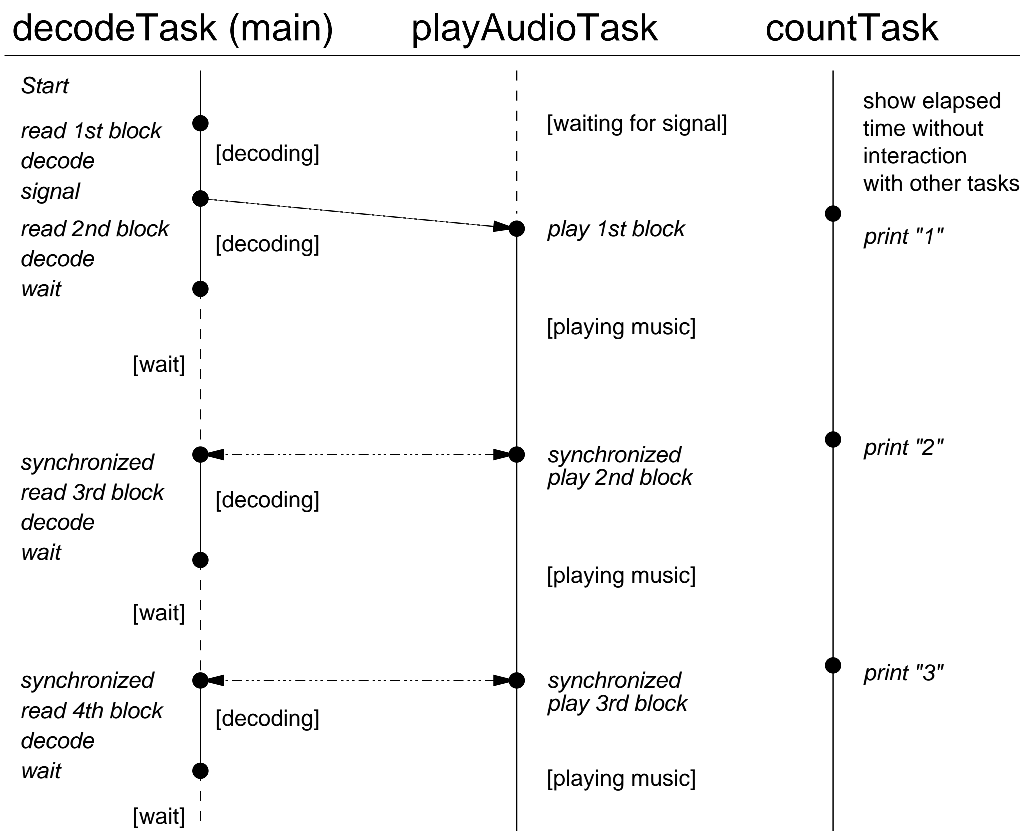

Figure 3.11: Communications between threads

As shown in 3.11, there are 3 tasks: *decodeTask*, *playAudioTask* and *countTask*. The countTask simply shows the elapsed second on standard output without any interactions with other tasks. Synchronization is needed between decodeTask and playAudioTask. The playAudioTask will be signaled when new data is decoded by decodeTask and playAudioTask signals the decodeTask when it finishes playing each block. This synchronization means both threads wait to meet each other after processing each block. Two audio buffers were used in the player so each thread always works on different audio buffers alternatively.

With the MDCT hardware core developed in hardware part (more information in Chapter 4), the decodeTask made use of this hardware core instead of doing

the calculation on its own. The final player was successfully tested on the real hardware board with the MDCT hardware core (see Section 4.4).

3.6.4 Summary

In this phase, the simpleplayer was extended. The RTEMS operating system was introduced because it offered the benefits of multi-threading (POSIX Thread), task synchronization and hardware abstraction. Device driver for audio device has been written to provide abstraction layer of the audio hardware core. The final player was finally written with complete Vorbis decoding and sound output functions. In the final test phase, it ran on the hardware board with the help of the MDCT core developed in hardware part successfully.

3.7 Conclusion

Experiences and Problems found

- TSIM was used extensively in the project. It increased largely software development speed and usually gave accurate performance result compared to LEON on the real hardware. However, not all settings in the real hardware were available under TSIM e.g. multiplier type or load delay number in the Integer unit. Care should be taken at these points.

- Most software used in the project (RTEMS, TSIM) were new and evolving. Sometimes we experienced lack of documentation. However great support from user-communities (e.g. public mailing list *rtems-users@oarcorp.com* and *vorbis-dev@xiph.org*) and direct support from software vendors (Gaisler Research) have largely helped.

- Lack of Vorbis official specification made it difficult to produce correct and well-optimized integerized Vorbis decoder library.

Summary

In the software part, the tools for hardware/software partitioning evaluation were delivered. Vorbis player for the target system was developed. It ran with RTEMS operating system and accessed to audio device via audio device driver. Around 40% of speed improvement was achieved from software optimization.

Hardware/software partitioning was done and two hardware/software partitions were proposed. The whole MDCT function as partition was the preferred

solution in order to decode the sample Ogg Vorbis data in real-time. The Mini-MDCT core as partition was suggested as the secondary solution in case that the whole MDCT core could not be implemented. Real-time decoding demonstration of sample data with Mini-MDCT could be obtained by adjusting the encoding parameters of the sample data.

Possible extensions to the project in the software part are adding features to store music file on storage devices e.g. USB flash disk or remote network server which is accessible via TCP/IP. The player could also be configured via WWW interface. Realization of Ogg Vorbis support for portable audio devices or gadgets based on the result of this project is also challenging.

Knowledge gained from software development in this project can be also applied further to other System-on-a-Chip systems development with hardware/software co-design techniques.

Chapter 4

Underlying Hardware

Observing Figure 1.1 the two bottom layers constitutes the hardware part. They must provide physical structures to support the player. The bottom layer holds the target technology, where software and hardware have to fit. For this project it is represented by the XSV-800 board, but it could be any low computation-power system. The second level is where LEON platform is located, with the audio core and the user defined core.

The exploration of these two layers consists in recognizing the borders of the bottom layer, such as maximum frequency to synthesize LEON, and configuring the platform to provide the best performance.

Once the limits of the technology were recognized, and the best configuration was found, take place the design and implementation of extra hardware to accelerate the execution of the player. After providing the right configuration, the partitioning process gives the new task to develop. This process can be observed in Figure 1.2.

In this chapter will be discussed how the platform was configured and the design, as well as implementation process of the modified discrete cosine transform (MDCT) core, which was the result of the partitioning presented in Section 3.5.

As Ogg-on-a-chip project started, the necessity to implement some functions of the software as hardware, in order to speed up the system, was clear. But the question was which part should be implemented as hardware. In Section 3.5 was explained how this decision was made, using the TSIM simulation environment, and modeling different parts of software as hardware. Some alternatives were considered to speed up the application, for example using multiple processors, and splitting the functions of the decoder among them. Nevertheless this approach introduces new problems. One of them is how to ensure cache coherency in the case that more than one processor work together decoding the same frame. In Ogg-Vorbis the data frames are independent of each other, therefore would be possible to assign the complete decoding process in one frame to a processor,

and decode 4 frames at the same time using four different processors. Here the problem is the memory required to store independent frames would increase proportional to the number of processors, and even for 2 processors the memory on typical low computation-power systems would not be enough.

On the other hand, the approach developing a MDCT core has some advantages; due to MDCT is present in many audio and video codecs (see Section 2.4) the application field is wide. In addition, because it is based on AMBA standard [20], the MDCT core is portable and reusable. Consequently it can be used with different processors.

According Figure 1.2 there are 3 Work Packages for hardware part and one final test. They are described in this Chapter as follows:

- Section 4.1: **Platform Exploration.-**Describes the hardware limitations and the process to import the audio core, and configure the platform with all necessary features.

- Section 4.2: **MDCT Core Design.-**Describes the specification, architecture and implementation of the requested MDCT core.

- Section 4.3: **Simulation, Synthesis and Test.-**Describes the process to generate the hardware (platform and new core).

- Section 4.4: **Final System Test.-**Describes the final version of the player.

4.1 Platform Exploration

Sections 4.1.1 to 4.1.4 covers the configuration of the platform, looking for providing the better support for the decoder, and describe the improvements done on the audio core imported from the Digital Dictation Machine (DDM) [2]. During the configuration phase, the new core (from now this core will be referenced as MDCT core) has no arithmetic function. It is only an APB slave and a very simple AHB master.

The file with the described configuration is located on *cvs* (see Appendix A), and it is named *target-ooac.vhd*. For synthesis or simulation purposes, the original *target.vhd* from LEON has to be replaced, and in *device.vhd* file the next line has to be set:

```
constant conf :  config_type :=virtex_4k4k_v8_m32;
```

4.1.1 Upgrading to latest LEON Version

The DDM [2] was written for Leon1-2.2.2 version. At the time Ogg-on-a-chip project started, the latest version was Leon1-2.3.7. In short time three different

version were released. After Leon1-2.4 the first version of Leon2 was introduced, this was Leon2-1.0.1. A couple of weeks later Leon2-1.0.2 came out, and only a week later Leon2-1.0.2a, which was the final version used by Ogg-on-a-chip. Because of those fast changes on Leon, importing DDM and Ogg-on-a-chip to new Leon versions is specially important.

The main difference between Leon1 and Leon2 is the addition of one Debug Support Unit (DSU) as an AMBA master. Therefore the Leon2 top entity has more connections than Leon1, and the configuration files of Leon1 can't be used anymore for Leon2.

Both cores, audio and MDCT are AMBA masters. The involved files to add these cores to Leon are the following: *target.vhd, iface.vhd, ambacomp.vhd, mcore.vhd* and *leon.vhd*.

Since *target.vhd* file contains the possible Leon configurations, it has to be changed to add new elements and configure them. In both cases, the memory mapped registers are accessible using APB. For this reason the APB slave has to be configured by adding DDM and MDCT values for **apbslvcfg_std** vector. The index value to assign can be different and depends on how many devices are configured. For Leon2 are new devices such as DSU master for example. MDCT is configured to use 0x800000300 to 0x800000318 space, meanwhile DDM uses 0x800000200 to 0x800000218 as memory mapped registers.

After configure APB slave, the number of masters in AHB configuration has to be increased by 2, for **ahb_std** vector located in the same *target.vhd* file.

In *iface.vhd* the data types for core signals are located. Only DDM needs some data types to communicate with external world, and they are declared here.

Ambacomp.vhd contains different elements used by AMBA. It is a package and both cores have to be declared there.

Actually the file that connects both cores with the rest of the platform is *mcore.vhd*, so they are here instantiated. The priority is given to each AHB master with and index value. The higher this value is, the higher the priority it owns. APB bus indexes have to be added according the declaration in *mcore.vhd*. In both cases irq signals are declared to connect them with the irq controller. For DDM the addition of in/out signals to mcore entity is needed .

Last file to modify is l*eon.vhd*, by adding pads for DDM. The case of MDCT is different, since it doesn't need any external pad, no change is required.

The modified files to use in Ogg-on-a-chip project are located in the *cvs* directory and named as *<original name>-ooac.vhd* (for example *mcore-ooac.vhd*). If there are no major changes in Leon for upcoming versions, it would be enough to replace the normal mentioned files by those ones, in order to import easily the project.

