A System-on-a-Chip for Audio Encoding

Fourth Year MEng Project Report

MEng Student:	**Jacob Bower**
Supervisor:	**Dr. Wayne Luk**
2nd Marker:	**Dr. Oskar Mencer**

Address:
Department of Computing
180 Queen's Gate
South Kensington Campus
Imperial College London
SW7 2AZ

Email: jacob.bower@alumni.doc.ic.ac.uk

Abstract

In this report I describe the design and implementation of an extensible 'System-on-a-Chip' (SoC) architecture which can be targeted at accelerating specific applications. To demonstrate its effectiveness this is used to create a completely self-contained digital audio encoder, based around a high-quality general purpose audio compression algorithm called Ogg Vorbis.

The resulting audio encoder is able to operate 33% faster than the original software only algorithm, and when running at 25MHz on FPGA hardware is able to achieve compression of 8KHz, mono, audio data in real-time.

To summarise, this project has 5 distinct achievements:

- The design of an extensible framework for creating application specific SoCs. This allows the addition of custom instructions and data processors to a general purpose CPU in a way abstracted from the underlying architecture.
- An implementation of such a framework based around an off-the-shelf soft-core microprocessor. This has specific support for fast memory access which is achieved by optimising data layout in memory banks.
- Running this system on an FPGA-prototyping board.
- Adapting a library of hardware real-number arithmetic units for reduced hardware area size and higher speed.
- An SoC implementation of the the Ogg Vorbis encoding process.

Acknowledgments

Before we begin, I would first like to express my thanks to the following people who helped make this work possible:

- Dr. Wayne Luk, for his supervision and support of this project.
- Dr. Oskar Mencer, for support and comments.
- The members of the 'Custom Computing' group who provided me with advice, particularly: Michael Rans, Altaf Gaffer, and Tero Rissa.
- Jiri Gaisler for both writing and answering my questions about the Leon processor, of which I made extensive use.
- All the people who worked hard on the many other Open Source projects I used in this project.
- My friends who provided suggestions and support. Especially Adilah Hussein, who drew Figure 2.2 and Tim Wood for letting me use his LaTeX cover sheet style.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Definition: System-on-A-Chip (SoC) - A highly integrated logic device, often containing the majority of components found in a complete computing system.

Definition: Audio Encoder - A system for capturing and representing analogue audio signals in a digital format suitable for storage and/or distribution.

1.1 Why a 'System-on-a-Chip' Audio Encoder?

"A video library in your pocket and camcorders with no moving parts" [1]. This was the opening to a recent article in 'IEEE Spectrum', heralding the demise of moving parts in portable multimedia devices. With off-the-shelf MP3 players now sporting flash memory cards capable of storing whole albums, it seems logical that the next step should be not only playback, but also the capture of digital audio with these devices.

To capture and store audio samples directly into a digital format, any time or place, is often a highly desirable goal. By keeping a digital version of captured audio it is possible to backup, distribute and replay a sample an unlimited number of times without having to worry about any kind of degradation, a problem that traditional analogue storage methods are often prone to.

Unfortunately in its raw digitally captured format, high quality audio can be rather unwieldy in terms of its data storage requirements. So to compensate for this, raw audio data are often stored in an 'encoded' form which greatly

reduces the storage requirement, possibly at the expense of losing some parts of the original signal which are redundant to human listeners.

Encoding digital audio data into a compressed format does however come at a price in the form of a high processing requirement. This comes into direct conflict with the goal of being able to capture an audio sample at any time and place, where in many cases computing power and resources may be limited.

One solution to this problem is to use a dedicated embedded audio encoding system that has been optimised for the capture and storage of digital audio. In this project I focus on the possibilities for creating a bespoke, self-contained, 'System-on-a-Chip' with dedicated hardware for accelerating the encoding process as a way to achieve this.

1.2 Project Inspiration

As I hope has been expressed above, the need to investigate embedded solutions to audio encoding itself is fairly intuitive. However the original inspiration for my undertaking of this work is a now completed project titled 'Ogg-on-a-Chip'[4]. This aimed at creating a 'System-on-a-Chip' implementation of an audio *de*coding system for the 'Ogg Vorbis' lossy audio compression format.

Ogg Vorbis is one of several existing audio encoding formats and like many of these, is capable of turning general high quality audio signals into relatively compact data streams with little discernible degradation in quality for the listener. What makes Ogg Vorbis distinct from all the other algorithms is that it has been developed with the intention of being a completely free and open standard. People or companies may feel free to use it in any application they wish with full access to the implementation and specification, without having to pay a penny in intellectual property fees.

Since the 'Ogg-on-a-Chip' project was completed, a number of other projects have sprung up looking at creating and improving embedded solutions to Ogg Vorbis decoders. It is now even possible to buy off-the-shelf portable audio players with built in support for decoding the Ogg Vorbis format [3].

Despite all the work that has been done looking at embedding the decoding side of Ogg Vorbis, almost no work appears to have been done investigating the possibilities for an embedded encoder. With this project, I hoped to try and change this.

1.3 Aims of This Project

At the outset of this project I had two primary objectives:

1. Create an extensible framework that can be used for the investigation of 'System-on-a-Chip' based implementations of audio encoders.
2. Use this framework to produce a working implementation of a digital audio encoder, by applying it to the Ogg Vorbis algorithm. This should be optimised to achieving real-time processing rates at as high quality possible using prototyping hardware available for this project.

A more detailed account of my plan for this project, can be found in Chapter 3.

1.4 Main Project Achievements

Having completed this project I believe that I can identify five major achievements which have resulted from my work. Below is a brief summary of these and references to the chapters in this document where more detailed coverage can be found.

1. **The design of an extensible framework for creating application specific SoCs** – *Chapter 3*
   This allows the easy addition of custom instructions and/or data processors to 'soft-core' CPUs in a re-usable manner. Part of this work includes an iterative approach which can be used for applying the framework to software algorithms.
2. **An implementation of such a framework** – *Chapter 4*
   This is based around an off-the-shelf 'soft-core' CPU design called Leon. The implementation utilises intended data bus and co-processor extension points in Leon, and also contains a replacement memory controller for Leon which allows faster RAM access for data processors by using an alternative layout for data in RAM banks.
3. **Running this system on an RC2000 FPGA-prototyping board** – *Chapter 5*
   Since the RC2000 does not include some facilities that Leon was intended to work with, particularly either ROM banks, RS232 ports or direct access to a PCI bus. I developed software and hardware which

together enable Leon to be booted and arbitrary programs executed using the aid of a 'host' computer.

4. **Customisation of hardware arithmetic units for real numbers** – *Chapter 6*
   As part of my design for an extensible hardware framework, I experimented with adapting some pre-made floating and fixed point hardware arithmetic libraries. My modifications to these allow an up to 28% decrease in hardware size, and a 30% increase in clock speed. These savings are mostly due to a the pipelining of a floating-point adder, and making use of hardwired FPGA multipliers.

5. **An SoC implementation of a hardware accelerated Ogg Vorbis encoder** – *Chapter 7*
   This is built using the framework developed, and is a completely self-contained SoC. It is able to encode up to 33% faster than the unmodified and unaccelerated Vorbis encoder running on the same hardware.

Ultimately, accelerating the Ogg Vorbis encoding software turned out to be extremely challenging. Nonetheless, the final design that I have produced is capable of encoding, in real-time, inputs at low sample rates using hardware available in college.

1.5 A note about RSI

It should be noted that when I began this project at the beginning of the university term in 2003, my ability to use computer input devices was severely impaired by a recent onset of 'repetitive strain injury' (RSI). In order to deal with this I deliberately unbalanced the number of courses I took between the first and second terms, such that I had more courses in the first term and spent less time on the project. As a result of this most of the major practical work done on the project took place during the Christmas break, and the second term. The first term was mostly used for background reading and research.

This approach turned out to be fairly successful, as my hands did recover enough for me to produce the work presented here. However working was still difficult at times, and there is perhaps less work done than I would have liked. Also this document was extensively written using voice recognition software, which should lead to a document with no spelling mistakes but perhaps with quite a few unexpected noise words or strange sentence constructions!

Chapter 2

Background

In this project I have investigated implementing the Ogg Vorbis lossy audio encoder using 'System-on-a-Chip' design. This covers a wide range of computing issues, both in hardware and software. Therefore before I begin with the main body of this report I will give here a brief overview of a few topics that are necessary to understand what exactly was available to me for this project, and what I am trying to achieve. In particular I will cover:

- Lossy audio encoders/decoders (CODECs) in general in Section 2.1.
- A more detailed introduction to Ogg Vorbis in Section 2.2.
- An introduction to SoC design, and the tools which are available to aid in its development, including Handel C of which I made extensive use, in Section 2.3.
- A comparison of some of pre-made 'soft-core' sequential processors available to build on, and an overview of a specific one that I chose for this project: Leon, in Section 2.4.
- The FPGA prototyping boards available for me to test my design on using real hardware in Section 2.5.
- Finally I will conclude with a review of some other works which also look at optimising Ogg Vorbis for embedded hardware systems in Section 2.6.

2.1 Lossy Audio CODECs

When audio is sampled and stored in its raw form, it typically takes up quite a lot of storage. For example a 1 minute audio clip sampled at 44.1KHz, in stereo, with a resolution of 16bits per-sample, takes up about 10Mb of space. This is generally far too unwieldy for the purposes of mass-storing audio data on digital media, or for transmission over typical communication links to the Internet for live streaming. The question that naturally arises is: 'How can we reduce the size of these data without loss of, or at least being able to control, the level of quality?'.

An 'audio CODEC' is a system for the enCOding, and DECoding of audio data for use in digital systems. Typically we are interested in using this process for the compression of audio data to alleviate the problems stated above. The term 'Lossy' refers to the fact that once audio data have gone through this process and been reconstructed, some information will be lost and the resulting signal will not be identical to that sampled. The key to a successful CODEC is being able to identify where the redundant information in the signal is and being able to remove it, while at the same time minimising the perceived impact on the listener of the reconstructed signal.

Redundancy in an audio signal arises by virtue of the way biological receptors used in the human hearing system have evolved. They work best at picking up certain types of sounds, particularly those that are useful for our survival, for example: speech. Researchers have been able to characterise which sounds humans are better at sensing and separating in so called 'Psycho-acoustic' models. Using these models, it has been possible to develop algorithms which can identify noises in a signal that a human would have difficulty perceiving, and so can be discarded.

At the core of most lossy audio CODECS, including MP3 and Ogg Vorbis, are two important features from these 'Psycho-acoustic' models that can be used to measure the importance of a component in a signal. These are:

- The absolute threshold of hearing (ATH).
- Auditory masking.

The absolute threshold of hearing makes use of the fact that humans do not perceive all frequencies of sound equally. For example, it is very difficult for humans to perceive frequencies at the limits of human auditory perception. Frequencies at these levels will require considerably more energy (i.e. need to be louder), than those that humans are good at perceiving, for example

the frequencies associated with speech. Figure 2.1 shows the threshold of energy required in the 20Hz-20KHz range for an average human to perceive a sound in quiet, measured in deci-Bells (dBs). This frequency range can be divided up into a number of non-linear 'Critical-bands'. Any frequency in an audio signal, which does not have enough energy to satisfy the 'threshold in quiet' of the band containing it, can be discarded.

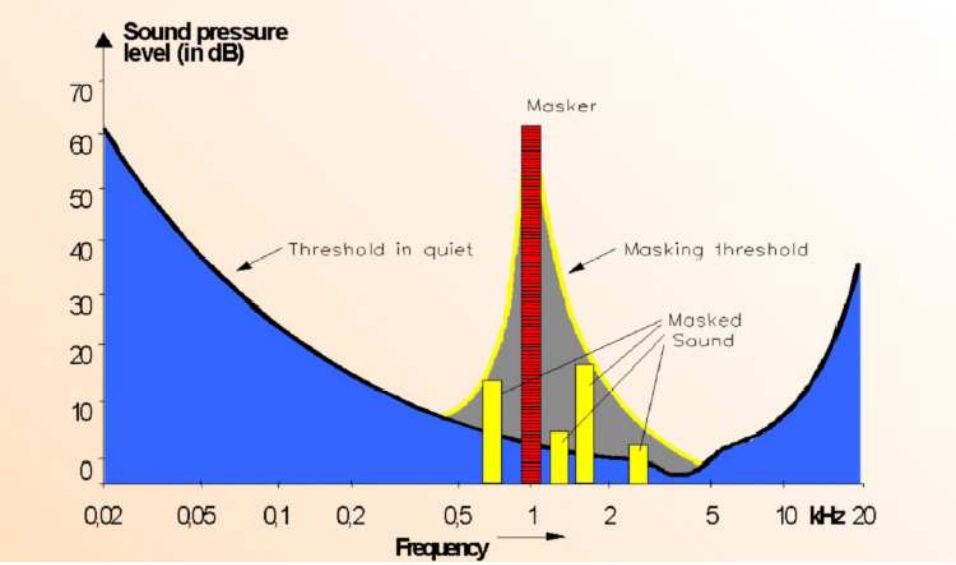

Figure 2.1: Minimum energy required to perceive a frequency, and the masking effects of a loud frequency or noise [27].

Auditory masking makes use of the fact that humans have difficulty distinguishing frequencies which are very close together. For example it is almost impossible to tell the difference between a 1,001Hz and 1,000Hz signal. So in a signal if there is a strong 1,000Hz frequency, it will easily over-shadow a weaker 1,001Hz frequency, and also many other surrounding weaker frequencies. This effect is called 'Masking' and can also be seen in Figure 2.1. If there is a frequency in a signal that has enough energy to over-shadow surrounding frequencies, information about them can be discarded. Masking can occur either as a result of single loud tones, or as a result of 'noise'.

Normally when audio data are captured digitally, the input is in the form of a Pulse Code Modulated (PCM) stream. In this the analogue audio is represented by a series of discrete values for the amplitude of the signal as sampled at fixed frequency intervals. From this we can derive amplitude versus time. This however does not allow us to easily separate out different frequency components of a signal, which is necessary to take advantage of the ATH and auditory masking effects. In order to operate using these

features, an audio CODEC must first 'transform' the PCM data from the time domain into the frequency domain.

A general technique for converting a signal to the frequency domain is the Discrete Cosine Transform (DCT), and a variant of this called the 'Modified DCT' (MDCT) is often used in perceptually-based audio CODECs [6]. This MDCT works by breaking up a PCM signal up into a series of completely overlapped windows. Each window overlaps precisely 50% of both its two neighbours. This can be seen in Figure 2.2.

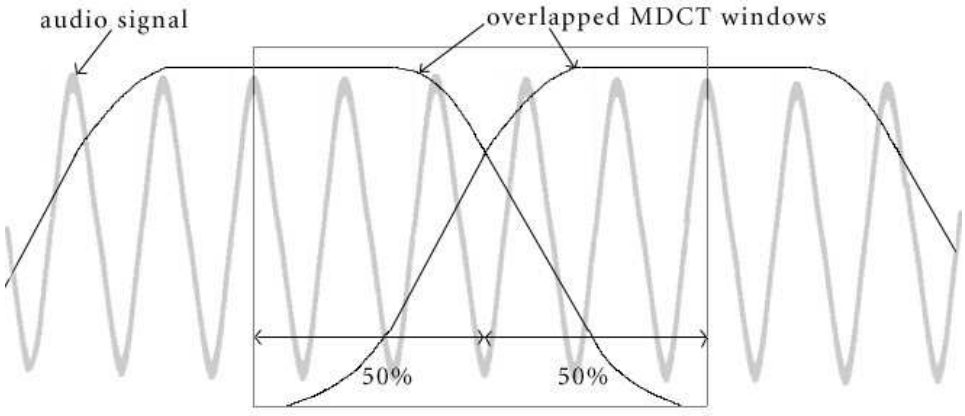

Figure 2.2: Overlapping windows in an MDCT.

Then for each window the data are converted to the frequency domain, with the amplitudes of data tapered at each end by some mathematical function as illustrated in Figure 2.2. In order to reconstruct the signal again after it has been through an MDCT, the Inverse MDCT is used (IMDCT). Each window has its signal reconstructed, and the overlapping portions of neighbouring windows are added together to produce exactly the same signal as the original. Apart from quantisation artifacts, no data are lost in this conversion process. Once PCM data have been transformed via the MDCT, it is possible to remove information using the Masking and ATH ideas.

The advantage of using this overlapped window scheme, rather than separating the signal up into completely disparate frames, is that this helps to reduce the effect of artifacts. These will arise when applying filtering to discretely transformed portions of data which are part of a continuous signal.

The MDCT transform, and the effects of ATH and Masking are 3 basic features that appear in most popular lossy audio CODECs. Some examples of these are:

- **MPEG1/2 Layer-III (MP3)** - Brought to public to awareness for its ability to transfer music over the Internet both legally and illegally.
- **AC-3** - The CODEC used for distributing audio on DVDs, and with HDTV [7].
- **ATRAC** - Sony's proprietary audio CODEC found on mini-disc players [8].
- **Ogg Vorbis** - The format discussed in the next section.

2.2 Introducing Xiph.Org and Ogg Vorbis

There seem to be a number of rather odd sounding names associated with Ogg Vorbis, so before we continue, the meanings of these need to be clarified. The best way to begin is probably a quick overview of the Xiph.Org Foundation. Their own explanation from their homepage [2] is:

> *"[The] Xiph.Org Foundation is a non-profit corporation dedicated to protecting the foundations of Internet multimedia from control by private interests. Our purpose is to support and develop free, open protocols and software to serve the public, developer and business markets."*

In order to achieve this goal, the Xiph.Org foundation host and promote a number of Open Source projects primarily concerned with the encoding and distribution of multimedia content. These include a number of video and audio CODECs, and a general multimedia container format called Ogg. All of these projects are either completely free of patented algorithms, or in the case of Ogg Theora (a general purpose video CODEC): "the Xiph.org Foundation has negotiated an irrevocable free license to the vp3 CODEC for any purpose imaginable on behalf of the public" [9].

'Vorbis' is the Xiph.Org Foundation's general purpose lossy audio CODEC, designed to be embedded in Ogg streams. The quality and compression ratios achieved by the algorithm are intended to be comparable to or better quality than most other modern lossy audio CODEC algorithms including: MP3, AAC, WMA and TwinVQ. The source code to a reference implementation of the CODEC is freely distributed on the Vorbis website [2], under the unrestrictive BSD License [10], making it suitable for commercial developers to include in their products without worry of legal problems. One

of the biggest users of the technology in an embedded capacity so far appears to be the video games industry, with a number of popular games using Vorbis to encode audio [11].

Like many multimedia standards, Vorbis is defined mainly by its data format and structure, and not by the algorithms for performing the encoding. The only rule when creating a Vorbis compliant encoder is that the stream produced must conform to the clear formats specified in the documentation [12]. Any encoder producing a stream that can be decoded by the reference decoder is considered correct.

The Vorbis CODEC itself has been borne mostly as a research project, for investigating various modern techniques for lossy compression of audio data. This is reflected in the design of the data format for a Vorbis stream, which has many points at which different modes can be selected, with flexible requirements on data available for each packet in a stream. This is also represented by the fact that unlike many other audio CODECs, all tables necessary to decode a stream are included in the stream itself. This is in contrast to algorithms like MP3 which have pre-defined tables which are the same for every stream.

The net result of all this, is that although the format and decoding steps are very clearly documented, the algorithms used in encoding are not. A comprehensive list of all the information on the encoding algorithms is as follows:

- The reference encoder source code.
- Hints in the format specification.
- A very out of date and largely incomplete document entitled 'Vorbis Illuminated' [14].
- Any discussions found in the logs of the Vorbis developers mailing list on *Vorbis-dev@xiph.org*.
- A brief overview in the introductions from the Ogg-on-a-Chip decoder project [4], and the paper 'Ogg/Vorbis in embedded systems' [30].

Fortunately in practice there are only a limited number of different algorithms/modes/datasets employed in the current stable releases of the Vorbis reference CODEC libraries. Further to this, there appears to only be one Vorbis encoder in existence, and this is the reference one. There are however a number of decoders, including one written entirely in Java [13]

A rough outline of the steps for encoding used in the current reference implementation of the encoder is as follows:

1. Divide input signal up into windows (which are possibly of non-equal size).
2. Perform an MDCT on these windows.
3. Apply masking, and ATH cut-offs.
4. The remaining frequency spectrum, is separated into a 'floor' which represents its overall structure and a 'residue' which represents fine details.
5. These are then separately encoded using a variable set of techniques and finally Huffman encoded.
6. The codebooks necessary for decode are included in the stream.

More detailed accounts of this process can be found in the various sources described above, but are not necessary to understand the rest of this project.

2.3 System-on-a-Chip Design

Ever since the first digital computer, the Manchester 'Mark I' [15], was successfully operated in 1948, the drive in the industry has been to build computers smaller, faster and less power-consuming. The first major development in this field was the invention of the transistor and simple integrated circuits (ICs) which replaced the high-power and unreliable valves used in the original computers. Computers were then constructed with dozens or hundreds of these ICs, rather than tens of thousands of valves. This was followed by the invention of the first general purpose microprocessor: the Intel 4004 in 1971 [16], which combined the functions of many of these separate ICs into a single Chip.

System-on-a-chip (SoC) design is the natural progression in this line of evolution, where now we combine the functionality of many high-density ICs like microprocessors and bus controllers again onto just a single chip. Figure 2.3 shows some of the elements we might expect to find in a typical SoC design.

The ramifications of this design, are our ability to build complete processing solutions in a tiny amount of space, with low power consumption due to

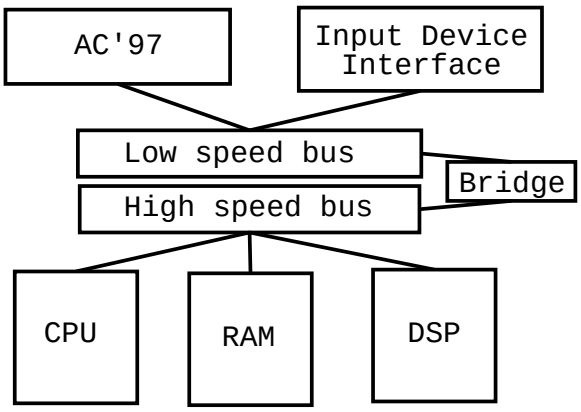

Figure 2.3: A SoC

the small size of the componentry. These advantages are gained mostly in embedded applications.

To take an example: my desktop computer at home contains high-performance RAM, CPU, graphics processor, bus-controller and various exciting little gadgets. With all this it can, among other things, play Ogg Vorbis encoded music. However it does also weigh several kilograms, and requires at least a 110 volt power source. This is of course a problem if I want to listen to my Ogg files while I walk to college in the morning. One solution to this might be to stick my computer in a trolley along with a car battery, and wheel it in with me everyday. This however is a little impractical. A far better solution is to buy a small portable personal audio device like the much talked about Apple 'iPod' [17].

So how is the iPod able to do one of the tasks I normally need a whole desktop PC to perform, in such a tiny amount of space and with such low power consumption? The answer is simple: it has an SoC processor which is specially designed for the purposes of decoding compressed audio. The specific chip they use is made by a company called PortalPlayer and technical details can be found on their website [18]. According to these details the chip itself is actually a fairly generic system, made up of two ARM CPUs, and a number of integrated peripherals for controlling LCD displays and mass-storage devices etc.

Another feature of SoCs, which does not appear to be exploited in this situation, is that not only can you integrate a whole generic system into just a single chip, you can also easily add custom logic. This could allow you to perform a heavy processing task which might normally require a high-performance general purpose processor, using a low-performance processor

with some additional hardware to accelerate specific parts of the application.

Clearly creating a digital circuit design for a complete 'System-on-a-chip' with dozens of different components and potentially millions of logic gates by hand would be rather impractical. Fortunately languages have been developed that make the design of integrated digital circuits in general more abstract and easier to work with. Two of these are VHDL and Handel C.

VHDL has been around since 1981, and in 1987 it became an IEEE standard (no.1076). Since then several revisions have been made. There were several reasons for the original invention of VHDL, the primary ones being that hardware manufacturers wanted a precise and standardised way of describing their digital logic designs. Such a standard allowed computer software to be developed to aid in the complex task of developing, simulating and testing of these designs. Also being a standard allowed the distribution of a digital circuit design to third parties with the reassurance that the design will be correctly interpreted. Some institutes including the US Department of Defence will now not accept a digital component that has not had its design clearly specified in VHDL so that its correctness can be verified.

VHDL itself allows the specification of a digital circuit at many levels of abstraction. At its lowest level it can specify electronic signal propagation delays in individual primitive logic components, and the wirings between them. In its most abstract form, it can describe a digital circuit at the behavioural level. At this level a designer need not consider how the digital circuit is actually formed but instead concentrate on the algorithm that it implements. From a behavioural description, a low-level circuit can be generated using clever compilers.

Handel C is another language for digital circuit description. It is a closed and proprietary standard created by Celoxica and shipped as part of their 'DK Design Suite' [19]. The Handel C language takes the abstraction level where VHDL leaves off one step further, and allows you to design digital circuits using a variant of the ANSI C standard.

This approach has three main advantages:

- It greatly simplifies the task of taking a software program written in the C language, and implementing it directly in hardware for performance benefits.
- It simplifies sharing common constants or even code that are required by both hardware and software parts of an implementation.
- It allows new hardware algorithms to be developed in a way that is

generally far clearer for experienced software developers than can be achieved using VHDL.

In this project, I have made extensive use of Handle C for the hardware portions of my implementation, taking advantage of all of these points.

2.4 Pre-made 'soft-core' CPU designs

One possible approach for this project could have been to create an entirely new general purpose sequential processor from scratch. The advantage of this route would be full control over every aspect of the design, potentially allowing for maximal efficiency. The disadvantage would have been having to do a considerable amount of work, most of which may be unnecessary. Particularly one must consider that designing a CPU from scratch will require the development of a full complement of compilers and base system libraries to go with it.