4.1.2 Hardware Constraints

Referencing again to the last layer on the system overview (Figure 1.1), the borders of this layers are given by the constraints of the target technology which is supposed to hold the demonstrator. In our particular case, the demonstrator was built on Xess XSV-800 board. According [7] the main issues to keep in mind are:

- Size.-The FPGA XCV-800 can hold designs with up to 800,000 gates
- Timing.-LEON can be synthesized for Xilinx XCV-800 speed grade 4 in range 25-28 MHz.
- System clock.-The on-board clock provides frequencies sub multiples of 100 MHz, i.e. 50,33,25,20,10,5 MHz. For this reason the maximum clock frequency for the system without an external oscillator is 25 MHz.
- Audio sample frequency.-The audio sample frequency is correlated to the system clock frequency, and it is a sub multiple of the system clock. As described in [2] the sample frequency of the audio core is given by the formula $fs = \frac{clk}{256(2+2scalerup)}$, where *scalerup* can be 0,1, or 2. For this reason the standard audio sample frequencies such as 44 kHz and 22 kHz are slightly different. The best approximations are 48kHz and 24kHz.
- Internal RAM on FPGA.-Since the memory on the XCV-800 has to be shared with different elements of the platform, such as cache and MDCT core, this is a very important issue having a big impact on the overall performance of the system.

4.1.3 Audio Core

In order to import and use for music the audio core designed by Daniel Bretz in [2], some improvements were required, for example music in stereo is better than one channel music. This Section describes the improvements for the audio core.

4.1.3.1 Audio Core Configuration

This core consists in an AHB master which has connection with the on-board audio chip AKM AK4520A [7]. Initially the core had a word length of 20 bits, and only one channel was used. Nevertheless only 16 bits were played. Due to this fact, and thinking about saving memory, the decision to deploy a word length of 16 bits was made, and to add the feature of stereo mode.

Besides this two modifications, the interrupt feature was enabled and the way to reach the end of the to-play data block was slightly changed. This two actions were done in order to allow easily interaction with software, and specially the second one was implemented trying to avoid a sound gap while re-filling the buffer.

In Chapter 3 the functional interface of this audio core was discussed, therefore will be assumed the reader is familiar with.

4.1.3.2 Stereo Function for Audio Core

In the DDM version [2], the audio core uses a register of length 20 bits, and only the left channel. Each time when a direct memory access (DMA) was performed, it reads a word of 32 bits from memory, but stores only 20 bits in the shift register, which content is sent bitwise later to the on-board audio chip.

In the version used by Ogg-on-a-chip, the DMA is carried out in the same way, that means reading one word 32 bits long, but storing the whole 32 bits in one register. For the software is known, that the higher 16 bits of each time domain sample store the left channel sample, and the lower 16 bits the right channel. For this reason the software of DDM can't be used with the audio core used by Ogg-on-a-chip.

In this way two samples are read at each DMA cycle, and because the word length is already 16 bits, two samples are stored in memory, where before only one was stored.

The register which stores the samples is 32 bits long and is called *audiobuffer*, but another register is used to shift the msb bit to audio chip. This is called *audioshifter* and is 16 bits long. The higher 16 bits or the lower 16 bits of *audiobuffer*, are copied to *audioshifter* depending on which value has the signal *lrsel*. When it is high, the higher 16 bits are stored, and when it is low, the lower 16 bits are stored. This because *lrsel* is the select channel signal for the audio chip. Instead to shift the bit 19 to the audio chip, is shifted the bit 15.

The word length changes apply for record function as well, but they were not tested, because record function is not required by Ogg-on-a-chip.

The final architecture of the improved audio core is shown in Figure 4.1.

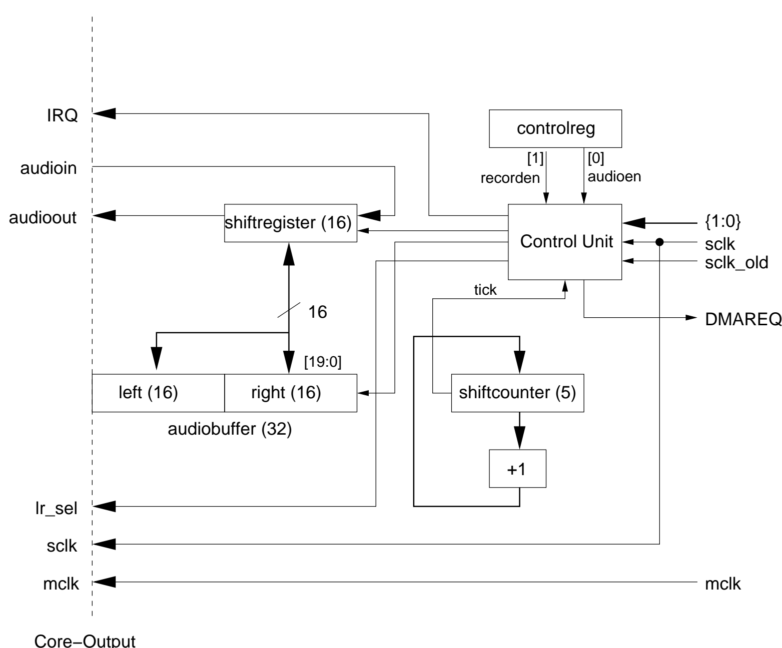

Figure 4.1: Audio Core diagram.

4.1.3.3 Interrupt and Internal Stop Address

In the original version of audio core, the interrupt was already there, but was not used by the software. The only thing missing was to send the interrupt signal to the output port. A normal audio card owns an internal buffer to play the samples, but implementing an audio buffer was not suitable due to the lack of internal memory on FPGA. In order to fix this problem the software has to modify the start and stop addresses while the audio core is in loop mode, in this way the core is changing block addresses, and the software has enough time to change the new start/stop addresses. But what happens if the new stop address is above the old start address? The answer is the audio core will continue reading until reaching the end of the memory. In order to avoid this situation, a new register was added, which holds a copy of stop address in the moment when the core start to process a new block. This new register is the internal stop address.

In the original audio core, the last sample of the block was not sent to the audio chip, but for this project it has to be played. Once the audio core reaches the last address of the block, it raises the interrupt 13 (if enabled) and copies the *audiobuffer* again to the *audioshifter* register. In this way the processor has enough time to refill the *audiobuffer*, before the core shift the whole data again. The file with the new version of the audio core is located in *hardware/vhdl_designs/DDM16.vhd*.

For more details how to implement this feature, please refer to the comments in the mentioned file.

4.1.4 Ogg-on-a-chip Hardware Configuration

LEON is a configurable platform allowing a trade-off between performance, size and power consumption. Once the constraints described in 4.1.2 were identified, and before do any kind of partition, the most suitable configuration has to be found. This step was made while the software part was providing tools and method to carry out the partition, so the results of the configuration on hardware were an input for the partitioning. In other words, the configuration on real hardware has to push the system to the edge in order to get the best performance, and see what is still missing to achieve the goal of decoding in real-time. It is important to remark that not all settings of the LEON platform are configurable in TSIM (see 3.5) and the performance results of simple player were different on TSIM and on real hardware. Some Items in the integer unit have ideal values (0 delay time) in TSIM. On the other hand, some items are well modeled by TSIM, such as cache configuration, allowing a fast modeling in the simulation, but the rest might be done on real hardware. The configuration phase ended when the best time for simple player was reached on the board. This configuration is shown in Figure 4.2 and described in the following sections.

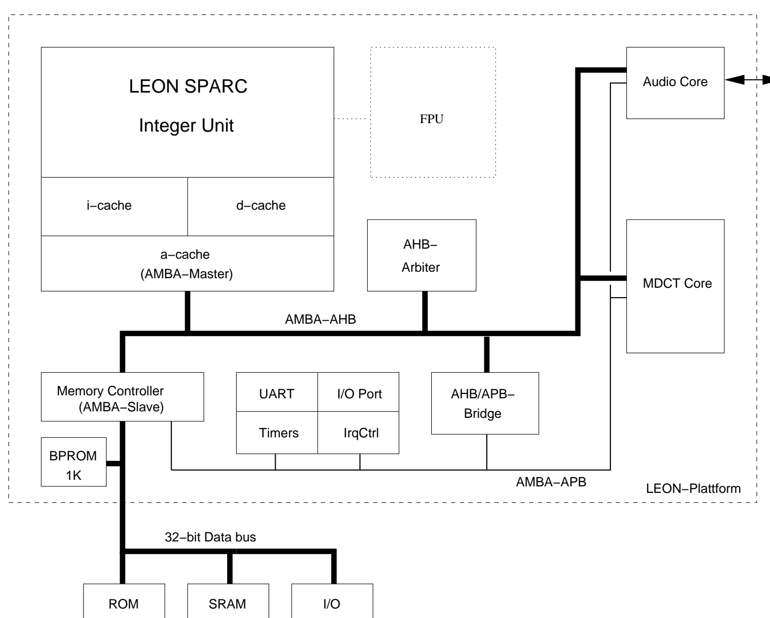

Figure 4.2: Platform Configuration.

4.1.4.1 Extraction and Integration of Meiko FPU to LEON

As described before in chapter 3, Ogg-Vorbis uses floating-point calculations, therefore at first sight a FPU was required, and during hardware configuration, on software side was not clear if an integer version of Ogg-Vorbis decoder was feasible. For this reason the FPU was integrated to the system.

Fortunately LEON can be connected to the Meiko floating-point core, which is part of MicroSparcII [21][1] model.

Meiko FPU provides full floating-point support according to the SPARC-V8 standard [28]. Two interface options are available: either a parallel interface identical to the co-processor interface, or an integrated interface where FP instruction do not execute in parallel with IU instruction. The FPU interface is enabled/selected by setting of the FPU element of the configuration record.The parallel interface lets FPU instructions execute in parallel with IU instructions and only halts the processor in case of data- or resource dependencies. Refer to the SPARC V8 manual [28] for a more in-depth discussion of the FPU and co-processor features. The Meiko floating-point unit (FPU) consists of the Meiko floating-point core and a fast multiplier.

According SCSL license, to use Meiko FPU it has to be extracted by the user. Since LEON platform is written in VHDL and MicroSparcII is in Verilog, a void Meiko entity is declared in *meiko.vhd* file and later instantiated in *FPU_core.vhd*; in old versions of LEON this step was done at the bottom of *fp1eu.vhd*. If a configuration in *device.vhd* is selected, and it uses Meiko FPU, the component will be included even when Meiko FPU is no there. Meanwhile the extracted Model can be synthesized within the same project as LEON (in SYNPLIFY PRO and SYNOPSYS is possible), but only after synthesis, when the modules will be expanded to create the edf file, Meiko is connected to Leon. This could be confusing, because if Meiko is not properly extracted, but right after synthesis seems to be correct integrated.

A simplified MicroSparcII FPU Block Diagram is showed in Figure 4.3. Since Meiko FPU is just a module, and it is an independent FPU, only this one is required to be extracted. In the code the division is not clear, and looks like the parallel fp multiplier is a part of Meiko, but that's no true. In [21] is mentioned that the fp multiplier was added later (not in the original Meiko core) to improve fp multiplication performance. The multiplier is rather big, it consumes about the same number of gates as the rest of the core. The size of Meiko is about 20 000 gates.

The files used by Meiko are located in following directories of MicroSparcII:

---

[1] MicroSparcII is available at http://www.sun.com/processors/communitysource/ under the Sun's Community Source Licensing (SCSL). This model is free for prototypes. Please refer to the same link for more details about SCSL.

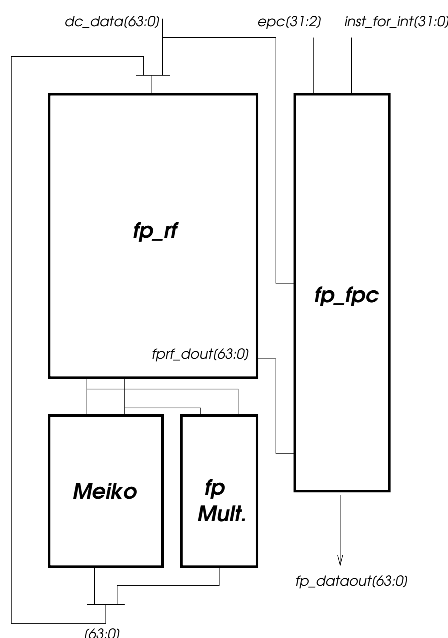

Figure 4.3: MicroSparcII FPU Block Diagram.

cells, ff_primitives, macros, and me_cells, sc. In directory FPU only FPU_ctl, fpu_exp, fpu_fpc, fpu_frac and fpu_rom module are necessary. Note that the following modules should not be used: fpufpc, fprf, fpm, fpufpc_spares.

The top entity is named FPU and located in *fpu_fpc.v* file. It should be uncommented, and the fpufpc entity (top entity of MicroSparcII) should be disabled.

One file has to be created containing the files in the directories mentioned above, and this file can be added as described before to a synthesis project.

When Meiko FPU is already synthesized with LEON, it can be tested used the program *paranoia*.c, located in the examples of LECCS [10].

In the final version of Ogg-on-a-chip, Meiko FPU was not included, because an integer version of the decoder described in Section 3.4 was implemented, which was faster than the version using FPU.

4.1.4.2 DSU Integration

Sometimes is useful having the ability-on-board to debug hardware, because simulations at different levels don't provide completely realistic scenarios, and only once a prototype is running on real hardware some bugs can be detected. During the FPU integration process, some errors on real hardware were difficult to reproduce using simulators (TSIM for software or Modelsim for Hardware) for this

reason was incorporated the Debug Support Unit (which is present in LEON-2) to add software debugging on target hardware The support is provided with the help of a Debug Communication Link (DCL). This Section describes how to synthesize it for the XSV-800 board.

The debug support unit is used to control the trace buffer and the processor debug mode. The DSU is attached to the AHB bus as slave, occupying a 2 MB address space. Through this address space, any AHB master can access the processor registers and the contents of the trace buffer. The DSU control registers can be accessed at any time, while the processor registers and caches can be only accessed when the processor has entered in debug mode. For more details about DSU and DCL please refer to [11].

In order to be used, the entity xsv800 created by Daniel Bretz has to be modified, because top entity leon on LEON-2 has more pads, whose are the following external DSU signals:

- DSUACT-DSU active (output)
- DSUBRE-DSU break enable
- DSUEN-DSU enable (input)
- DSURX-DSU receiver (input)
- DSUTX-DSU transmitter (output)

On XSV-800 there is only one serial port available, therefore when the DSU is used, the serial port is connected to signals DSU-RX and DSUTX, and can not be used anymore as standard output. It could be possible to use the PS2 port as serial by using a transceiver. More information about this topic available on LEON mailing list.

The file *hardware/vhdl_designs/XSV800_32b-DSU.vhd* is located in *cvs* and contains the modified xsv800 top entity. Using this file the processor will enter by power up to debug mode, because the signal DSUEN-DSU is always high, enabling DSU, and DSUBRE is connected to clk. In this way at least one low-high transition occurs, generating break condition and putting the processor in debug mode.

Gaisler research provides LEON DSU Monitor, which is a debug monitor for the LEON processor debug support unit, and is the interface to receive and send data to XSV-800 through serial port. DSUMON can operate in two modes: stand alone and attached to gdb. In stand alone mode, LEON applications can be loaded and debugged using a command line interface very similar to TSIM environment. A number of commands are available to examine data, insert breakpoints and

advance execution [9]. When attached to gdb, DSUMON acts as a remote gdb target, and applications are loaded and debugged through gdb.

The inclusion of the DSU doesn't play any role on the decoder. It was included as support to develop the requested suitable configuration.

4.1.4.3 Integer Unit Configuration

The integer unit on LEON is actually the processor, therefore important configuration settings take place here, and those settings are until now not configurable on TSIM.

In standard configuration, the multiplier used is 16x16 bits and takes 4 clock cycles to execute one multiplication. It consumes 6,000 gates. Nevertheless this configuration is far away to provide a good performance of the player, since a big amount of multiplications have to be done. Consequently using the fastest multiplier have a big impact on the overall system performance.

The configuration used is 32x32 bits and performs one multiplication in one clock cycle, meanwhile the size is 15,000 gates.

According [11] for FPGA's the best way to synthesize this multiplier is when **infer_mult** is true, that means the synthesis tool will infer a multiplier.

The support of SPARC v8 instruction set is also a requirement for the system.

Extra hardware has to be configured in the integer unit in order to speed up branch address generation. This hardware is a separate branch address adder, and it is done setting option ***fastjump*** in integer configuration (*target.vhd* file).

The option ***fastdecode*** will improve timing by adding parallel logic for register file address generation.

The pipeline can be configured to have either one or two load delay cycles. It is clear using value one the pipeline is faster, therefore ***lddelay*** is set to 1.

4.1.4.4 Cache Configuration

The size of separate instruction and data caches has a very important effect on the player. Using TSIM was found a simple relationship: more cache equals more speed. It was tested with different configurations e.g. 2k i-cache 4k d-cache, 4k i-cache 2k d-cache and 4k i-cache 4k d-cache. The last one was by far the best one, and using Virtex target technology is the biggest configuration possible, due to the limitation of internal RAM on XCV-800 FPGA. The line size is set to 4 words/line.

4.1.4.5 AMBA Configuration

When the system was configured, there was still no extra hardware to speed up the decoder, nevertheless was clear it has to be implemented as an AHB master and have communication with software via APB bus. Accordingly the extension of AHB and APB buses was done, based on the old configuration of DDM, which has two AHB masters (IU and audio core) and an extra APB slave (audio core), after been imported to LEON-2 (see Section 4.1.1).

Audio core uses memory mapped registers range x800000200-218 while the MDCT core x800000300-318.

The result of this step give a configuration using LEON2-1.0.2a with 3 AHB masters and 2 extra APB slaves. The new master was tested in a very simple way, consisting in read eight elements from memory, add the constant 8 to each of them, and store the result again in memory. At the same time the DMA was tested as well. Software and hardware designs for this test are located in cvs.

The AMBA masters have the following priority (higher number is more important):

- Audio core (2)
- MDCT core (1)
- Processor (0)

The audio core has the highest priority, because depends on it if the music is in real time or not. When it has no more data, the other masters have to suspend activities and let the bus free for it. The next important task to have music to play is the MDCT. If it needs to access the memory, the processor has to wait. Hence the lowest priority is assigned to the processor. If enough internal RAM for the MDCT would be available, the bus would be free and the processor could continue executing the program. In other words, the system would be faster.

4.2 MDCT Core Design

Since Mini-MDCT is a subset of MDCT, and both are the result of the partitioning, there was no doubt to start implementing the function *mdct_backward()* of Ogg-Vorbis in hardware. A logical and intuitive approach is to start with implementing Mini-MDCT and in the case the system would not fast enough, and still hardware resources would be available, then extend the implementation to whole MDCT.

In this Section are presented the different aspects developing the Mini-MDCT core. Starting from the Ogg-Vorbis *mdct_backward()* function, then how the architecture was develop, continuing with the description of the practical work, and finally configuring software and hardware together.

Since the decoder uses only the inverse transform, is important to remark that the operation referred as MDCT in this Chapter is the Inverse MDCT introduced in Section 2.4.

4.2.1 MDCT Algorithm in Ogg-Vorbis

Different algorithms can be used to calculate the MDCT [34]. The one used by Ogg-Vorbis is described in [19]. Some functions of the MDCT such as the windowing are outside the *mdct_backward()* branch, and for this reason, implementing a different algorithm would mean to change substantially other functions of the decoder. The algorithm for Mini-MDCT implemented in hardware is based on *mdct_backward()* code.

The MDCT has the following requirements:

- Uses a block of size *n* (either 256 or 2048) as input.
- The result is a block of size *n*
- Uses $\frac{5n}{4}$ twiddle-factors as trig coefficients and $\frac{n}{4}$ constants as Bit reverse coefficients (not used by Mini-MDCT).

These requirements are shown in the block diagram shown in Figure 4.4.

In order to carry out this function, the MDCT core needs to store the next information in registers:

- Block size (256 or 2048)
- Address where the input data is stored
- Address where the result must be written
- Address where the twiddle factors (Trig vector) are located
- Address where the Bit Reverse coefficients are stored (not used by Mini-MDCT)

4.2.1.1 Twiddle Factors

Since the twiddle factors (Trig vector) are independent of the frequency, they can be calculated only once by software, and stored as constants in Look-up tables (LUT's), using the following formulas [19]:

for $k = 0..\frac{n}{4} - 1$

$$A_{2k} = cos(\frac{4k\pi}{n}),\ A_{2k+1} = -sin(\frac{4k\pi}{n})$$

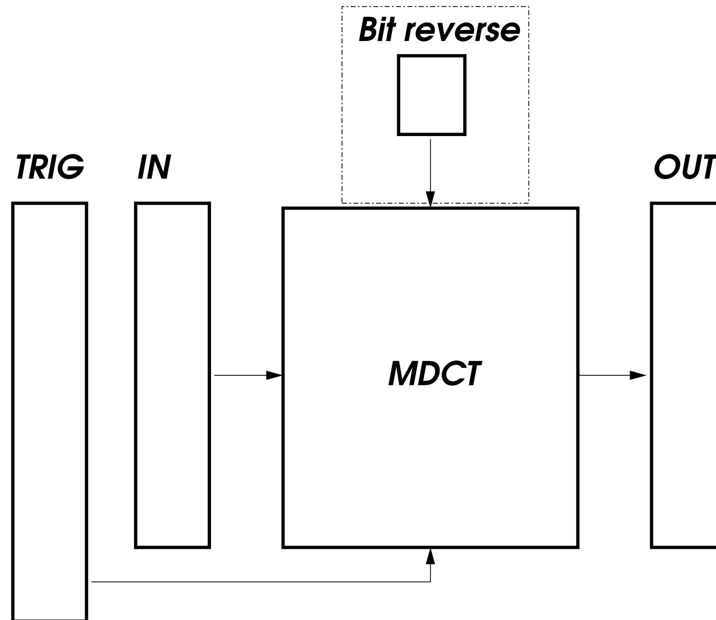

Figure 4.4: MDCT Block Diagram.

for $k = 0..\frac{n}{4} - 1$

$$B_{2k} = cos(\frac{(2k+1)\pi}{2n}), B_{2k+1} = sin(\frac{(2k+1)\pi}{2n})$$

for $k = 0..\frac{n}{8} - 1$

$$C_{2k} = cos(\frac{2(2k+1)\pi}{n}), C_{2k+1} = -sin(\frac{2(2k+1)\pi}{n})$$

Furthermore they have to be integerized. This option is optional in Ogg-Vorbis, but for Ogg-on-a-chip is required, since an integer version of Ogg-Vorbis is used. In Figure is shown how these values are stored in memory.