In lieu of the extra work that would be required to implement my own CPU from scratch, I opted to chose one of several pre-made 'soft-core' processor designs and develop an extension framework based around that as necessary. The term 'soft-core' refers to the fact that we are talking about an abstract hardware design which can be altered before being realised, rather than a pre-made hard-wired chip. The main CPUs that I considered for this project were:

- Nios - Developed by Altera for use on their range of FPGAs and ASICs.
- MicroBlaze - Developed by Xilinx for use on their range of FPGAs and ASICs.
- The Xilinx Virtex II Pro FPGA, which includes an embedded Power PC core.
- OpenRISC - A free and open source soft-core CPU.
- Leon - Another free and open source soft-core CPU, that implements a complete SPARC v8 compliant ISA. It also has an optional high speed floating point unit called GRFPU, which is free for download but is not open source and is only for evaluation/research purposes.

There were many distinguishing features amongst all of these, but in essence (apart from the PPC core in the Virtex-II Pro) they all sport a 32bit RISC

	Nios	MicroBlaze	Virtex II Pro	OpenRISC	Leon
Open Source	No	No	No	Yes	Yes
Hardware FPU	No	No	No	No	Yes
Bus standard	Avalon	CoreConnet	CoreConnect	WISHBONE	AMBA
Integer division unit	No	No	Yes	No	Yes
Custom co-proc/instr.	Yes	No	No	Yes	Yes
Dhyrstone 2.1 MIPS/MHz	0.2	0.68	1.5	1.0	0.85
Max freq. on FPGA (MHz)	123	150	400	47	69
Max MIPS on FPGA	24.6	102	600	47	58.7

Table 2.1: Some of the major distinguishing points of the pre-made SoCs

architecture with single issue 5 stage pipelines, have configurable data/instruction caches, and have support for the GCC compiler tool chain. Thus making them all reasonable candidates for executing the Vorbis Algorithm. They also all feature bus architectures suitable for adding extra processing units as slaves or masters that could be used to accelerate the algorithm, although some go further and allow the addition of custom instructions/co-processors.

The Ogg Vorbis encoding algorithm is itself heavily arithmetic based. For this reason, in order to develop an efficient embedded solution, I needed a good floating point unit, or at least a good integer unit in the case where I might develop a fixed point version.

In Table 2.1 is a brief comparison of some of the distinguishing features of these different SoCs. It should be noted that the measurements for MIPS/MHz were mostly taken from marketing blurb, and likely to fluctuate wildly depending on the configuration of the CPUs, as they are all highly configurable. Much of this information was gathered from the paper 'Overview of Embedded Processors used in Programmable SOC' [22]

After consideration, I decided to base the framework created in this project around the Leon CPU/SoC. There were three main reasons for this choice. Firstly it has hardware floating point support available. The entire Vorbis algorithm is based around floating point arithmetic, and I also eventually concluded that converting the entire program to fixed point would be too time consuming and error prone. Secondly Leon is completely open source and not tied in to any particular FPGA/ASIC vendor, fitting in nicely with the Xiph.Org and Ogg Vorbis ideals. Finally Leon seemed to be quite a mature and feature packed processor with a huge number of configurable options.

2.5 FPGA Hardware available

An important step to developing a successful digital system, is to validate that it actually works in real life conditions. Previously this has mostly been achieved through intensive simulation using computer software, due to the prohibitive cost and time associated with making a single Application Specific Integrate Circuit (ASIC) for a one-off test. However recently it has become possible to test designs in minutes on real life hardware. This is made possible by field programmable gate array (FPGA) technology.

An FPGA is a logic IC for which the structure of the gates and memories are re-programmable such that practically any digital circuit can be implemented. There are many different ways of physically implementing an FPGA and more details can be found on major manufacturers' websites such as Xilinx and Altera. Fundamentally though, they are all programmable using languages like VHDL and Handel C.

Also available are prototyping boards for SoCs. These usually centre around an FPGA with a number of pre-made connections to real life inputs including: RAM, analogue/digital data I/O, and a system for easily uploading designs to the FPGA(s). These can further be used to support the rapid development and testing of SoCs.

For this project there were two candidates for testing my digital design in hardware available as college resources. These were the RC200 [20], and the RC2000 [21] (Figure 2.4).

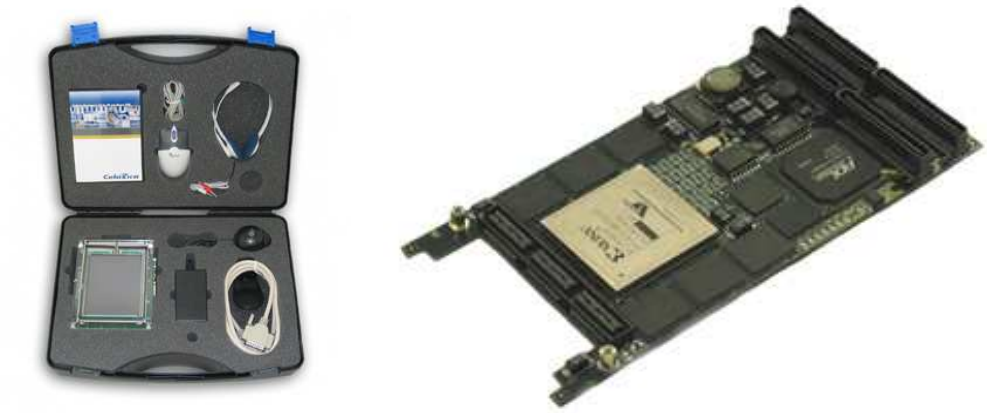

Figure 2.4: RC200(left), RC2000(right)

The RC200 is a stand-alone FPGA prototyping unit, based around a Xilinx XC2V1000 Virtex-II FPGA. The board itself contains a number of peripher-

als connected to the FPGA for the input and output of multimedia signals, and several data interfaces including Ethernet and RS232 serial ports. It also has about 8Mb of high-speed SRAM (ZBT, 2-cycle read/write). Hardware designs are uploaded via a parallel cable connected to a PC or stored on a flash card.

The RC2000 is a PCI FPGA prototyping unit, based around a Xilinx XC2V6000 Virtex-II FPGA. It has no external input or output connections beyond the PCI bus, and instead relies on all data including hardware designs to be uploaded and downloaded via a host PC. It has 12 or 24Mb of SSRAM (ZBT, 3-cycle read/write), and 128Mb of DDR SDRAM. The host PC can communicate directly with the FPGA across a 4Mb PCI address range.

In an ideal world it would have been possible to perform all development on the RC200, using its audio input and data output connections. Unfortunately this turned out be impractical for two reasons:

1. The FPGA on the RC200 did not have enough resources to accommodate Leon and the GRFPU (FPU).
2. Programs which execute on Leon had to be uploaded via the RS232 connections, which proved to be far too slow as the compiled Ogg Vorbis encoder was several megabytes due to embedded tables.

Instead the RC200 was used for some testing of my Leon Handel C framework only, while the RC2000s were away being repaired.

The FPGAs available on both boards are almost identical in the types of resources that they provided, the only difference being their capacity. As well as reconfigurable logic they also provided hard-wired RAM resources, and hard-wired 18x18 signed integer multipliers, which were both of use in this project.

2.6 Similar work in this field

During this project, the two main pieces of relevant work I have come across have been the 'Ogg-on-a-chip' project [4], and the paper 'Ogg/Vorbis in embedded systems' [30].

In the paper 'Ogg/Vorbis in embedded systems', the authors explore some of the issues surrounding running the Ogg Vorbis decoder in highly resource constrained environments with DSPs. Their work mainly focused on two

points. Firstly reducing the memory overheads required by the Vorbis decoder, which are relatively large when compared with other CODECS like MP3 and AAC. Secondly, replacing the IMDCT implementation used in the reference decoder with an FFT, which was more conducive to the DSP environment and produced a considerable speed up. This was due to the fact the IMDCT consumes most of the run time of in the Vorbis decoding process.

Unfortunately this paper was not published until late into the duration of my project, and I was not aware of it until even later. As such it has not been a tremendous influence on my work. However it is definitely worth noting not least of all because it has one of the clearest descriptions of the encoding/decoding process used in Vorbis that I have found.

Of much more influence to me was the 'Ogg-on-a-chip' project, which was the master thesis of two students from the University of Stuttgart in 2002. In this work the authors were looking at using SoC design to implement a completely self-contained Ogg Vorbis decoder.

For this they also used the Leon processor, and were able to synthesise their design on to an XESS XSV-800 FPGA based prototyping board. The final version of their hardware implementation fell approximately 16% short of producing real-time output when encoded data was at a sampling rate of 44KHz.

On the hardware side of their design, they focused on optimising the already existing IMDCT used by the decoder. They investigated several different partitions of software and hardware for this, and eventually decided upon what they describe as a 'Mini-MDCT', which implements about half of the transform.

Also as a result of their work they produced an early integerised version of the Vorbis decoder, which as I have already mentioned, is heavily floating point based. This was after concluding that the floating point units available at the time that worked with Leon were insufficient. However since then the high-speed GRFPU has been released, which I have chosen to use in my project. Also since their work, there has now become an official integerised version of the decoder library made by a different group. There is however, no sign of an intergisation of the encoder.

Chapter 3

Design Methodology

The development of a hardware accelerated, SoC based, audio encoder will involve a number of intricate and complex tasks including the design of software, hardware and system architecture. Although each of these can be studied in an independent manner, success will only be made possible by a co-ordinated approach which allows these individual components to function together in a coherent system.

In this chapter, I will outline the methodology I planned to use to achieve this within the time frame available for this project. Specifically I will cover:

- A general overview of the issues involved in audio encoding, and how a SoC design might be helpful. Section 3.1.
- Creating an extensible SoC framework. This will be based around a general purpose sequential processor with support to allow the easy and rapid development of hardware data processors and custom processor instructions. Section 3.2.
- An iterative approach to applying this framework to gain an effective acceleration solution. Section 3.3.

3.1 Issues in Audio Encoding

In the 'Background' chapter of this report, I gave a brief introduction to the processes involved in lossy audio encoding with the specific example of Ogg Vorbis. From this I have extracted three general comments that can be made about the encoding of audio:

1. There are usually several fairly independent transformation/analysis steps involved.
2. Each step typically seems to fall into one of two categories. The first are heavily arithmetic based with adjustable levels of precision and often with high levels of parallelism, for example computing the energy spectrum of a sample using a MDCT. The others are highly precise and sequential in nature, for example correctly packing data into a stream at the bit level.
3. The audio encoding process is generally considerably more processor intensive than the decoding process. This a common feature in the design of multimedia algorithms, the idea being that encoding need only be done once so one should put quite a bit of effort in to make decoding, which may be performed many times, easier and thus cheaper.

Considering these points, it would seem that for a portable/embedded audio encoding solution, a System-on-a-Chip design would be appropriate. Using this approach we can use a general purpose and low-power sequential processor for dealing with fiddley tasks with low levels of parallelism, for example packing bits into a stream. We can then add custom hardware to improve the potentially highly parallel arithmetic processing parts. Having these separate parts grouped together in a single chip, will hopefully make for a small and low power solution.

3.2 An Extensible Framework to Allow Hardware Support

Having now roughly characterised audio encoding algorithms and identified that a System-on-a-Chip design may be beneficial, I will now suggest a systematic approach to taking advantage of this. Critical to this will be the creation of a generalised hardware framework, which allows rapid and easy exploration of software/hardware partitions.

The idea of augmenting the software algorithm with hardware support is certainly not new. So before creating or such a framework, it seems sensible to consider previous approaches to this problem.

Broadly speaking attempts at accelerating a sequential processor based SoC design fall into three categories:

1. Customising the processor directly but in a general way, for example increasing the number of arithmetic units, or varying the size and

width of available registers. An example of customising a processor in this way can be found in [33], where the authors explore adjusting the number of ALU's present in a processor, which can then be taken advantage of by using a suitably customisable compiler.

2. Customising the processor in a highly application specific way, for example adding specific macro-instructions which perform common operations faster than can be done sequentially. An example of adding custom instructions can be found in [34] where the author investigates adding custom macro instructions to a processor which are generated from a software algorithm automatically by a compiler.

3. Adding external peripheral data processors which operate on data independently of a main CPU. Examples following this model to accelerate a specific application include the Ogg-on-a-Chip project discussed in Section 2.6.

The former two of these are currently in an area still subject to a great deal of research, some of which is being carried out here in the Imperial College, Custom Computing group. The later approach of adding an external peripheral processor is a far more established technique and the traditional solution to these kinds of problems. Typically a soft-core processor will have a data bus which allows the addition of a processor which functions independently of the main CPU, but shares access to a common memory pool. It is of course also possible to use a hybrid combination of all these approaches.

When considering the most effective way to quickly build a framework that could be used to accelerate audio encoding, I decided to focus my efforts on adding custom macro instructions, and data processing units. The primary reason behind limiting myself to these two categories of processor extension, was that not much work appears to have been done on creating highly customisable general processors that are capable of working with real numbers. This would be quite a hindrance when we consider that most audio encoding algorithms are based on floating point arithmetic, particularly the Ogg Vorbis encoder which is to be my case-study. I also considered that producing a processor of this nature would be outside of the scope of this project.

To build my own framework, I decided on a minimal list of requirements:

1. The architecture should be as independent of the base sequential processor used. It would be preferable if the framework could completely insulate the extension units from the underlying base SoC/CPU.

2. Adding or removing custom instructions or data processors should be simple, i.e. require few steps with limited distributed alterations.

3. Custom data processors should have a simple and fast interface for accessing shared system RAM.

4. Custom instruction interfaces should be as flexible as possible, with variable input and output parameter numbers and sizes.

5. There should be support for real number processing in hardware, and this should be modular in nature to allow exploration of different real number representations.

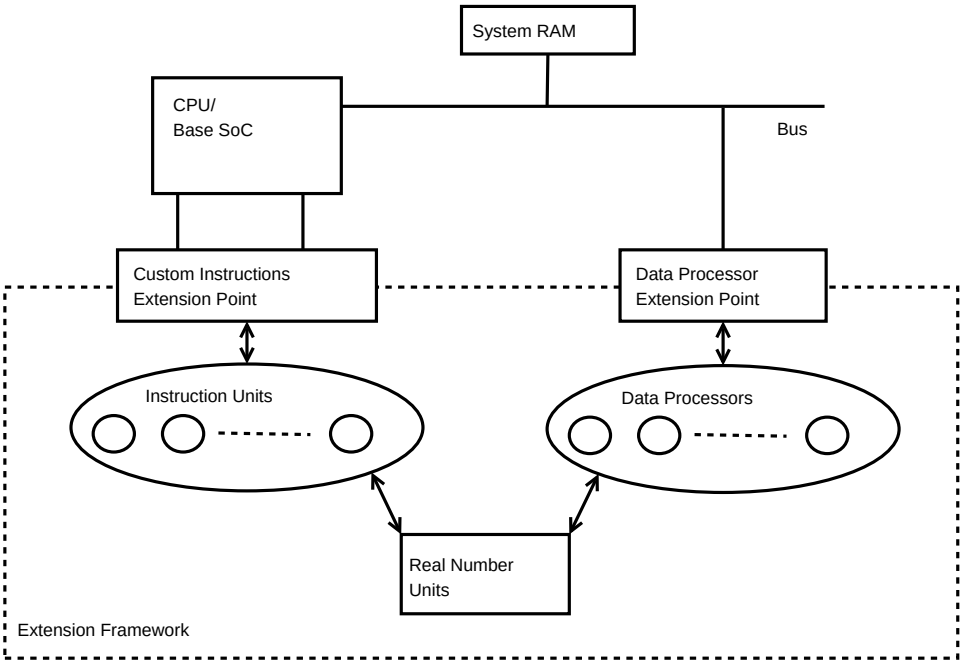

Figure 3.1: Target extensible framework architecture.

The ultimate goal is to produce a system as illustrated in Figure 3.1. In this design, the 'Extension Points' represent an interface abstracted from the underlying base SoC/CPU to which multiple custom data processor or instruction units can be connected. For data processors this will represent a connection or translation layer to connect them to an addressable data-bus which allows access to at least shared system RAM. For custom instructions, this will provide a mechanism for the CPU to: upload parameters, download results, activate an instruction unit, and be notified when execution of an instruction is completed.

In this project, a realisation of these requirements is spread between Chapters 4 and 6. In Chapter 4, I explain how I implemented a hardware design

that attempted to meet requirements 1-4 by building an extension framework for the Leon 'soft-core' processor. This design took advantage of the AMBA bus present in Leon for implementing an interface for data processors, and the Leon co-processor interface for adding custom instructions.

In Chapter 6, I describe the creation of a modular interface for different real number implementations. Along with this I present the results of an exploration of different pre-made libraries for dealing with real numbers in hardware, and how I attempted to adapt and optimise them. The resulting real number back-ends were then added to my Leon based framework to fulfill requirement 5.

An evaluation of how well these requirements were met is presented in Section 9.1 of the Evaluation chapter of this report.

3.3 Applying the Framework To Software

Once a framework for quickly developing hardware is created, it will be possible to begin investigating applying it to the acceleration of audio encoding algorithms. To do this, I will start with a software only version of an algorithm, for example a reference implementation. Then, roughly speaking, the steps to attempting to achieve a speed up that need to be followed will be:

1. Get the SoC + framework running on an FPGA based prototyping board, and see how fast a software only version of the algorithm runs.
2. Identify possible components of the algorithm for acceleration using software profiling tools.
3. For those that do not look like they would benefit from hardware acceleration, attempt to increase the speed of the software versions by using traditional software optimisation techniques. For example pre-compute tables for a specific target input rates, re-arrange array access to increase cache performance, or remove phases of the encoding scheme which do not adversely affect the output quality.
4. Once all the obvious software optimisations are complete, determine how components can be optimised by using the framework created to add acceleration logic such as custom instructions, or dedicated data processors.
5. Repeat from step 2, until the desired encoding speed can be reached.

In Chapter 7, I apply this method to the Ogg Vorbis encoder algorithm using the framework developed.

3.4 Summary

In this chapter I began by describing some of the issues involved in audio encoding algorithms. Based on these, I suggested that an SoC design could take advantage of the general structure of these algorithms to overcome the resource constraints in an embedded processing environment. I then drew up a list of requirements for an extensible hardware framework, and a complimentary optimisation strategy for software which could be used to implement a complete SoC audio encoder.

Chapter 4

An Extensible Handel C + Leon Framework

In this chapter I will present a realisation of the framework outlined in my methodology for this project. This is based around the Leon soft-core processor which was briefly introduced in Section 2.4 of the Background Chapter. The framework itself is written in the Handel C language, and so one of the main focuses here is the division between this and the VHDL language in which Leon is written.

By its open source nature, Leon can of course be extended in any way I had wished by just modifying the source code to support my ideas. But to simplify the process, Leon already provided two major points at which custom hardware can be added cleanly:

- An implementation of the AMBA Bus standard [26], which allows simple memory mapped access to peripherals attached.
- An interface that allows custom co-processor units. These are accessed by generic pre-defined SPARC machine code instructions, and can run in parallel with the main Leon 'Integer Unit' (which provides the core functionality of Leon SPARC CPU).

Discussion in this chapter will be broken down into the following categories.

- The two main extension points to Leon: the AMBA bus, and the custom co-processor interface, and how I made these accessible in an extensible way from a Handel C design. These are described in Sections 4.1 and 4.2 respectively.

- A new memory controller I developed for Leon, which allowed faster and more efficient access to RAM from my Handel C hardware designs in Section 4.3.
- Then in Section 4.4 I will show how these components fit together to produce a simple extension framework for Leon written in Handel C, similar to that proposed earlier.

4.1 Utilising the AMBA bus

The Advanced Microcontroler Bus Architecture (AMBA), is a freely available bus standard provided by ARM [26], and implemented in Leon. The standard specifies three types of bus including two backbone buses. Leon implements the Advanced High-speed Bus (AHB) backbone and the Advanced Peripheral Bus (APB) parts of the standard.

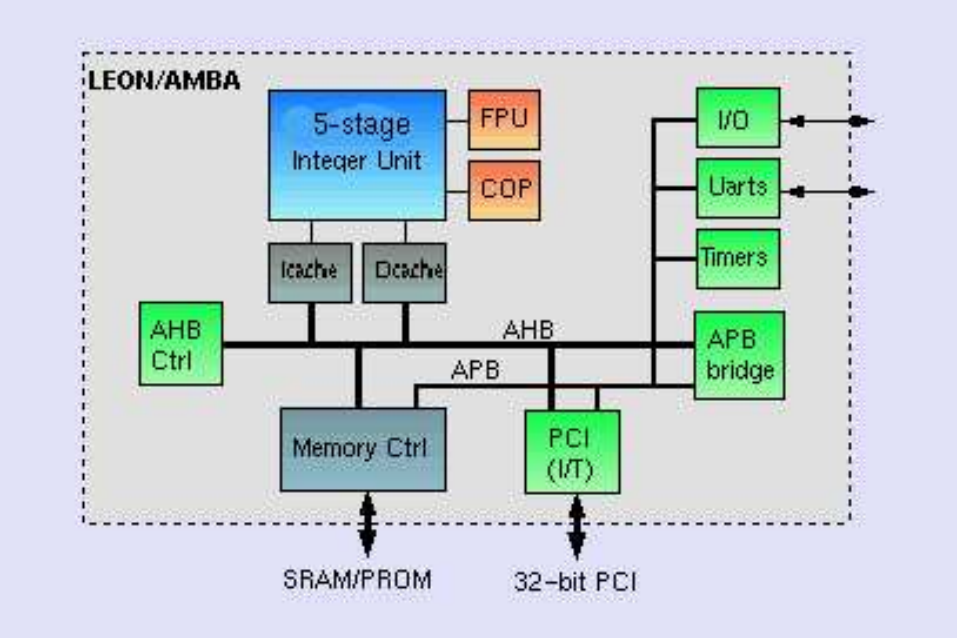

Figure 4.1: Block diagram of the Leon SoC, illustrating extension points

The AHB allows high-speed, pipelined, burst transfers between masters and slaves. The APB provides a simpler and low power/speed interface for peripherals. All devices on the APB are Slaves, and are accessed via a bridge from the AHB, as illustrated in Figure 4.1.

Typically in order to implement a new peripheral/data processor that requires direct access to system memory, a design will have a connection to the APB for a simple interface to expose control registers, and a connection to the AHB as a bus master to provide fast DMA transfers. Two classic examples of this design are the memory controller in Figure 4.1, and the 'Mini-MDCT' data processor produced as part of the 'Ogg-on-a-chip' project.

For my framework I began by following a similar design pattern, with an emphasis on making it easy to add multiple new data processing units. By easily I mean only having to modify my Handel C design and not any VHDL code. In order to achieve this, I implemented the Handel C part of my design as a combined single AHB master and single APB slave, as illustrated in Figure 4.2.

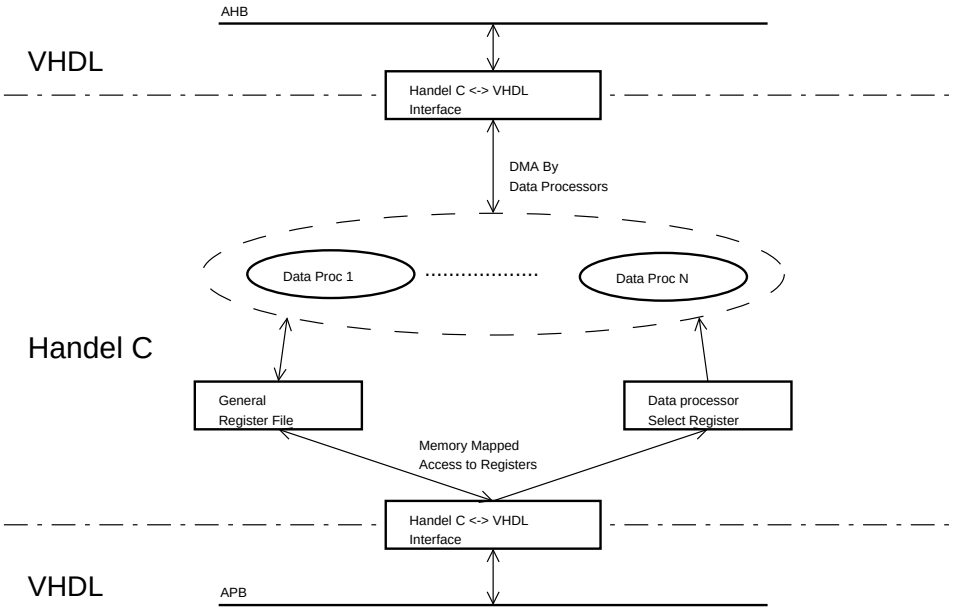

Figure 4.2: My design for adding data processing units which require memory access to Leon using Handel C

In this design an arbitrary number of data processors can be added to the Handel C part of the implementation. In order to activate one of them, code executing on the Leon CPU first initialises a data processor by filling in the registers which are shared between all the data processors and exposed by the APB using memory mapped I/O. It then writes into a special register a value which selects which data processor is to be activated. When a data processor is activated it can access memory directly using the Handel C connection to the AHB, and when complete can update the register file

exposed by the APB to indicate its completion status. The disadvantage of this approach is that only one data processor can be active at a time, as they all share the same access to the buses. Fortunately this did not turn out to be a problem for any way in which I used the framework in this project.

In order add a new data processor in my implementation, all that needs to be done is to add an entry to an array. A snippet of code in Table 4.1 shows this array, which is indexed using the lower `PROC_FUNC_BITS` bits of the value written into the data processor selection register exposed on the APB.

```
void DataProc1(void) { ... }

void DataProc2(void) { ... }

typedef void (*PROCESSING_FUNC)(void);
void NullProcFunc(void){}
#define PROC_FUNC_COUNT  4 // should be a power of 2
#define PROC_FUNC_BITS width(PROC_FUNC_COUNT)
PROCESSING_FUNC processing_func[PROC_FUNC_COUNT] = {
        DataProc1,
        DataProc2,
        NullProcFunc,
        NullProcFunc
};
```

Table 4.1: Snippet of Handel C code illustrating adding a new functional unit

4.2 Adding a Custom Co-Processor to Leon

As well as providing the traditional extension point of a memory mapped data bus for adding new peripherals, Leon also provides an interface for adding a co-processor that can be controlled by machine code instructions.

Normally this interface is used for adding a floating point unit, and indeed in my project is also used for this purpose. However Leon allows two co-processors to be added, so the second slot was free to be used for any purpose. So I attempted to create a more abstract interface to this for use in my extensible framework. Figure 4.1 shows the relationship between co-processors (COP/FPU) and the Leon 'Integer Unit' which covers the main program execution pipeline.