4.2.1.2 Mini-MDCT Calculation Process

The call graph of *mdct_backward()* is presented in Figure 4.6, and it is a detail of the whole Ogg-Vorbis algorithm presented in Figure 3.8. Mini-MDCT definition implements a part of *mdct_backward(), butterfly_first_stage()* and *butterfly_generic()*. In the same way whole MDCT should implement all presented functions.

The algorithm performs the following steps:

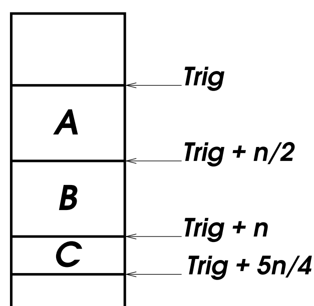

Figure 4.5: Twiddle factor LUT in memory.

1. **Pre-twiddling:** The frequency coefficients are multiplied by twiddle factors. This step is executed in *mdct_backward()*

2. **Butterflies calculations:** The second half of the final result is pre-calculated from result values of the previous step using butterflies structures. It uses *butterflies(), butterflies_1_stage(), butterflies_generic(), butterflies_32(), butterflies_16()* and *butterflies_8()* functions.

3. **Bit reversal:** The first half of the final result is pre-calculated reversing the bit order of the second half and multiplying it by constants depending on block size. It uses *bit_reverse()*. (Not in Mini-MDCT)

4. **Post-twiddling:** The final result is obtained multiplying again the pre-calculated result by twiddle factors. It is done also in *mdct_backward()*. (Not in Mini-MDCT)

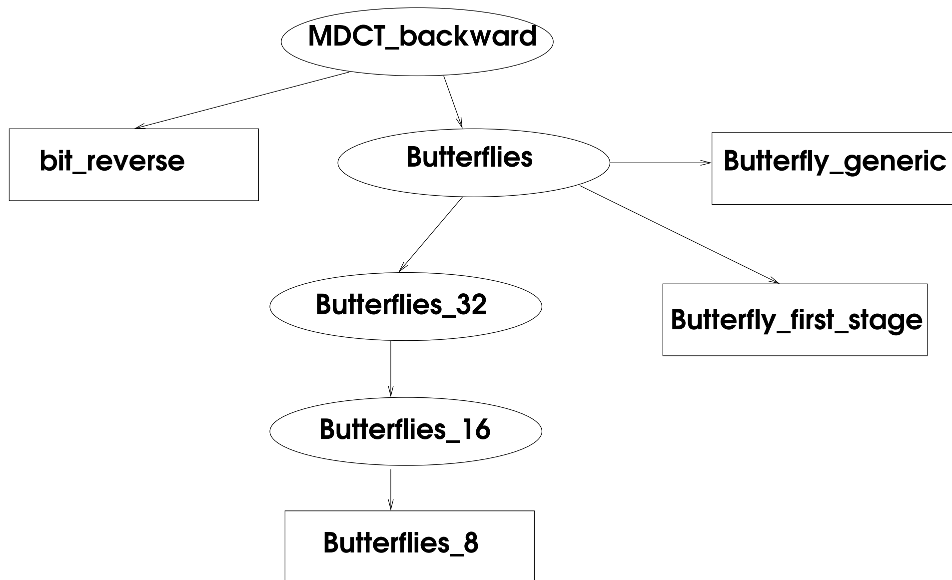

Figure 4.6: MDCT callgraph.

4.2.1.3 Pre-twiddling

The pre-twiddling process consists in multiply the first half of the original input vector by the twiddle factors, and store the result in special order in the second half of the output vector. Only multiplications and additions are required.

The process is presented graphically in Figure 4.7. It begins with 8 elements as input, from $\frac{n}{2} - 7$ to $\frac{n}{2}$. The odd part ($\frac{n}{2} - 7, \frac{n}{2} - 5, \frac{n}{2} - 3, \frac{n}{2} - 1$) is processed using the structure shown in Figure 4.8 and the result is stored crossed (i.e. first two elements below and last two above) from $\frac{3n}{4}$ to $\frac{3n}{4} + 3$. In the same way the even part ($\frac{n}{2} - 6, \frac{n}{2} - 4, \frac{n}{2} - 2, \frac{n}{2}$) is calculated with a similar structure shown in Figure 4.9 and the result is stored in-place from $\frac{3n}{4} - 4$ to $\frac{3n}{4} - 1$. After processing the first 8 elements, the next 8 (from $\frac{n}{2} - 15$ to $\frac{n}{2} - 8$) are the inputs, but this time the result of the odd part is stored from $\frac{3n}{4} + 4$ to $\frac{3n}{4} + 7$, and the result of the even part in $\frac{3n}{4} - 8$ to $\frac{3n}{4} - 5$. It can be observed that the input is moving towards the top, the even result towards $\frac{n}{2}$, and the odd result towards the bottom (element n). When those values are reached, the pre-twiddling process finishes.

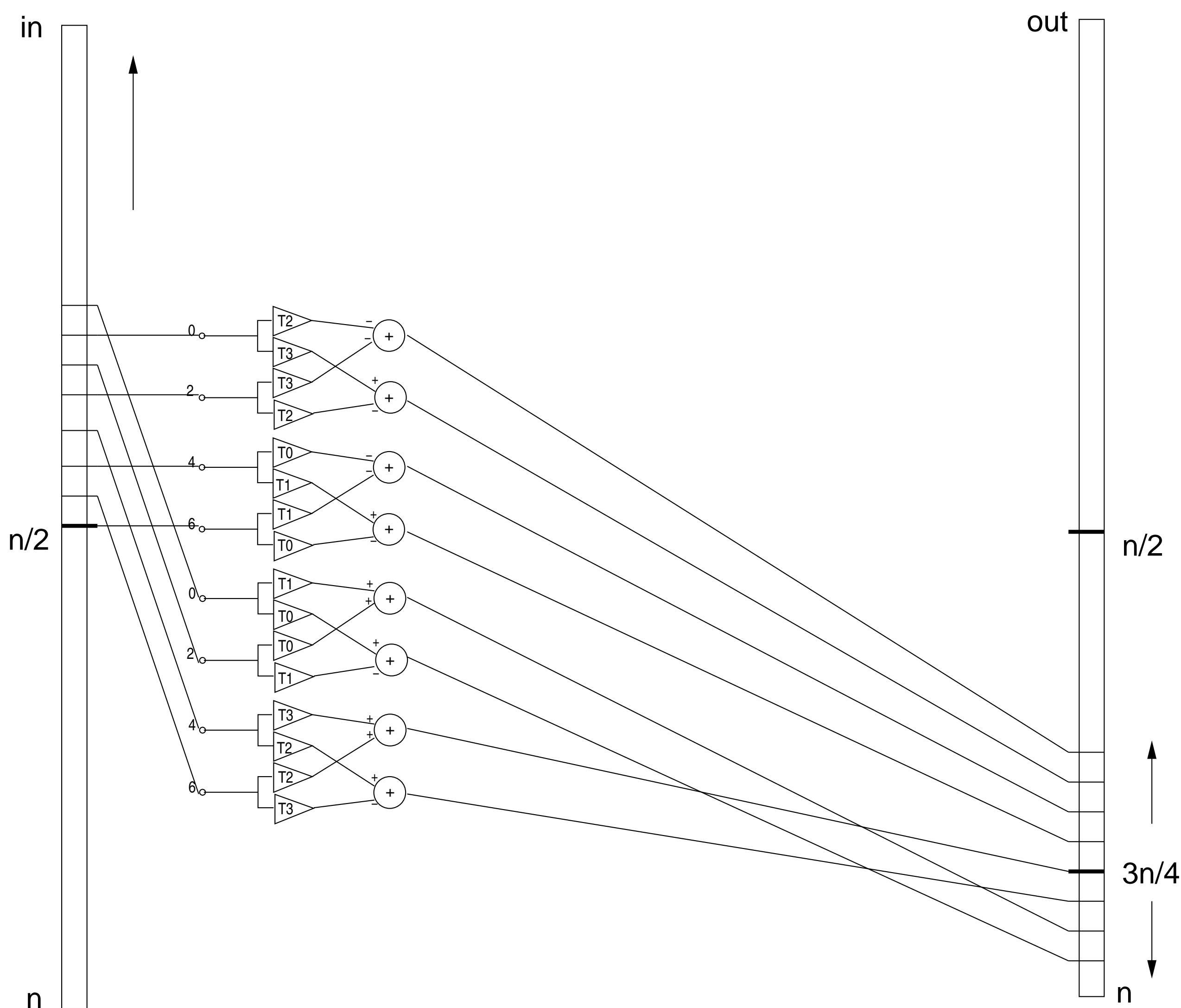

Figure 4.7: Pre-twiddling process

Looking again at Figure 4.8 the arithmetic structure can be observed. Triangles represent multipliers. They multiply the input by the Trig constant marked inside the triangle. The circles are adders, and they perform an addition using the signs of the inputs. For example, to calculate output element 3, the input element 3 is multiplied by T1 and this result is subtracted from the multiplication of element 1 by T0. Comparing with Figure 4.9 the differences are the order of the constants in the multipliers, the signs at the adders and the order of the output.

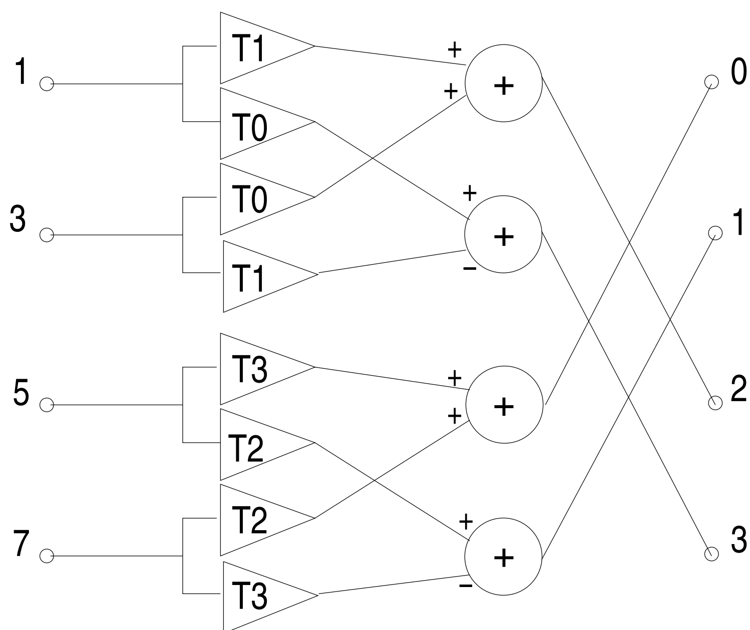

Figure 4.8: Odd part process

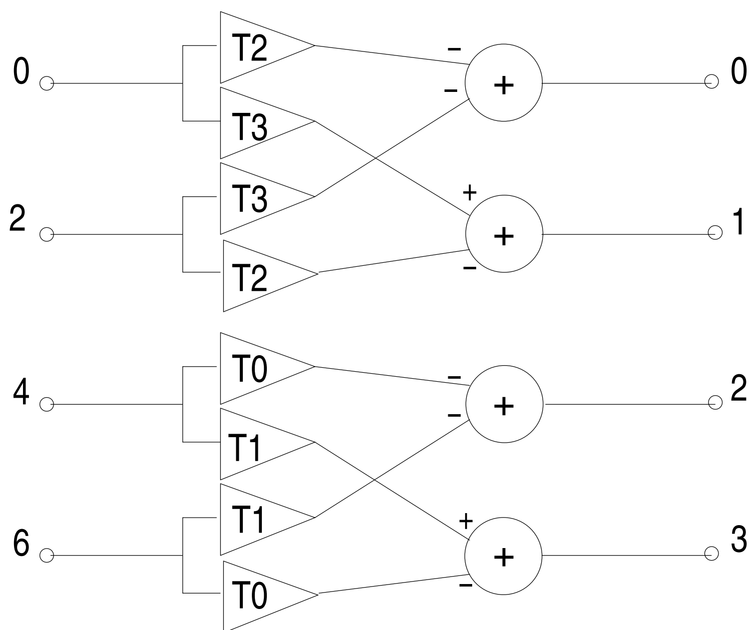

Figure 4.9: Even part process

4.2.1.4 Butterflies calculations

After pre-twiddling, the second half of the output will be the input for the butterflies calculation step. The result will be stored in the output's second half again, repeating this process *m* times, where *m* is the number of *stages.* In each stage, the number of inputs is reduced by 2, until the number of inputs is 8. Since $8 = 2^3$, the number of stages is $m = (log_2 n) - 3$. This fact can be appreciated in Figure 4.10.

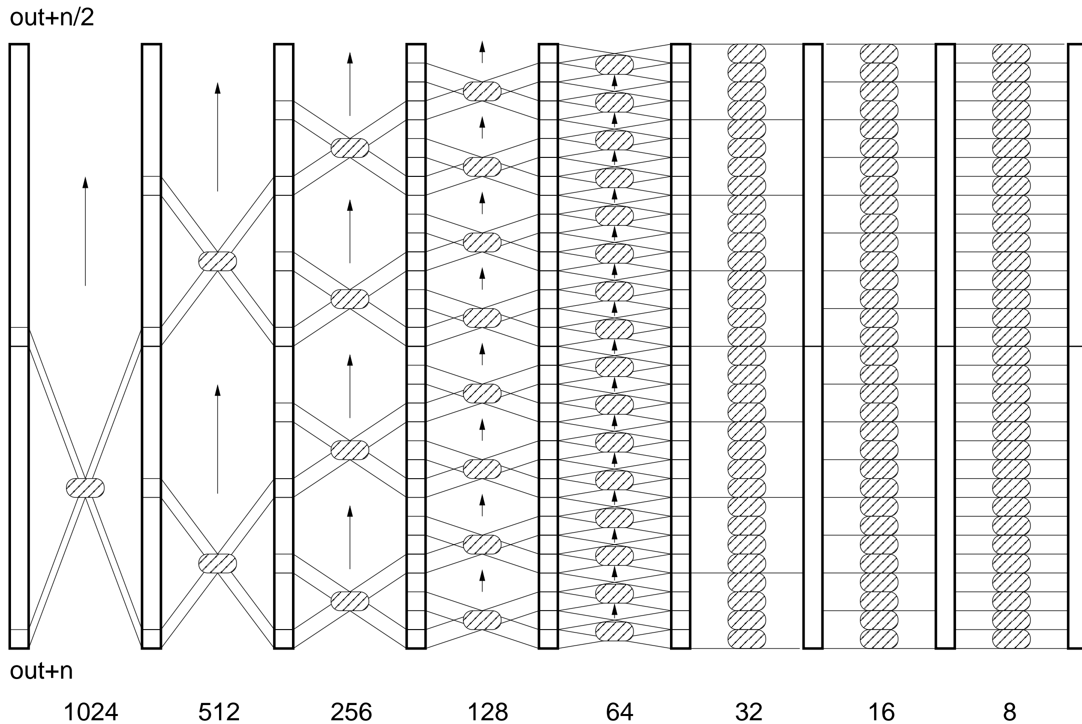

Figure 4.10: Butterflies for big block (2048 elements)

Between two stages, there is at least one butterfly which takes two different groups of 8 elements as input, and returns 2 groups of 8 elements as output. After processing both inputs the next two groups are processed. The effect is that the butterfly is "flying" towards the top. If there is no other butterfly, it will reach the top. In the case other butterfly is located in the same stage, the first one will stop when reaches the start position of the second one. In each stage the number of butterflies is increasing by two, and the distances group 1 - group 2 input and group 1 - group 2 output are decreasing by two as well. These distances become zero at the third stage from right to left.

The arithmetic operations executed for each butterfly are shown in Figure 4.11. The structure is the detail of the little cloud in Figure 4.10, which is exactly the so called "butterfly" [2], with 2 inputs (each one of 8 elements) and 2 outputs (each one 8 elements). Again there are only multiplications and additions.

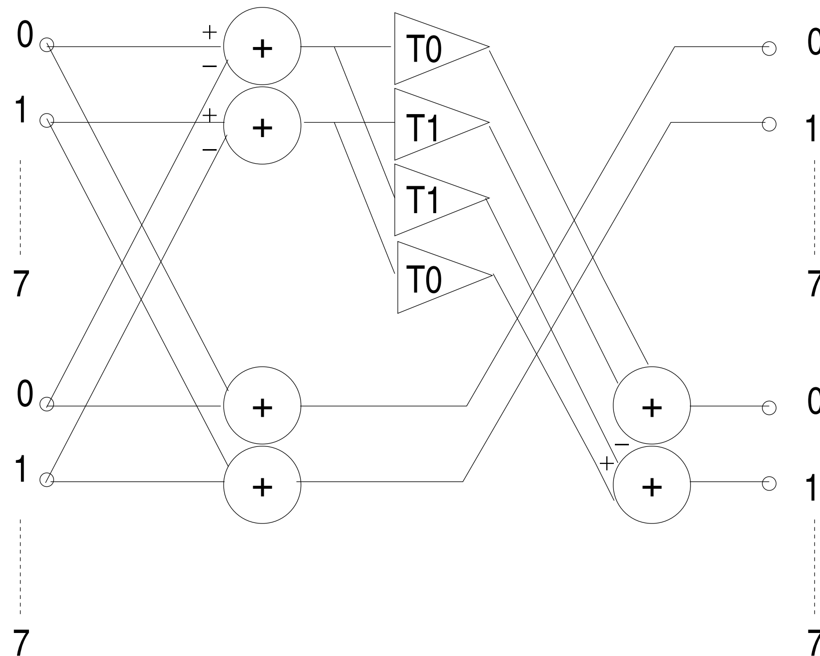

Figure 4.11: Basic butterfly

4.2.1.5 Remarks

Mini-MDCT finishes when the two inputs of a butterfly are not separate anymore. It is in Figure 4.10 at column marked as 32 (3th stage), and the reason is as follows: The software is highly optimized, thus although the butterflies process can be calculated only with function *butterflies_generic()*, it split the process in *butterflies_first_stage()*, *butterflies_32()*,*butterflies_16()* and *butterflies_8()* (see Figure 4.6). When the data are in consecutive addresses, they can be loaded in cache and there been processed until the end. In other words, before stage 3 a whole stage is processed and then proceed with the next stage, but from stage 3 data are processed in row until stage 1 (the final stage). In this way the process is much faster, because data are in cache until the end result is calculated. In the next Section will be explained the architecture of the MDCT core, and will be seen, that there are

[2]Because the shape looks like "wings" of a real butterfly.

no way to have a kind of cache there (because of the reasons explained in 4.1.2 and 4.1.4.4), therefore a DMA has to be used, and it is not needed much faster than the software.

Mini-MDCT can be calculated with the arithmetic elements presented in Figure 4.11, because pre-twiddling and butterflies calculations use the same. Those are 4 multiplier and 6 adders.

4.2.2 MDCT Core Architecture

So far is identified how the algorithm to implement inverse MDCT introduced in Section 2.4 works. Now it has to be translated into hardware structures.

In 4.1.4.5 was explained how the empty MDCT is connected to AMBA, and in 4.2.1.4 was mentioned that the MDCT can be implemented using finite arithmetic elements (multipliers and adders). Besides these two items extra logic has to be added to generate addresses, transfer the desired input values as operand for the arithmetic elements, and store the results in the right position in memory. In other words, 3 elements conforms the architecture of the core:

- **AMBA Interface.-** Provides the connection with AMBA to have access to the RAM via AHB, and support the communication with software holding memory mapped registers via APB.
- **Arithmetic Unit.-** A set of arithmetic elements, 4 multipliers and 6 adders, to process data according Ogg-Vorbis MDCT algorithm.
- **Control Unit.-** It is the director of the orchestra. Commands the activities of the AMBA interface, and uses the arithmetic unit to calculate the MDCT.

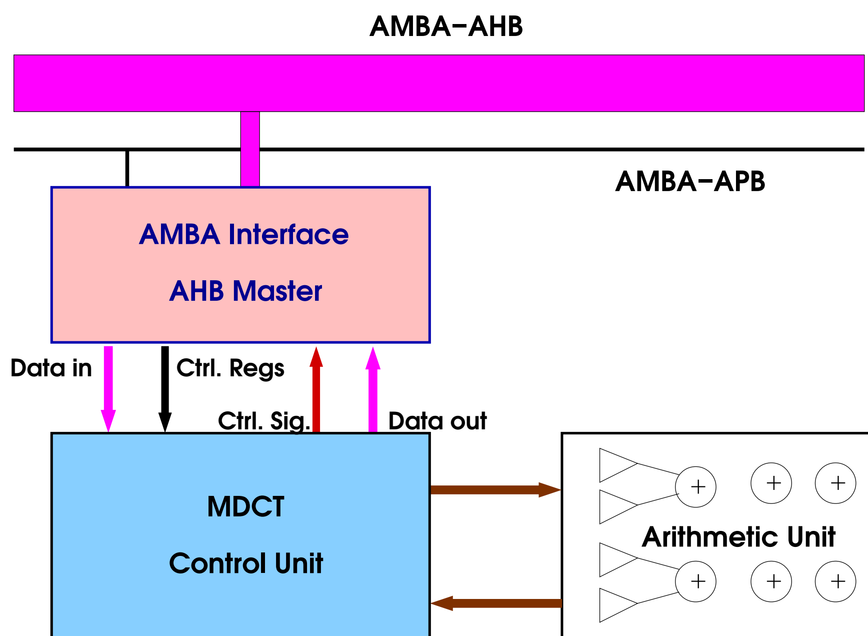

Figure 4.12: MDCT core architecture.

Figure 4.12 shows a block diagram of the architecture. It works as follows: Software writes required information to process a block (block size, input address, output address and trig address) into memory mapped registers through APB bus. They are received by the AMBA interface and stored physically in registers. When the start signal is sent, again via APB, the control unit calculate addresses using the arithmetic unit, and indicates the AMBA interface where the required data are located in memory. Furthermore AMBA interface gets the data and they are stored in one buffer on control unit. Meanwhile the control unit can calculate the address to store the data after processing with the help of the arithmetic unit. Then the data are feed into the arithmetic unit, and the result is stored again in a different buffer, to proceed to give the signals in order to store the processed data into RAM again. Once the whole block is processed, the control unit falls in a waiting state for next block.

After the description of this process can be observed the speed limitation due to the necessary DMA, which is the bottle neck. The core works as fast as the DMA is performed. At the same time it blocks the AHB bus, meaning that nobody else can use it. For this reason even the processor has to wait until the MDCT core finishes, in order to read again the memory and execute new instructions. Only the audio core is allowed to interrupt MDCT calculations. For the reasons exposed before, if enough internal memory could be available for MDCT core, the process

could be faster and the processor could do something else in between.

4.2.2.1 AMBA Interface

The interface has connection with both buses in AMBA; APB and AHB.