In the Leon Architecture co-processors are effectively divided into two parts. The first part is an 'Execution Unit' which runs in parallel with the main 'Integer Unit'. This has a pipeline similar to the main 'Integer Unit', handles instruction decode and provides a register file dedicated to the co-processor. Once an instruction has been decoded and any data dependencies resolved, control flow is then passed on to the second part which is a custom co-processor core which actually executes the operation. This core receives two 64-bit operands, and returns a 64bit result plus any completion/exception flags. Provided the core only takes only a single cycle to execute an instruction and there are no data dependencies, the co-processor can be issued one instructions per cycle.

An implementation of the execution unit part of the co-processor was provided with Leon [1], so in order to make use of a custom co-processor in my framework all I had to implement was the co-processor core. To do this in a way that was accessible from my Handel C design, I divided the core into two halves implemented in VHDL and Handel C respectively. The VHDL part handled loading the operands (which are delivered over several cycles), and the Handel C contained units to perform the operations. This mechanism is illustrated in Figure 4.3.

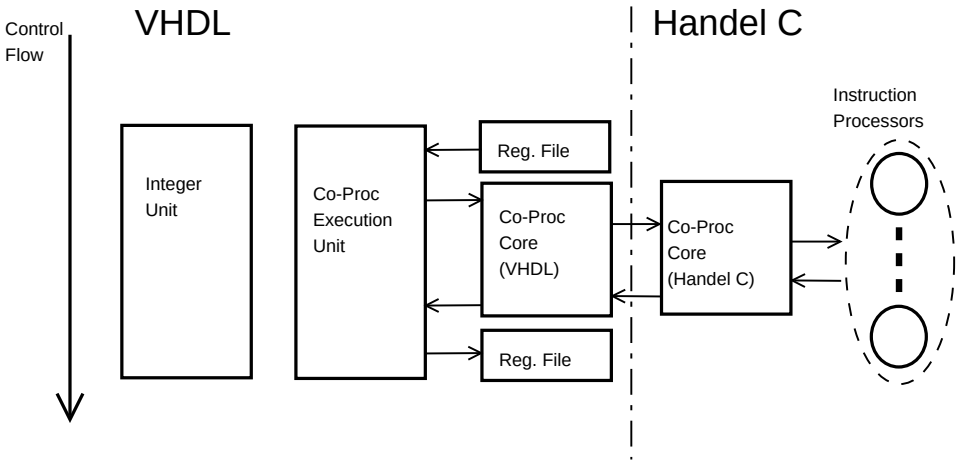

Figure 4.3: A Leon co-processor implemented with Handel C

Using this design, it became trivial to implement any new custom instructions. Using the same technique of an array of pointers to functions as used for data processors, adding a new instruction simply involves creating a new Handel C function with the prototype:

[1] Actually to be more precise, a floating point only execution unit was provided. I had to modify this slightly to use the generic SPARC co-processor instructions.

```
word64 Instr1(word64 param1, word 64 param2)
```

4.3 Replacing the Leon Memory Controller

As we shall see in Chapter 5 in order to get Leon to run on the RC2000 prototyping board, it was necessary to share the available RAM banks with other parts of the hardware design. So to support this with Leon, it would be necessary to delay memory accesses until other devices had finished using the RAM banks.

In my implementation I achieved this by replacing the default Leon memory controller with a new one. This was partly written in Handel C, and it was in the Handel C part in which the arbitration of access to the RAM was performed. It turned out that although this design was initially developed through necessity, it actually provided a number of advantages.

By placing part of my Handel C design in-front of the memory access performed by Leon and providing a locking mechanism, this introduced two side effects:

1. Once a data processor was activated it could acquire a lock on the memory, and then access RAM directly using the same Handel C functions as used in the new memory controller. This immediately eliminated the overhead involved in setting up an AHB transaction.
2. Because my design now had full control of the organisation of data between the multiple RAM banks attached to the FPGA/SoC, I was able to alter the layout of data in memory.

Combining these two side effects, I found there were rather large speed benefits for the data processing portions of my design.

In the original Leon memory controller, when consecutive words were written to memory they all went to the same memory bank until a boundary was crossed and then they were written into the next bank. In other words, RAM banks were selected using the *upper* bits of the memory address being accessed.

However I found by selecting RAM banks using the *lower* bits of the memory address being accessed, it was possible to 'stripe' consecutive words across the RAM banks. Now, as my Handel C data processors had direct access to the individually accessible RAM banks using the same functions used in the

memory controller, they were able to read and write consecutive memory locations, as seen by the CPU, in a parallel manner. This is illustrated in Figure 4.4.

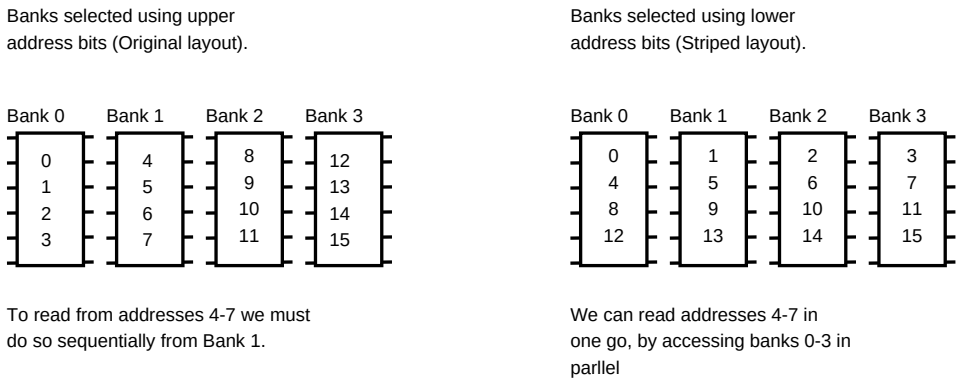

Figure 4.4: Old vs. New RAM layout

So for example, in a system with four 32bit RAM banks which each allow a complete read in two cycles, a data processor needing to access data from a linear array could do so receiving 128bits of data every two cycles. If we were instead to do this as a burst transfer on the AHB using the original Leon memory controller, we would first have to set-up the transfer (which can optimally be done in two cycles: request then grant), and data could then only be delivered in 32 bit words every 2 cycles. Thus giving us a $(128 * 2)/(32 * 2) = 4$ times speed increase for long transfers, and even a $(2setup + 2transfer)/2transfer = 2$ times speed increase for single word accesses using my modified memory layout.

4.3.1 Memory Controller Handel C/VHDL Division

The new memory controller I developed is basically a complete drop-in replacement for the original Leon memory controller, which itself was simply an AHB slave device. It should also be noted though that the original memory controller also had a number of registers exposed on the APB bus. These were used to interrogate and alter the configuration of the memory controller. However I did not replicate this functionality.

As we shall see in the next section, it would be highly inconvenient not to implement at least some of the new memory controller in Handel C, for easy access to the locking system. Also on the RC2000 and RC200 boards, pre-written libraries were supplied by Celoxica making RAM access in Handel

C very simple. So in my implementation, all the actual accesses to RAM and locking are implemented in Handel C.

There is also a very small VHDL part of the device, which provides a clean interface to the Handel C part, abstracted from the AHB connection. The decision to implement any part of the device in VHDL is fairly arbitrary, but it just happened that expressing setting up the AHB transfer from VHDL turned out to be quite neat and concise. The Handel C part receives a memory address, a read/write signal, a transfer size (half-word, word, double word) and 32 bits of data (during a write), and provides 32 bits of data on a read. There is also a signal for the VHDL part to activate the Handel C part, and a signal for the Handel C part to signal when an operation is completed.

In my design, access to RAM banks from Handel C by both the memory controller and other components use a common set of access units. There are as many units as there are RAM banks, and each unit is connected to all the RAM banks as illustrated in Figure 4.5.

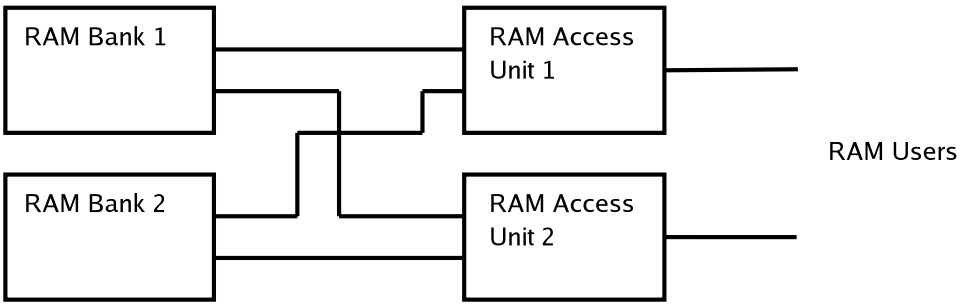

Figure 4.5: Handel C RAM access units with 2 banks.

Units are activated using a simple function call of the form:

```
ram_accessN(rw, address, write_data, &read_dest)
```

This configuration both keeps code to access RAM abstract from the underlying system hardware, and also makes it easy to vary parallelism in RAM access. For example if only one word of data are needed, code will just call `ram_access1(...)`. However if two words of data can be read consecutively we would use a call of the form:

```
par {
   ram_access1(...);
   ram_access2(...);
}
```

```
(The par{} construct in Handel C runs statements in parallel.)
```

This design also prevents the need for a data bus wide enough carry all data possible in parallel from the RAM banks to all registers that receive data. Instead if only one word of data are accessed at a time, a bus only one word wide is needed from a single RAM access unit.

4.3.2 Providing Mutually Exclusive RAM Access

As mentioned earlier the original reason for this design was to allow externally arbitrated access to RAM. To implement this, I used a simple prioritised lock-request algorithm in the Handel C part of my design. So whenever a device, including the Leon memory controller, wishes to access RAM they must first request a lock using a macro-procedure written in Handel C:

```
macro proc ram_lock(device_id)
```

This returns once a lock has been acquired. Then whenever a device has finished accessing memory, it must call:

```
macro proc ram_unlock(device_id)
```

Locks are granted by a lock granting process also implemented in Handel C, which checks every cycle to see which devices are requesting a lock and then provides one based on a priority order. In my final design the Leon processor has the lowest priority. This is to prevent it starving other devices with lower priority as it is likely to be accessing memory very often during program execution.

4.3.3 Improving Leon Memory Access Speed

The disadvantage of this new memory access architecture is that every memory access made by Leon now requires the granting of a lock. Unfortunately as I have already mentioned this was unavoidable, due to the RAM becoming a shared resource.

In order to try and mitigate this I took advantage of the higher bandwidth access to RAM created by the stripping and parallel access design, and introduced a small read cache. This reads as many consecutive words of data as possible during a single read access, and when a read-hit occurs on the final element of the cache also attempts to read further consecutive data if the memory controller becomes idle.

An alternative method of reducing the locking overhead might have been to acquire a lock at the beginning of an AHB burst transfer, and release it at the end.

4.4 Summary

In this chapter we described extending Leon using the AMBA bus, and the co-processor extension point. These individual components can now be viewed as a single extensible SoC framework in Figure 4.6.

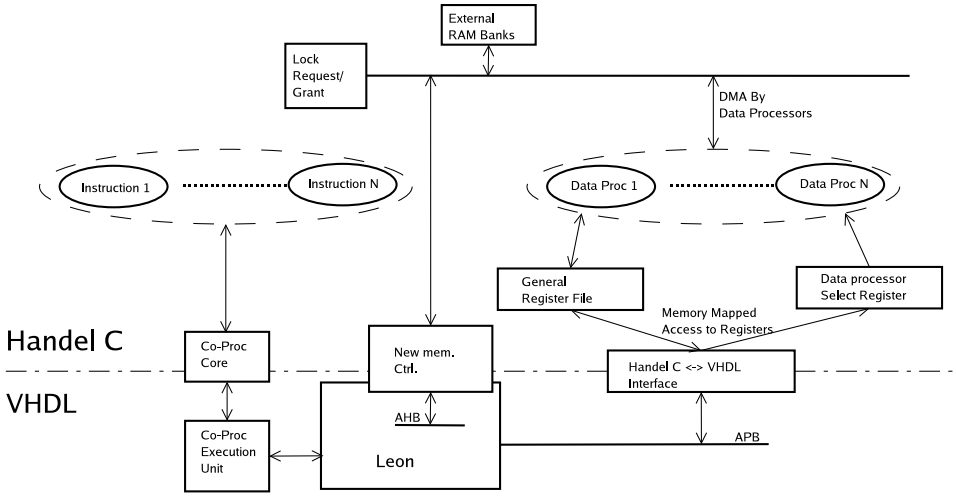

Figure 4.6: Final extensible hardware architecture for a Leon based SoC

This framework is quite close to that proposed in the methodology chapter earlier in Figure 3.1. It mostly achieves the goals of an abstract SoC extension interface built around a sequential processor, by having Handel C layers between Leon and application specific data processors/custom instructions. It also has a potentially fast memory access system by virtue of the replaced Leon memory controller.

Chapter 5

Running On The RC2000

In the previous chapter I discussed how I made a generic framework for extending Leon with Handel C. Once this had been developed my next step was to get this design to work on real hardware, so that I could begin investigating the Ogg Vorbis encoder process.

This turned out to be not as straight forward as it might have been due to some feature incompatibilities between the FPGA prototyping hardware available and Leon. These problems are described in Section 5.1, and a system to solve them using was devised using a combination of hardware described in Section 5.2 and software described in Section 5.3.

5.1 Problems with the FPGA hardware

As mentioned in the background chapter, there were two prototyping boards available to me: the RC2000[1] and the RC200. I found that Leon plus the Gaisler Research FPU (GRFPU), consumed too many resources to fit on the RC200 FPGA, thus leaving the RC2000 as the only viable option.

The RC2000 is a FPGA development board in the form of a PCI card and as such relies on a host PC to function and for all data upload/download. This would turn out to be a bit of a problem when running the Leon processor. For two reasons:

1. Leon expects find the programme it is to run in an attached ROM or to be uploaded via an RS232 connection. Unfortunately neither was

[1] Which is also sometimes referred to by another similar name: ADM-XRC-II.

available on the RC2000.

2. In order to debug programmes running on Leon either an RS232 connection or direct access to a PCI bus is needed. But again an RS232 connection was not available, and although the RC2000 uses the PCI bus, the FPGA is separated will from it by a PCI to 'Local Bus' bridge.

In this chapter I shall discuss how I managed to work around these problems, and produced a combination of hardware and software to run programs in my Leon extension framework on an RC2000.

5.2 Hardware Architecture

The main features of interest available on the RC2000 board are: a high-density FPGA, which is directly connected to 6 independent banks of 512k x 32bit = 12Mb SSRAM, and a local bus which is also connected to the FPGA. This is shown in Figure 5.1.

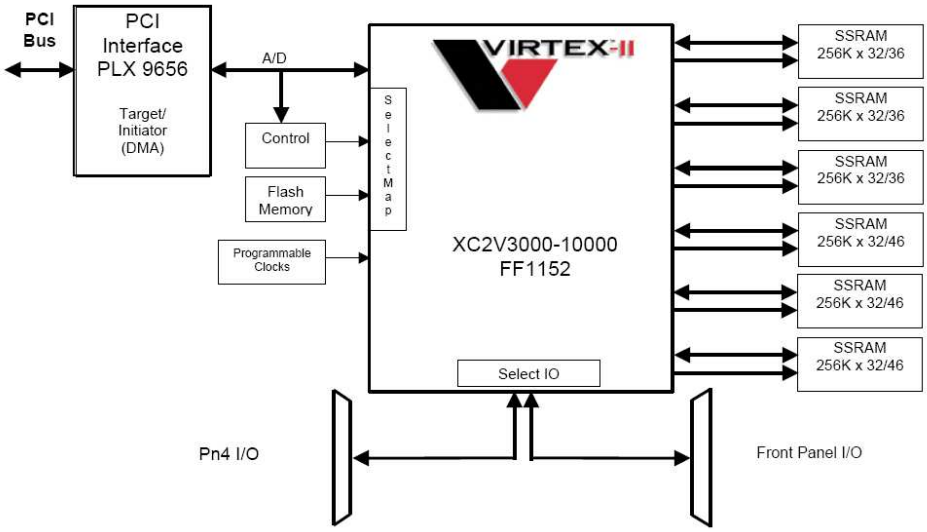

Figure 5.1: RC2000 local bus layout from the user manual[32].

4Mb of the local bus address space is dedicated to the FPGA, and the local bus is bridged onto the host PCs PCI bus as a slave device. The net result of this is that the FPGA can receive and transmit data indirectly but transparently onto the PCI bus across a 4Mb address range. To make use of this, Celoxica provide a Handel C library which can register event handlers

for the reading and writing of data to the local bus and thus the PCI bus. It is also possible to generate IRQs onto the host PCI bus.

In my implementation I divided up the PCI accessible memory range into a number of registers that could control the operation of Leon and to provide a window to directly access to the RAM on the board. This is shown in Table 5.1.

This architecture provides enough control from a host PC to be able to run programs on Leon, and provide minimal debugging support. The operation of this will be described in the remainder of this chapter.

5.3 Software Operation

To go with the hardware I also developed software in the form of a control programme running on the host PC and a simple library for programmes running on Leon. Between the two the boot up of Leon is achieved, and data I/O is provided.

5.3.1 Booting Leon

The uploading of the FPGA design to the RC2000 board, and the initial boot sequence of Leon is orchestrated entirely by the Control Programme running on the Host PC. A brief outline of the operations required is as follows:

1. Upload the Leon bitstream to the FPGA using an SDK provided.
2. Block Leon from accessing RAM by requesting a memory lock.
3. Upload an initial memory image to the ram on the RC2000 using memory mapped access.
4. Patch the image in memory on the RC2000 to correctly set the initial Leon stack pointer to point to the top of Leon accessible memory.
5. Reset Leon.
6. Release the memory lock, allowing execution of the uploaded programme to begin.

Relative Address (in 32bit words)	Description
0x00000 – 0xFFFF5	**SSRAM memory access** – Reads or writes to this range cause a respective read or write to the SSRAM banks. These are laid out as described in Section 4.3.
0xFFFF6	**The Leon 'liveness' register** – Reading this address returns a pattern of ones and zeros from the Leon entity which can be used to check that it has been instantiated correctly.
0xFFFF7 – 0xFFFFA	**Debug/trace registers** – Described in Section 5.3.3.
0xFFFFB	**The memory locking register** – Writing '1' or '0' to this address requests or releases, respectively, a RAM lock as described in Section 4.3.2. Reading from this address returns a nonzero value when a lock is granted.
0xFFFFC	**Leon IRQ register** – Writing to this address causes Leon to receive an IRQ. A read returns the number of IRQs Leon has generated on the PCI bus.
0xFFFFD	**Buffer pointer** – Reading from this address returns a value that can be set by software running on Leon, which indicates the address of a buffer (relative to the Leon CPU address space) used to exchange messages with the host PC.
0xFFFFE	**Leon error register** – Reading from this address returns a nonzero value provided that Leon is not in an error mode.
0xFFFFF	**Leon reset register** – Writing any value to this address causes a reset of Leon

Table 5.1: My RC2000 PCI mapped address space layout.

Under normal operation with Leon, patching of the stack pointer is not necessary. Instead programs are pre-processed with a program called 'mkprom' which turns them into ROM images. This includes a bootstrap loader that copies the program into RAM for execution and sets up memory configuration. However I cannot use this programme as I am not using a ROM. Setting the stack pointer to the top of RAM seems to be sufficient for programme execution though.

Memory images are generated from programs compiled into the ELF format used by GCC, using the 'objdump' utility provided with LECCS (a modified GCC compiler tool chain for use with LEON). This can provide output in the S-record format [31], and in my control program I have written a cut down S-record file reader for this purpose.

5.3.2 Data Exchange Protocol

Once a programme is executing on Leon it will probably require some data I/O. For example in the case of the Ogg Vorbis encoder, data input will be in the form of PCM samples, and data output will be an Ogg stream. It may also be necessary to exchange other messages during execution.

In my control software this is achieved by a simple protocol using IRQs on both Leon and the host PC. An initial request is always made by the program running on Leon. This is done using a simple hardware unit which is added using the framework described in Chapter 4. Once activated this unit generates an IRQ on the PCI bus. When the control programme receives the interrupt it performs whatever action is required, and then uses a special register in the PCI memory mapped range to generate a response interrupt on Leon.

A buffer within the host PC accessible region of the RAM banks on the RC2000 is used to transfer data, and is indicated using the 'Buffer pointer' register in Table 5.1. The buffer has 'type', and 'length' fields, and space for data to be exchanged. In my implementation I have provided for the following different types:

- **Get audio configuration** – Allows the user to supply configuration options for the encoding process e.g. input sampling rate and output quality. These are then requested when necessary by the programme running on Leon. Details of exactly which options can be specified can be found in Appendix D.
- **Send/Receive data** – Send/receive some data, the length of which is

specified in the buffer. Files to be used for the data I/O can be specified on the command line for the Control Programme, as described in Appendix D.

- **Print text** – Allows the programme running on Leon to send textual output which can be displayed by the control programme on the 'stdout' stream.
- **Start/Stop timing** – These messages are used to produce timing results for interesting parts of the program using the Host PCs clock.
- **Program complete** – indicates to the host that the running programme is now complete.

5.3.3 Debugging running programmes

Occasionally there will be problems when running programmes which are so severe that they cannot print an error message for debugging purposes, for example unexpected processor exceptions. To help deal with this I introduced a memory access trace buffer. This is an optional part of the memory control unit which can be enabled at hardware design compile time. This logs all memory accesses including address and data value into a circular buffer. This can then be queried using registers and used to estimate where a programme malfunctioned.

In order to take full advantage of this feature, it is useful to disable caching of instructions in the running programmes. This makes it possible to completely trace programme flow within the limits of the buffer size. This can be done by the control programme patching a `NOP` instruction into the programme memory image at boot time, over the instruction that normally enables the cache.

5.4 Summary

In this chapter I presented a combination of hardware and software which allowed Leon to run on the RC2000 FPGA-based PCI prototyping board. This was able to overcome some of the limitations of the RC2000 which did not supply some of the supporting hardware Leon expected for normal execution, particularly an RS232 connection or a direct connection to a PCI bus.

Between the software and hardware, the boot up of Leon is achieved and rudimentary debugging and data I/O for programmes running is provided.

Chapter 6

Real Numbers In Hardware

One of the key requirements for the extensible hardware framework for creating SoC based audio encoders outlined in the 'Methodology' Chapter, was the development of real number support in hardware. In the Ogg Vorbis encoder, before data is even submitted to the algorithm, each PCM sample must first be normalised to a floating point value in the range -1 to 1. This means that within the domain of the Ogg Vorbis algorithm, almost all operations are performed in floating point. This is also likely to be the case with many other audio encoding schemes. So how to deal with real numbers in an embedded hardware implementation became an issue fairly quickly.

In this chapter I will cover the following topics:

- Different possible approaches to the real numbers problem in Section 6.1.
- Developing a modular interface for real numbers implementations for use in the extensible hardware framework in Section 6.2.
- How I used and adapted the Celoxica floating-point libraries in Section 6.3.
- How I used and adapted the Celoxica fixed-point libraries in Section 6.4.

A quick review of the IEEE 754 float-point number format has also been included for reference in Appendix A.

The resulting hardware investigated and developed in this chapter, is used extensively in Chapter 7 by the acceleration hardware produced for the Ogg

Vorbis encoder. Specifically in the `_vp_noisemask` data processor. The overall effect of varying the real numbers implementations used in the context of the Ogg Vorbis encoder is analysed in the results of testing presented in Chapter 8.

6.1 Options for Working With Real Numbers in Hardware

At the outset of the project I considered three possible alternatives ways of dealing with real numbers within my implementation:

- Follow a similar path to the Ogg-on-a-chip project, and first convert the software algorithm to use fixed point operations. These should be easier to deal with in hardware, and not require the support of an FPU for the software parts to attain a reasonable speed.
- Use floating point computation throughout including in hardware when implementing acceleration. The software side of floating-point can also be supported by an FPU.
- Use a hybrid of floating point arithmetic in software, and fixed point arithmetic in hardware performing conversions where necessary.

I ruled out the first option fairly quickly. While this approach may have been successful for the Ogg-on-a-chip project, I believed it would be less appropriate for this work. The Ogg Vorbis encoder contains considerably more steps than the decoder, and I am working alone rather than in a pair. It looked like this approach would be just too much work to for me to perform well in the time available and particularly given my lack of experience with DSP algorithms. Also unlike in the Ogg-on-a-chip project, I had available to me the new high-speed Gaisler Research FPU for use with Leon, which reduced the need to transform the software parts of the algorithm.

In this project I mostly looked at following the second two options, and shall presently discuss how I made use of Celoxica's floating and fixed point arithmetic libraries to achieve this.

6.2 A Modular Interface for Real Number Implementations

As part of the requirements for my extensible hardware framework I stipulated that the interface for real number units should be modular. The reason for this is that there were likely to be a number of trade-offs in terms of precision and hardware resource utilisation by changing for example between a fixed point or floating point number representation. These trade-offs would be easier to consider if there was a clean interface for switching real number implementations.

To create a modular interface I compiled a list of feature which were common to different real numbers back-end implementations available to me, and that would likely be useful to custom hardware. I then proceeded to develop a simple Handel C interface to encapsulate these.

The common features I identified were as follows:

- Some form of real number data type.
- Constants that represent real numbers used for initialisations, fixed increments etc.
- Arithmetic operations.
- Conversions between the hardware and software representations of real numbers.
- Real number based conditionals.

The first four of these were fairly straightforward to implement in Handel C using: typedefs, constants, and function prototypes respectively. Conditionals were a little trickier, as the floating point libraries I had available did not directly support conditionals. Fortunately there were only a very limited number of conditional cases that I came across when developing custom hardware, and these were easy to implement manually (for example check to see if a sign bit is set to see if a value is less than 0), and I encapsulated these cases using standard C-style macros. Table 6.1 shows the interface in my final design which the different real number implementations followed.