APB slave An APB slave is used as software-hardware interface, in order to receive the required information to process MDCT (see 4.2.1) as well as the starting signal, and stores them into registers, using range 0x800000300-0x800000318 They are:

```
Control register 0x80000300
bit 0:  MDCT- core, 0 = off, 1= on
bit 1:  not used
bit 2:  0=irq disabled, 1=irq enabled
bit 3:  irq (read only)
Block size 0x80000304
bit 0:  0=256, 1=2048
Trig address 0x80000308
Read Start address 0x8000030C
Write Start address 0x80000310
Status register 0x80000314 (read only)
bit 0:  0=ready, 1=busy
bit 1:  0=reading, 1=writing
Current Memory address 0x80000318 (read only)
```

The APB slave is configured with index number 13 in APB slave vector in *target-ooac.vhd* file used by LEON.

AHB Master As DMA was identified as bottle neck, and the AHB master carries out the DMA, it is clear the AHB master must be as fast as possible.

According [11] RAM access can be done (read or write) in three clock cycles: In the first one the address must be valid, in the second one the data come out (read case) or must be valid (write case) and the third one is a lead-out cycle to prevent bus contention due to slow turn-off time of memories. The lead-out cycle is only required when the addresses are not consecutive.

AMBA specification [20] allows burst transfers to improve the bandwidth of the memory bus. It is only useful when consecutive addresses are accessed. The difference with the normal timing, is that the lead-out cycle will be added only after the last element of the burst is processed. The memory controller in LEON uses this feature to load or write a whole row from/to the cache.

To be fast enough the AHB master of MDCT core must use this feature as well. Looking in Figure 4.11 the minimum number of input elements to get a result is 4. Hence the maximum burst size is 4, but it can be smaller, for example only for read constants.

The control signals to AMBA interface shown in Figure 4.12 inform how many elements have to be processed, the operation to do (read or write), where or from which buffer position have to be read/stored, if the MDCT has finished, the start address of the burst and the increment for next address. For this reason an adder to increment current address is required.

AMBA arbiter has to be informed about burst cycles, and it is done using AMBA signals *htrans, hburst,* and *hsize.* For example, whenever a burst access is initiated, *htrans* is set to *nonseq* because the access is no sequential, *hsize* to 32 (the length of the word) and depending on the increment for the next address *hburst* to *incr4* or *incr8*. Once the first element is processed, *htrans* has to be set to *seq*, signaling a sequential access for up coming elements. For more details about AMBA signals, please refer to [20].

In VHDL code, the entity *mdct* is the AMBA interface. Behind are connected the control unit and the arithmetic unit, but they are invisible for the rest of the platform, therefore from processor's point of view, who performs the MDCT is the entity attached to AMBA. After processing a burst, the AMBA interface sends to the control unit the *dataready* signal.

4.2.2.2 Control Unit

Due to AHB master determines the speed of the core, the control unit has to be DMA oriented. The control unit was implemented using a final state machine (FSM) and uses a signal based on *dataready* named *stateclk* to change from state to state, which is zero while the MDCT is not active, and equal to *dataready* during activity. Hence the FSM is a synchronous machine.

In Figure 4.13 is presented the FSM. Fourteen different states are defined as follows:

- S0: Wait for activation
- ST: Read a group of twiddle factors
- S1-S12: Data process

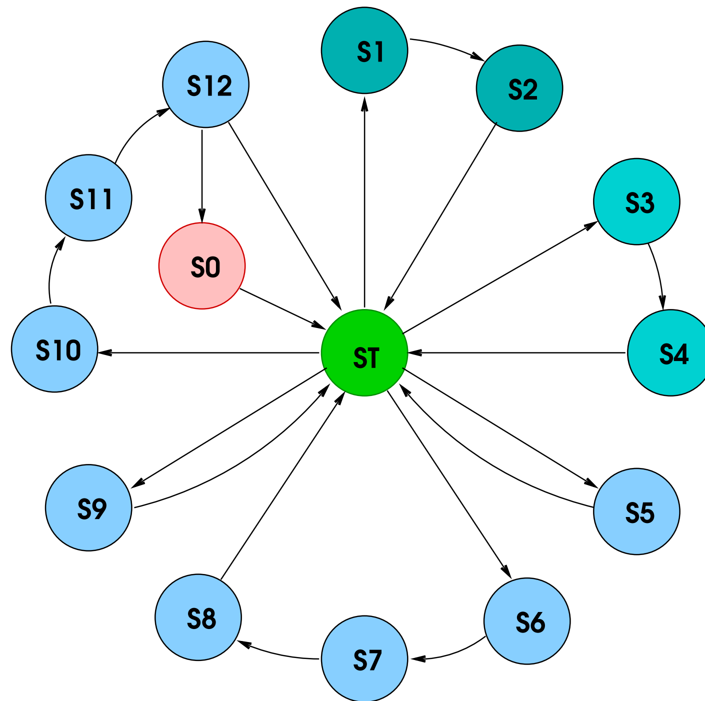

Figure 4.13: FSM of control unit.

In the same Figure, 3 different loops might be recognized among data process states:

1. ST,S1,S2
2. ST,S3,S4
3. ST,S5,ST,S6,S7,S8,ST,S9,ST,S10,S11,S12

Referencing to the process described in 4.2.1.3 can be stated that loop 1 corresponds to the odd pre-twiddling part, and loop 2 to the even pre-twiddling part. Firstly the loop 1 will be executed so many times as required by the block size, then loop 2 will be execute exactly the same times. Of course ST is always a read state. S1 is a read state again, and the result of the operations is written during S2. Nevertheless during both states data arithmetic calculations are done. S2 is read state and S3 write state for loop 2 .

In a similar way, the process described in 4.2.1.4 is elaborated by loop 3 so many times as the block requires. On the original *mdct_backward()* function (see Figure 4.6) the functions *butterflies_generic()* and *butterflies_first_stage()* correspond to loop 3 as well. In order to understand deeply the complexity of loop 3,

the original *mdct_backward()* code has to be compared direct with VHDL code of *mdct_syn.vhd* file. In this file are commented whenever possible, the C lines that the VHDL code implements.

The FSM in VHDL has two different combinational processes. One for the arithmetic of MDCT, and the second one for address calculations and send control signals to AMBA interface.

In ST state, with a rising edge of *stateclk* signal, the FSM will go to the state stored in *nextstate* internal register.

Finally, when all data have been already processed, the FSM goes into S0 and waits for the next activation.

The states representation is done in VHDL with a user-defined enumerated type, according the FSM guidelines described in [17].

```
type mdct_state is (s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,sT);
```

Then they should be encoded by the synthesis tool. SYNPLIFY PRO uses either gray code one-hot coding.

The control unit is represented in VHDL code by the entity *mdctctrl* in *mdct_syn.vhd* file.

4.2.2.3 Arithmetic Unit

The arithmetic unit is composed by different two-inputs arithmetic elements, such as multipliers and adders. This elements have to comply with the arithmetic precision established in Ogg-Vorbis decoder, which is for variables type long (32 bits). As mentioned in 4.2.1 the MDCT can be calculated using only the arithmetic elements shown in Figure 4.11. It could be even possible to perform the same calculation with less arithmetic elements, however in that case the bottle neck of the process would not be anymore the DMA, but the arithmetic process. The only limitation for include more arithmetic elements is the size available on the FPGA. For this reason the elements shown in Figure 4.11, 4 multipliers and 6 adders, were implemented in the arithmetic unit. Adding more arithmetic elements won´t speed up the process, because it is constricted by the DMA.

Due to computer arithmetic has been so successful that it has, at times, become transparent, is not necessary to reinvent the wheel. Arithmetic circuits are no longer dominant in terms of complexity; registers, memory and memory management, instruction issue logic, and pipeline control have become the dominant consumers of chip area in today's processors [22]. Modern synthesis tools use libraries to synthesize arithmetic operators easily, meaning that no model at gate level is required. The synthesis tool used in this project, Synplify pro, is one of those tools, hence it was no need to redesign multipliers and adders. In the

VHDL code the arithmetic unit is not an entity like the control unit or the AMBA interface. The arithmetic operations are located in the concurrent process at the end of the *mdctctrl* entity. During synthesis the tool will infer how to implement these operations in hardware. In this way the best results are obtained for FPGA's (not necessary for ASIC's), even LEON uses this feature for the multiplier in the Integer Unit [11].

In file *mdctlib.vhd* the arithmetic input and output types are declared. Later on in *mdct_syn.vhd* registers with that type are declared, which are clocked by the system clock. That means for the synthesis tool, the multiplications and additions have to be performed in one clock cycle, but the size is not restricted.

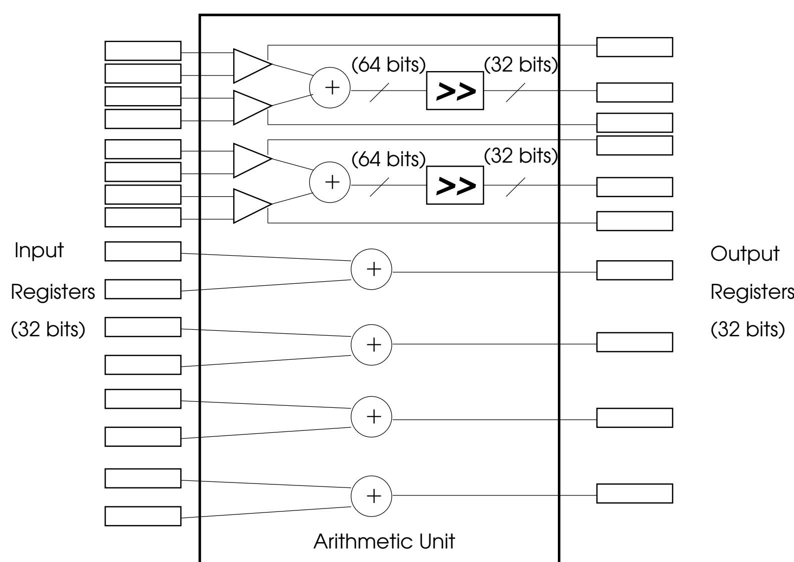

Figure 4.14: Arithmetic Unit.

In Figure 4.14 is shown the structure of the arithmetic unit. Two adders can be observed after the four multipliers, and 4 adders connected from input to the output. The adders after the multipliers are 64 bits long, because the result of the multiplication has this length. These 64 bits adders are only connected to the multipliers, therefore is not possible for control unit to use them. Each of those adders has a signal add_fun, which signalizes if the result of the second multiplier has to be added or subtracted from the result of the first multiplier.

In the case of the path multipliers-adders, the result is 64 bits long. Consequently it has to be returned again to 32 bits in order to be stored in memory. Hence two shifters are inserted with input 64 bits and output 32 bits. This is done by function *MULT_NORM* declared in file *mdctlib.vhd*. It shifts the 64 bits regis-

ter *TRIGBITS* to the right. *TRIGBITS* is a constant declared in the same file, and for the used integer Ogg-Vorbis version is set to 14.

The rest 4 adders are used to calculate values of the MDCT or calculate addresses. They can be accessed by the control unit with the input registers.

4.3 Simulation, Synthesis and Test

This Section is the last one about hardware, and will discuss the process how to produce the files required by the XSV-800 board. It is no easy to separate simulation phase from the design phase. Here it is done only for clearness, but in reality the simulation and the design phases ran together.