Using this approach meant that adding an alternative real number implementation, was simply a case of following the same interface, and no modifications to any data processing designs would be necessary. Although having said this certain features of the interface were driven by the data processors

```
// Common real data type
typedef ... Real;

// Constants
#define REAL_ZERO ...
#define REAL_ONE  ...

// Conditional macros
#define REAL_IF_LTZERO(x) ... if( ... )
#define REAL_IF_LTONE(x)  ... if( ... )

// Conversion functions
void IEEEFloatSingleToReal(unsigned 32 float_bits, Real *dest);
void RealToIEEEFloatSingle(Real src, unsigned 32 *float_bits);

// Arithmetic units/functions (repeated to allow parallelism)
void AddSub0(unsigned 1 sub, Real a, Real b, Real *c);
void AddSub1(unsigned 1 sub, Real a, Real b, Real *c);
void AddSub2(unsigned 1 sub, Real a, Real b, Real *c);
void AddSub3(unsigned 1 sub, Real a, Real b, Real *c);
void Mult0(Real a, Real b, Real *c);
void Mult1(Real a, Real b, Real *c);
void Mult2(Real a, Real b, Real *c);
void Div0(Real a, Real b, Real *c);
```

Table 6.1: Real numbers interface

using them, for example having 4 add/sub functions was the most efficient number of adds that could be performed in parallel in any users of the interface.

6.3 Celoxica's Floating Point Library

For use with Handel C, Celoxica provide the source code to a rudimentary floating point library. This allows a number of operations to be performed on IEEE 754 floating point numbers but with variable sized mantissas and exponents which can be defined at compile time. In this project I made use of the following operations:

- Add/subtract/multiply, which all operate in a single clock cycle.
- Division, which operate in N + 4 cycles, where N is the size of the mantissa in bits.

Unfortunately the single cycle operations, make this library highly susceptible to producing circuits with high clock cycle periods, and hence slowing down the overall design. However when implementing hardware acceleration I decided to use this library as a starting point, as it was simple to work with in terms of timing, and able to easily replicate, in terms of accuracy, operations that would be performed on a normal processor.

Using this library alone also allowed quite a bit of flexibility in terms of the trade-off between hardware size and numerical accuracy. This can be controlled by varying the size of the mantissa used. Figure 6.1, shows approximately how changing the mantissa effects the hardware size of the floating point units in a design with 4 add/sub units, 3 multipliers, and 1 divider (as used in Table 6.1). Varying the mantissa size also has a profound effect on the speed of the divide operation.

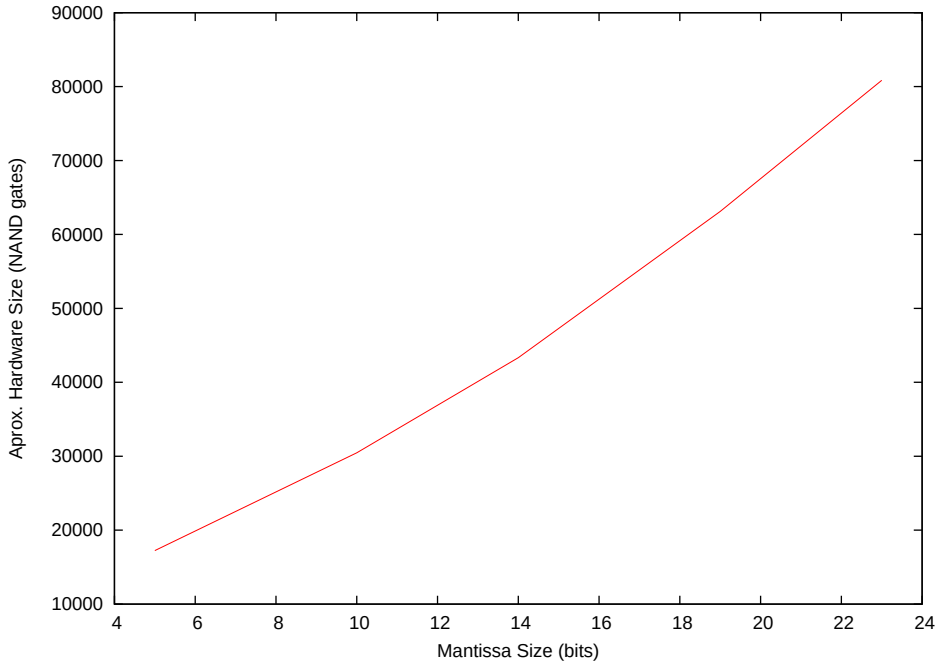

Figure 6.1: Hardware size of a design with 4 add/sub units, 3 multipliers, and 1 divider, as estimated by the Handel C compiler.

Initially I produced the data-processor to accelerate the `vp_noisemask` function as described in Chapter 7 using just the Celoxica floating point library. Once this was operational my next step was to begin to look at ways to improve performance by adapting the library using the source code available.

In the library, I modified the add/sub, multiplication, and division functions in an attempt to improve hardware speed and area. Here I will briefly cover these changes, and then present a summary of their impact on area and clock period independently of the design actually using them. I will look more closely at how the changes affected the Ogg Vorbis acceleration data processors in the 'Testing and Evaluation' chapter.

Multiplication

Of all the floating point units, multiplication had the most potential for acceleration and area use reduction. The multiplication function provided, performed all of the operations necessary for floating point multiplication in a single cycle. This included multiplying the mantissas of the two parameters using purely combinatorial logic implemented using FPGA generic logic resources. This resulted in both large and slow hardware.

However, the Xilinx FPGAs I was using had a number of hardwired 18x18-bit signed 'block' multipliers, which could be accessed from Handel C code. So to take advantage of this I replaced the combinatorial multiplier logic with a block multiplier, and split the complete operation over two cycles.

The unfortunate downside of this modification was that it limited the maximum size of the mantissa to 14 bits, due to constraints in the pre-made floating point hardware design, and the unused sign bit.

As well as replacing the multiplier core, I also removed the logic that checked for the special floating point number cases of INF and NaN values and propagated them. My reasoning behind this decision was I observed that these cases only very rarely arose in the software implementations of the Ogg Vorbis algorithm, and mistakes at these points could fall into the class of imprecision caused by the hardware.

For a 14 bit mantissa, the hardware size was reduced from approximately 7091 NAND gates to 1103 NAND gates + 1 block multiplier.

Division

In the division unit I found very few opportunities to optimise for speed or area. It has no particularly long data paths that could be split apart to decrease minimum clock-period, or any parts which looked like they might benefit from any hard-wired features of the FPGA.

All I changed in the end was to remove the input and output INF/NaN checks. This resulted in an area decrease from approximately 4052 to 3344 NAND gates when using a mantissa of 23, a saving of roughly 17%.

Addition

When using the floating point library, I actually used an addition unit for both addition and subtraction. This was achieved by xor'ing the sign bit of the second parameter passed to the unit with another parameter bit used to indicate whether the operation is to be a subtraction or addition. Thus changes to the addition implementation also affect subtraction.

Of the three basic floating point operations, the hardware used to implement addition is by far the most complex. It has a large number of different signals and parallel operations that need to be performed in order to complete the operation, which is a huge drag on clock speed to do in a single cycle.

As with the divide and multiplication functions, I removed all references to handling the floating point special cases. However beyond this the only improvements that could really be made, were by spreading the operation of the unit over several cycles, using registers to buffer intermediate values. This would hopefully shorten some of the path lengths, allowing a lower clock period.

A discussion of how I went about splitting the adder into two cycles, effectively pipelining its operation, is covered in Appendix C.

Effects of Changes

As a result of these cumulative changes in a system with just 4 add/sub units, 3 multipliers, and 1 divider and no other hardware, I was able to increase the maximum clock frequency attainable from approximately 29MHz to 41MHz, roughly a 30% increase. The overall saving in hardware area size is illustrated in Figure 6.2, and with a mantissa size of 14bits, the total area size is reduced by 27%.

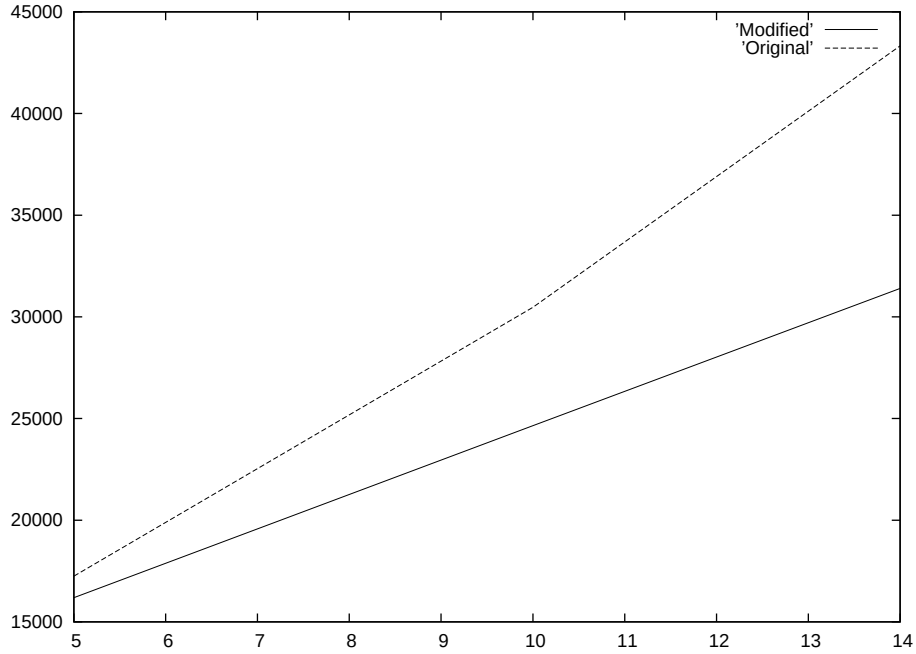

Figure 6.2: Hardware-size saving in a design with 4 add/sub units, 3 multipliers, and 1 divider, as estimated by the Handel C compiler.

It may at first seem self defeating to double the time taken for the addition and multiplication units, but not double the maximum possible clock rate as a result. However Amdahl's law tells us this is not the case. The time spent in these units is likely to be small compared to the time spent in the rest of the system which will be benefited also if the maximum clock rate can be increased.

6.4 Fixed-Point Real Numbers

Sadly it turned out that the fixed point hardware I developed was not of much use when accelerating the Ogg Vorbis algorithm. The main reason for this is that the `vp_noisemask` function implemented in hardware seems to require roughly 44 integer bits of accuracy in a fixed-point number before the results become meaningful. The sizes of hardware at this level of accuracy however were considerably bigger and slower than floating point hardware, which seemed to mostly negate the advantages of fixed-point. So this part of my project is partially unfinished with not much attention paid to optimising the design. However I shall still briefly discuss it here as it is completely

reusable, and may be useful for use in other projects.

Along with Handel C, Celoxica provide a fairly comprehensive fixed point arithmetic library. This makes use of parameterised data-types, and so it is possible to perform fixed point operations at an arbitrary precision either signed or unsigned provided these variables are constant at compile time.

Unfortunately this library alone was not sufficient to effectively implement the interface described in Table 6.1, for two reasons:

1. There were no pre-written functions to translate between IEEE floating point representations and fixed point. For the floating-point library this had not been much of an issue as conversion had simply involved padding with zeros or dropping bits appropriately.
2. The multiplication and division functions were implemented entirely using combinatorial logic which completed within a single clock cycle. So for even modest sized fixed point numbers multiplication and division operations generated enormous and slow circuits.

Conversion Functions

To deal with the lack of conversion functions, I implemented my own hardware units for performing this. For the conversion of floating to fixed point, where the source number was outside of the range representable by fixed point size used, I set the resulting fixed point values to be the highest/lowest values possible or zero depending on positive/negative overflow or underflow respectively.

Improving Multiplication

Multiplication in fixed point arithmetic is fairly straight forward. For two fixed point numbers in the same format, it basically just consists of a regular integer multiplication followed by a right shift. So in order to try and improve the multiplication core situation, I again made use of the block multipliers available on the Xilinx FPGAs.