The workflow is presented in Figure 4.15. At the top can be appreciated the VHDL files of the LEON model (start point), and at the bottom the XSV-800 board (end point). Here can be seen how the result of the previous phases on the hardware part are together. In first place the configuration files described in Section 4.1.1 (result of configuration process. See Section 4.1.4) have to replace the original configuration LEON files. Then the MDCT VHDL file (result of core design process) is added. After that is only missing the result of the whole system, which is actually the result of this phase. Two branches can be observed: to the left towards the simulation (simulation path), and to the right towards the implementation on hardware, going through synthesis (synthesis path). The order is simulate the system firstly and then proceed to the synthesis. Tools used are MODELSIM for simulation, SYNPLIFY PRO for synthesis, and XILINX ISE 4.1I. In Figure 4.15 rectangles with round corners represent tools with Graphical User Interface (GUI), ellipses are line commands and rectangles are files.

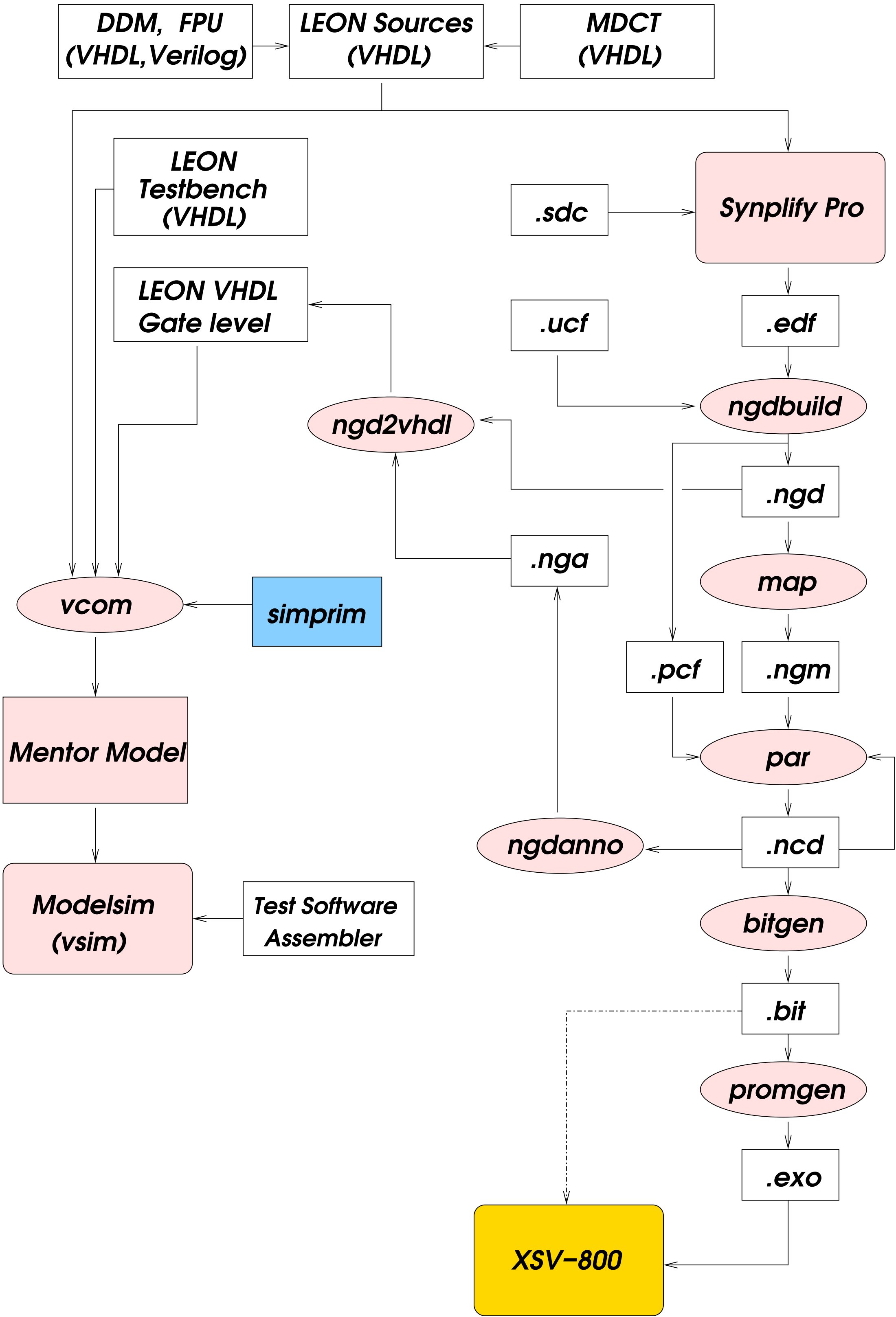

Figure 4.15: Hardware workflow

4.3.1 Simulation Branch

Simulation path is the main support for the design process. Once the simulation succeeds, is possible to proceed with the synthesis. In *cvs* directory (*hardware/vhdl_designs*) there are two different designs of the MDCT core:

- *mdct.vhd* is a simulable model of the whole MDCT. It uses *mdctlib.vhd, mdctcomp.vhd, mdctrom256.vhd* or *mdctrom2048.vhd.*
- *mdct_syn.vhd* is the simulable and synthesizable version of Mini-MDCT. It uses *mdctlib.vhd.*

LEON platform is ready to be simulated. It comes with testbenches where parameters of the system can be adjusted, such as the clock frequency and the data bus length. After unpacking LEON and calling *make* on top directory, the *work* directory containing the simulation model will be created. It can be manually done using command **vlib** *work* to create the directory and then **vcom** <LEON_files_list>[3]

The above described operation compiles the testbenches as well. Then Modelsim can be started by calling **vsim** command. After selecting the test bench configuration (for Ogg-on-a-chip is always *tb_32_0_32*) the simulation can be started with the *run* command (or the *run* button on GUI).

To simulate Ogg-on-a-chip system, macro files to compile the model are located in *cvs* directory under *hardware/modelsim*. Macro *compileMDCT_syn.do* is used for the Mini-MDCT. It compiles the whole platform and testbenches.

Ogg-on-a-chip requires software for simulation. Simulating the real player takes too long (a couple of days), for this reason special software has to be used. It is only the software-hardware interface via memory mapped registers written in SPARC assembler. Different programs are located in *cvs* (*hardware/test-software*). They call MDCT (256,2048) and the audio core. In order to execute a program it has to have the *dat* extension, and be renamed as *ram.dat* in LEON's */tsource* directory.

An assembler program can be compiled using LECCS as follows:

```
sparc-rtems-as -o <file>.o <file>.s
sparc-rtems-ld -s -Ttext 0x40000000 -o <file> <file>.o
sparc-rtems-objdump -s <file>
```

The result is *<file>.dat* and to be executed must be copied to */tsource/ram.dat*, and then start the simulation as described before. In Figure 4.16 can be observed the moment when the MDCT core starts to work after configuring the control

[3]The files have to be exact in the same order as in *Makefile*

registers. In other words, at the left of the cursor (vertical line) takes place the APB communication with software and at the right starts the AHB execution.

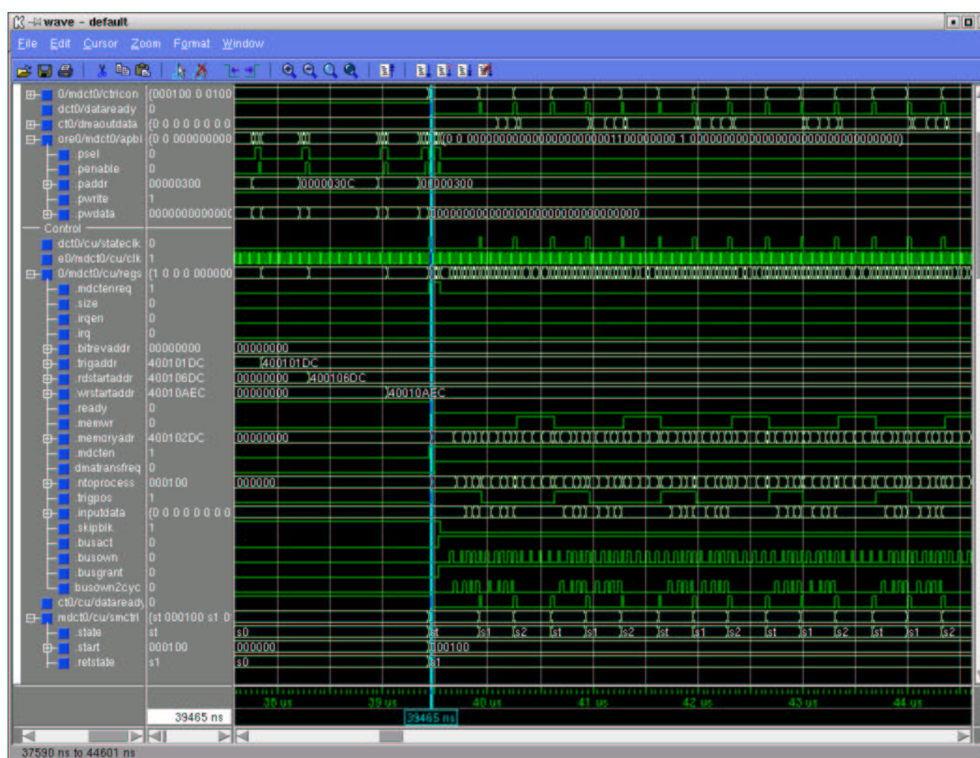

Figure 4.16: Screen-shot of Modelsim

4.3.1.1 Post-synthesis Simulation

Some times there are no errors on the RTL (register transfer level) simulation but after synthesis the design doesn't work on the board. Therefore simulations at different levels (register level, gate level) are not only desired but required. In Figure 4.15 two paths connect right (synthesis) with left (simulation) branches, and they represent post-synthesis simulation.

The first path appears after creating the *ngd* file with **ngdbuild** of the right branch, then is possible to convert it again into VHDL code using **ngd2vhdl** and simulate the model with the *simprim* libraries.

The second one is after Place and Route (PAR), using the *ncd* file to create a *nga* file with **ngdanno,** and the *nga* file is the input for **ngd2vhdl** in order to have again a VHDL description.

In both cases the result is only one *vhd* file, [4]

which can be compiled using *simprim* libraries. Then the testbenches have to be recompiled. The macro *tb-compile.do* can be used for this purpose.

4.3.2 Synthesis Branch

The right branch of Figure 4.15 starts with SYNPLIFY PRO. This synthesis tool has a GUI (see Figure 4.17), but it can be used at the command line as well. Is important to remark, that not all simulable designs are synthesizable, because the synthesis tools use a subset of VHDL. The VHDL subset of Synplify Pro is described in [17].

A project containing different files is synthesized according given constraints in *sdc* file. It has optionally the feature to add attributes in this file and there is no required to re-compile the project if some attribute is changed after compilation. If these attributes are in the VHDL code, then it has to be re-compiled. The result is the netlist in one *edf* file.

After this step the process becomes technology specific. Then the XILINX ISE 4.1I tools are used.[5]

With a user constraint file (*ucf*) **ngdbuild** generates the *ngd* file, which is the input for **map** or can generate VHDL code (see Section 4.3.1.1). Afterwards **map** generates a *ngm* file as input for **par**.

Place and route is the critical step for timing. Because of that, if after executing **par** the time constraints are not met, the output can be the input again, until meeting the constraints [18]. The *ncd* output file can be simulated again as described in Section 4.3.1.1.

Finally, once the timing is correct a *bit* file is generated. This *bit* file could be downloaded already to the board, and program the FPGA, if LEON was synthesized with *softprom* booting type [11]. For our project this feature was not used, instead the flash RAM on the board was programmed using an *exo* file, in this way the board works stand-alone. Each time that the power is on, the FPGA is programmed from flash RAM.

[4]The original model is composed by *65* vhd files.

[5]The tools ngd2vhdl and ngdanno described in Section 4.3.1.1 also belong to Xilinx ISE

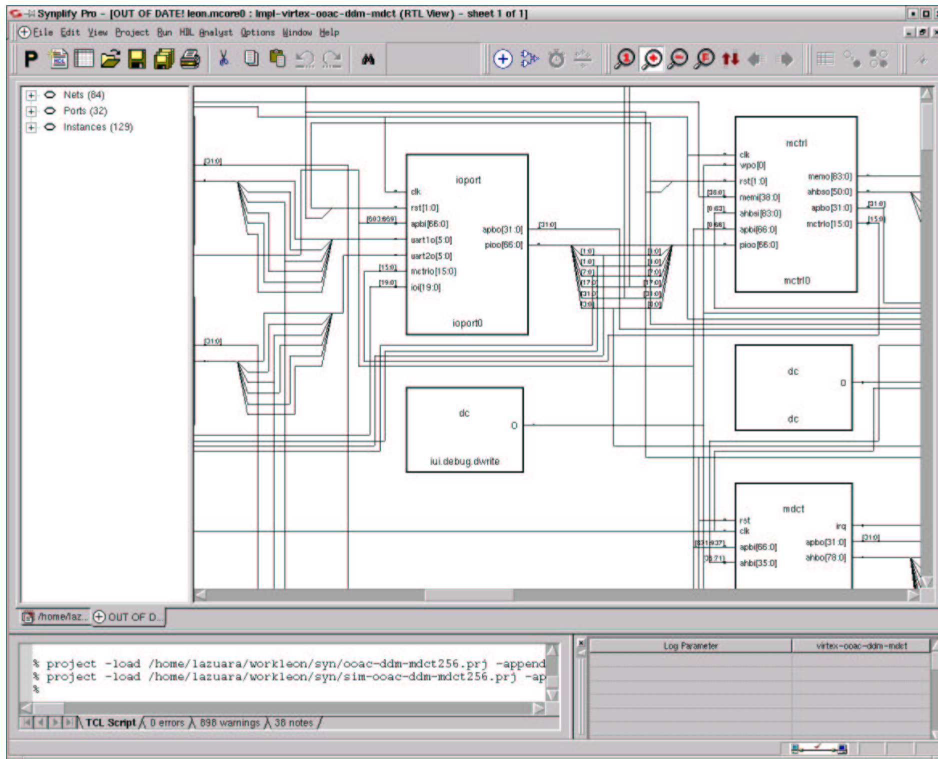

Figure 4.17: Screen-shot of Synplify Pro

4.3.3 Hardware Test

Once the *exo* file containing the hardware design was downloaded to the board with help of the vendor provided XSTOOLS[6], and before testing with the real player, the correctness of the hardware needs to be ensured. It is done using the test programs for the MDCT TSIM module described in Section 3.5.

Two main issues are tested: software-hardware communication, and the functionality of the MDCT core. If the right inputs (See Section 4.2) are read by the hardware, and it start to work at the right moment, the first part of the hardware-software communication is confirmed. Consequently the numerical result has to be compared with the result calculated by the original *mdct_backward()* function. If it is correct, both, second part of hardware-software communication and hardware functionality, are confirmed.

[6]The version used was ported to Linux by Daniel Bretz. More details in [2].

4.4 Final System Test

Testing the system is done by downloading the hardware design (*exo* file) and player (*exo* file) to the board. This is the point where hardware and software join each other in Figure 1.2.

The hardware *exo* file, which has all features to work together with software is located under *hardware/fpga_designs/ooac-v1.exo* in *cvs*.

Table 4.1 is an extension of Table 3.3. In the first column are the results given by TSIM, and in the second column are presented the results on real hardware. The hardware is speeding up the optimized player by a factor of 1.18. It isn't still real-time (should be 15 sec.) at the sample frequency used for this test (48 kHz. See Section 4.1.2). On the board, both versions (only software and software+hardware) present a gap, but in hardware+software version the gap is notable smaller.

If enough internal RAM would be available on the FPGA, it could be 2-3 times faster. Besides of that, the Mini-MDCT could be extended to whole MDCT, in order to achieve the high quality music in real-time on this particular low computation-power system (XSV-800).

Table 4.1: Software-Hardware result comparison at 25 MHz

Program	**TSIM result (seconds)**	**XSV-800 result(seconds)**
Final Player	19.42	21.1
Final Player + Mini-MDCT	15.41	17.9

A different test was done using 24kHz as sample frequency, and this is properly executed in real-time by the hardware-software cooperation.

Chapter 5

Conclusion

An audio decoder using Ogg-Vorbis was designed and implemented as embedded system by using Hardware/Software co-design techniques, therefore the goal of Ogg-on-a-chip project was achieved. The main results of this project can be summarized as:

- Some applications for desktop computers can be implemented as an embedded system after software optimization and addition of extra hardware.
- A demonstrator using a particular low computation-power system was built, according the result of the partition.
- The Hardware/Software partition depends on the target technology. For the demonstrator built during this project, the results of the partition are:
  - an integer version of Ogg-Vorbis player on software side
  - an AMBA-compatible Mini-MDCT core on hardware side.
- Projects using open source are feasible and reliable, since Ogg-on-a-chip uses open source software and hardware elements: Ogg-Vorbis player, RTEMS and LEON.
- The knowledge gained during this project is extensible to different fields, not only multimedia applications.

Appendix A

CVS

CVS is the Concurrent Versions System, the dominant open-source network-transparent version control system. CVS is useful for everyone from individual developers to large, distributed teams.

Its client-server access method lets developers access the latest code from anywhere there's an Internet connection. Its unreserved check-out model to version control avoids artificial conflicts common with the exclusive check-out model. Its client tools are available on most platforms.

CVS is used by popular open-source projects like Mozilla, the GIMP, XEmacs, KDE, and GNOME.

In the division Computer Architecture the *cvs* repository is accessible under:

http://rai16.informatik.uni-stuttgart.de/cgi-bin/cvsweb/

By selecting Ogg-on-a-chip on the combo box, the following modules will be displayed:

- Hardware: LEON source code, synthesized designs, hardware VHDL and exo files.
- Papers: Important papers and documents for Ogg-on-a-chip.
- Presentation: Presentation and figures in Open-Office.
- Report: Documents and figures used in this report.
- Software: Ogg-Vorbis player, tools, RTEMS and TSIM modules.

It is possible to use cvs from the command line[1] using the next environment variables:

[1] More information about CVS and CVS commands in www.cvshome.org

CVSROOT=<*username*>*@rax3.informatik.uni-stuttgart.de:/usr/local/cvs/oggonachip*

CVS_RSH=*ssh*

Contribution List

- Chapter 1: co-written
- Chapter 2
  - Section 2.1: Pattara Kiatisevi
  - Section 2.2: Pattara Kiatisevi
  - Section 2.3: Pattara Kiatisevi
  - Section 2.4: Luis Azuara
  - Section 2.5: Luis Azuara
- Chapter 3: Pattara Kiatisevi
- Chapter 4: Luis Azuara
- Chapter 5: co-written
- Appendix A: Luis Azuara

Bibliography

[1] Chris Bagwell, *Sox - Sound eXchange*, `http://home.sprynet.com/~cbagwell/sox.html`, 2002.

[2] Daniel Bretz, *Digitales Diktiergeraet als System-on-a-Chip mit FPGA-Evaluierungsboard*, Master's thesis, Institute of Computer Science, University of Stuttgart, Germany, February 2001.

[3] Mike Coleman, *Vorbis Illuminated*, `http://www.mathdogs.com/vorbis-illuminated/`, 2002.

[4] Dan Conti, *iv-dev, Integerized Vorbis Library*, `http://sourceforge.net/projects/ivdev/`, 2002.

[5] OAR Corp, *RTEMS Documentation*, 2002.

[6] OAR Corporation, *RTEMS Web Site*, `http://www.oarcorp.com`, 2002.

[7] XESS Corporation, *XSV Board Manual*, `http://www.xess.com`, 2001.

[8] ______, *XESS Web Site*, `http://www.xess.com`, 2002.

[9] Jiri Gaisler, *LEON DSU Monitor User's Manual*, `http://www.gaisler.com/`, 2002.

[10] ______, *LEON Web Site*, `http://www.gaisler.com/`, 2002.

[11] ______, *The LEON-2 User's Manual*, `http://www.gaisler.com/`, 2002.

[12] Guenter Geiger, *Man page of play command*, 2002.

[13] Free Software Foundation Inc., *GNU Autoconf manual*, `http://www.gnu.org/manual/autoconf/index.html`, 2002.

[14] ______, *Man page of gprof command* , 2002.

[15] Rational Inc., *Rational Software – Purify*, `http://www.rational.com/products/pqc/index.jsp`, 2002.

[16] Red Hat Inc., *Newlib C Library*, `http://sources.redhat.com/newlib/`, 2002.

[17] Synplicity Inc., *Synplify Pro Reference Manual* , `http://www.synplicity.com/`, 2001.

[18] Xilinx Inc., *Development System Reference Guide* , `http://www.xilinx.com/`, 2001.

[19] B. Edler Kh. Brandenburg, Th. Sporer, *The use of multirate filter banks for coding of high quality digital audio*, 6th European Signal Processing Conference (EUSIPCO) (1992).

[20] ARM limited, *AMBA Specification 2.0*, `http://www.arm.com`, 1999.

[21] Sun Microsystems, *microSPARC-IIep User's Manual*, `http://www.sun.com`, 1999.

[22] Berhooz Parhami, *Computer arithmetic, algorithms and hardware designs*, Oxford University press, 2000.

[23] Nicolas Petre, *Fixed-point Vorbis Library*, `ftp://ftp.arm.linux.org.uk/pub/linux/arm/people/nico/vorbis/`, 2002.

[24] Florent Pillet, *Kprof Web Site*, `http://kprof.sourceforge.net/`, 2002.

[25] Bradley A Princen J, *Analysis/synthesis filter bank design based on time domain aliasing cancellation*, IEEE Transactions (1986).

[26] Bradley A Princen J, Johnson A, *Subband/transform coding using filter bank designs based on time domain aliasing cancellation*, Proc. of the ICASSP (1987).

[27] Georg Sander, *Visualization of Compiler Graphs*, `http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`, 2002.

[28] Inc. SPARC International, *SPARC Architecture Manual Version 8*, `http://www.sparc.org/standards/v8.pdf`, 1992.

[29] ______, *SPARC International Web Site*, `http://www.sparc.org`, 2002.

[30] Peter Symes, *Video compression demysitified*, McGraw-Hill Professional Publishing, 2000.

[31] LEOX Team, *LEOX*, `http://www.leox.org`, 2002.

[32] 4Front Technologies, *OSS Programmer's Guide v.1.1*, 2002.

[33] uClinux, *uClinux – Embedded Linux/Microcontroller Project*, `http://www.uclinux.org`, 2002.

[34] Vincent Vanhoucke, *Block Artifact Cancellation in DCT Based Image compression*, `http://www.stanford.edu/~nouk/mdct/`, 2001.

[35] XIPH, *Ogg Vorbis Web Site*, `http://www.xiph.org/ogg/vorbis/`, 2002.

Wir versichern, dass wir diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet haben.

Luis Azuara

Pattara Kiatisevi