I quickly realised that the 17 bits of unsigned precision provided by the block multipliers would not allow numbers of a sufficient range for the algorithms I was using. So I attempted to combine four of them together to make a 34 bit multiplier, as shown in Figure 6.3. This pattern could have been repeated again to allow an even larger multiplier, however I gave up before

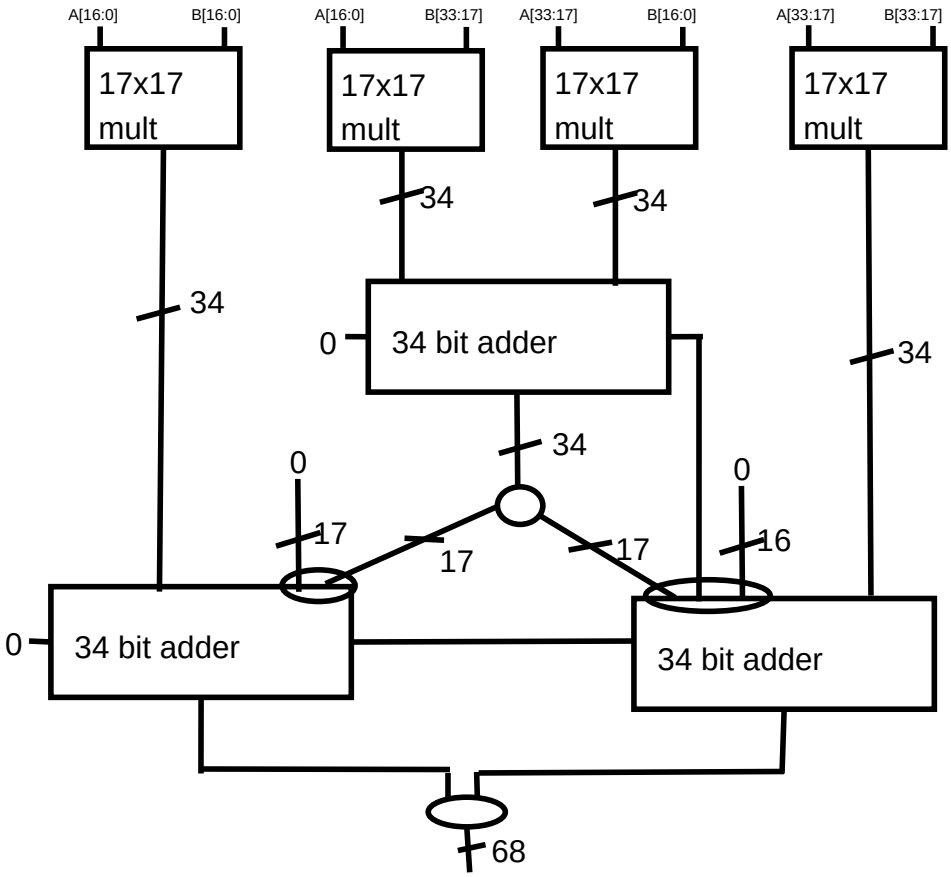

Figure 6.3: A 34x34 bit multiplier built from 17x17 multipliers. Inputs are A and B.

this point as the hardware usage for fixed point had already become greater than that for floating point with sufficiently accurate mantissas.

Multi-Cycle Division

As a stop-gap solution to implementing a more economical division unit, I re-used the modified floating point divider described in 6.3. This however was a rather inefficient solution, as it involved the overhead of converting from fixed point to floating point and back. Again I stopped investigating better solutions once I realised that fixed point representations were not yielding a particularly beneficial alternative to floating point.

6.5 Summary

In this chapter I proposed a modular interface that could be used in my extensible hardware framework for working with real numbers. I considered implementations based around both floating and fixed point real number representations. As part of this, I attempted to adapt the pre-made libraries supplied with Handel C to improve their performance when synthesised onto an FPGA. The results of these modifications resulted in a best case 30% increase in clock speed, and a 27% decrease in hardware size at the expense of halving the possible processing rates of the multiplication and addition units.

The resulting hardware produced from this chapter is used extensively in Chapter 7 by the acceleration hardware produced for the Ogg Vorbis encoder. The results of varying the different float point number implementations in the context of the encoding process are assessed in Chapter 8.

Chapter 7

Running and Accelerating The Ogg Vorbis Encoder

Once I had created enough of my extensible hardware framework to begin executing programmes, I turned my attention to the second goal of this project. This was to use this framework to produce an implementation of an audio encoder SoC by applying it to the Ogg Vorbis algorithm.

To do this I roughly followed the scheme suggested in Section 3.3 which described an iterative approach to applying the framework to software algorithms. Due to time constraints, I was only able to complete roughly one iteration of this, in which I implemented the 'noise masking' portion of the Ogg Vorbis encoder using a hardware data processor.

In this chapter I describe the steps I took to produce an accelerated version of the Ogg Vorbis encoder using the framework developed. The discussion is broken down into the following Sections:

- Section 7.1: Profiling the Vorbis encoder, and identifying areas that might benefit from hardware support.
- Section 7.2: Tweaking the Leon configuration options for running the Vorbis encoder software.
- Section 7.3: An overview of my hardware implementation of the Vorbis 'noise masker' as a data processor.

An analysis of the speed-ups gained and the resulting quality trade-offs caused by this implementation is presented in the next chapter.

All work described here was based around a reference implementation of an Ogg Vorbis encoder, included with libvorbis version 1.0 which can be downloaded from the Vorbis website [2].

7.1 Profiling The Encoder Software

In order to develop a system on a chip implementation for supporting the Ogg Vorbis encoder software, two important features needed to be considered:

1. Will the hardware resources available be sufficient to run the programme, and at what quality level.
2. Which sections of the algorithm are well suited to hardware acceleration, and would provide sufficient benefit to warrant it.

To evaluate the first point, I made extensive use of the simulation software provided with Leon: 'Tsim'. This is claimed to be a highly accurate and fast SPARC v8 simulator developed to match the Leon hardware. Unlike Leon itself, the simulation software is not free, and I was restricted to using a 'demo only' version which had a number features missing. These included the ability to adjust simulated cache settings, RAM size, and profiling support.

Fortunately using the demo, I was still able to determine that the Ogg Vorbis encoder would have sufficient resources to run on Leon using the FPGA hardware available. However it quickly became clear that at the speed I would likely be able to run my synthesised hardware at, the maximum input data rate and output quality that would be achievable in real-time were likely to be highly restricted.

This was an important early discovery as it allowed me to determine a range of input rates and output quality[1] to target support for. This was useful to know, as increasing quality settings and input rate in the Vorbis algorithm non-linearly increases the amount of processing that needs to be done. This is due to extra features for example 'Stereo Coupling' being introduced for multiple channels and higher qualities, and larger windows of data being processed.

[1] 'Quality' in Ogg Vorbis is usually defined by a single floating point value in the range -0.1 to 1.

Once I had determined that the encoder would run with the resources available, I turned my attention to looking at where it could be optimised by profiling executions of the code. As profiling was not available in the demo version of Tsim, I profiled the algorithm executing on a regular x86-based desktop PC using the 'Valgrind' software profiler [28] with the 'Calltree' skin [29]. I then analysed the results of this using the 'Kcachegrid' visualising utility [29]. My hope was that this would be sufficient for identifying heavy processing areas of the algorithm, independent of processor architecture.

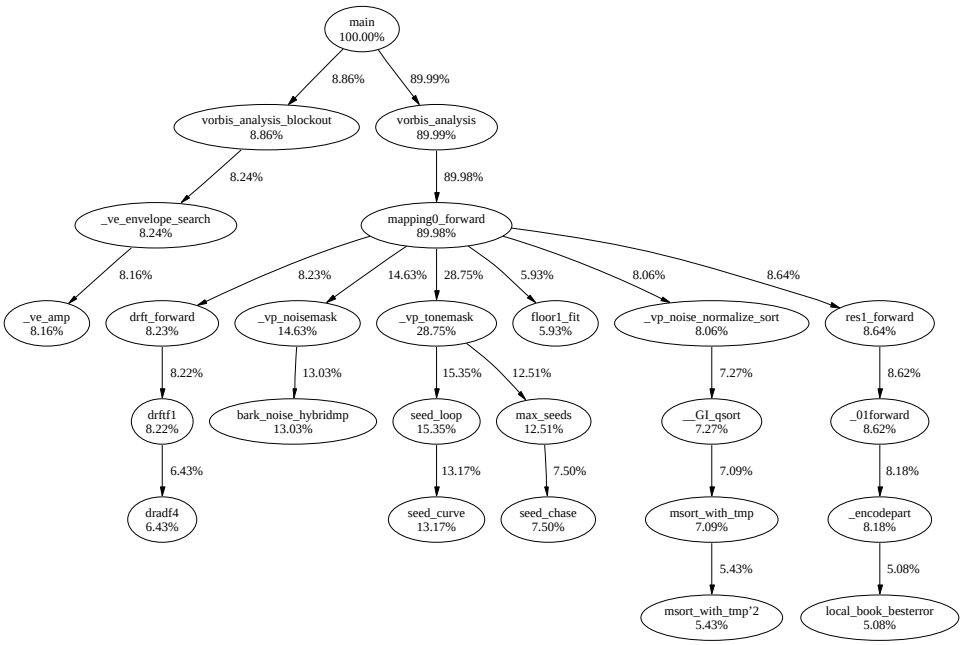

Figure 7.1: Call-tree of the unmodified encoder with relative processing times.

Figure 7.1 shows a call-tree of some of the heaviest processing functions in the encoder produced by Kcachegrind. The percentage values are the relative amounts of time execution took in that part of the tree, when encoding a 6 minute 8000Hz mono audio file using a quality setting of 0.4. Note that the graph only shows the biggest functions in terms of run-time, and many sub-calls are not shown.

Looking at this graph, and examining the source code I determined my first two functions of the interest to be `_vp_noisemask` and `_vp_tonemask`. These had no sub-calls not shown on the graph, and were reasonably easy to follow the execution paths of.

`_vp_tonemask` and its sub functions, did not look particularly hardware friendly. The executable content consisted mostly of large numbers of pointer dereferences, and loops containing conditionals. These would have involved a large number of sparse memory accesses which would not be very efficient to do without a proper cache, and would put a hardware implementation at a disadvantage to a software processor. However what I did find was that the majority of this function could be bypassed at the expense of reducing the final compression ratio achieved by the algorithm, with output files increasing in size by about 17-20%. Figure 7.2 shows another call-tree profile of the encoder, with the majority of this function disabled.

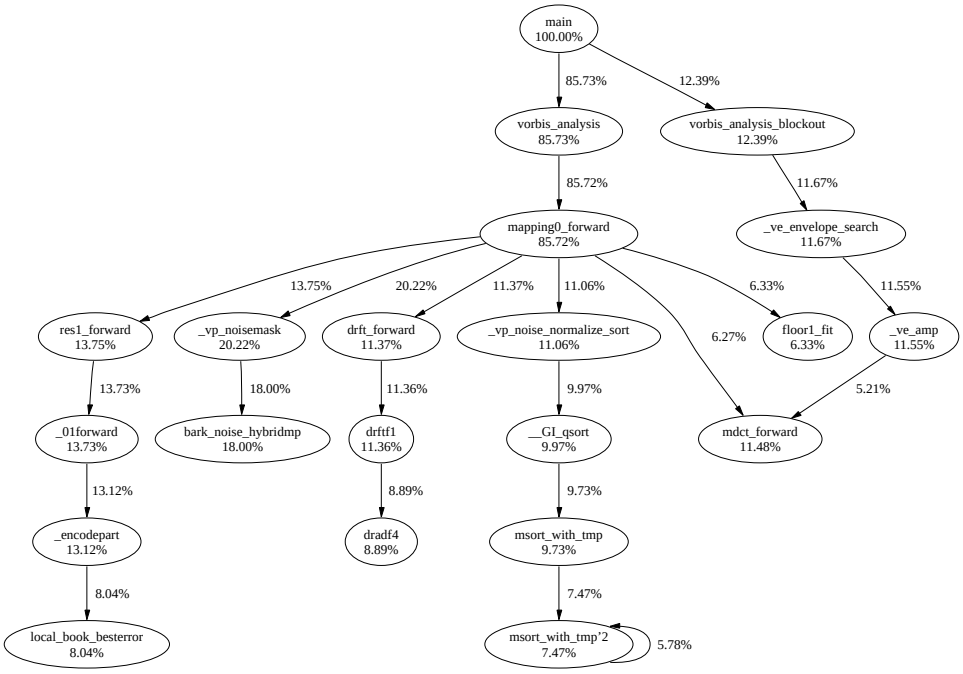

Figure 7.2: Call-tree of the encoder with reduced tone-masking functionality.

This second graph reinforced that `_vp_noisemask` would be a good candidate for hardware acceleration. Inspection of the code revealed that both `_vp_noisemask`, and `bark_noise_hybridmp` made intensive use of linear memory accesses and repeating patterns of arithmetic computations both of which could quite easily be taken advantage of in a hardware data processor. The graph also revealed `mdct_forward` as another potential candidate.

In the time available for this project I was able to implement the `_vp_noisemask` and `bark_noise_hybridmp` functions in hardware. These were both rolled together into a single 'Noise masking' data processor using the extensible Leon framework.

Within the Ogg Vorbis code I also noted a number of repeating patterns, or common blocks of code which might have benefited from being implemented as custom instructions. However I did not attempt to implement them. The reason for this is I had trouble seeing how implementing them it would be possible to beat the Leon processor and high-performance GRFPU. One of the main problems was the interface provided by Leon, which requires the co-processors to work on their own private set of registers. So although it might be possible to run a common operation faster than could be performed by the FPU + CPU, the only way to transfer the results back to either of them for further processing was by memory load/store operations on the co-processor register file. The overhead of this appeared to generally negate the benefits of highly specific custom instructions.

7.2 Tweaking Leon for The Encoder Software

The Leon SoC has a large number of configurable options, and thus it would be foolish to not investigate which of these would be optimal for running the encoder. Fortunately most of these options cover adapting Leon and its peripherals to the hardware environment it is to be run on. Of most significance to the potential execution speed of programmes were the cache configuration options.

Leon sports a highly configurable, single level, Harvard architecture cache sub-system. It is possible to select independently for the data and instruction caches: the replacement algorithms, cache sizes, line sizes and set associativity.

Again due to the restrictions of the demo version of 'Tsim', it was not possible to investigate different cache configurations with Leon directly. So instead I once again turned to 'Valgrind' now using the 'Cachegrind' skin. Using this, I varied an emulated Pentium L1 data cache configuration and attempted to predict cache performance on Leon, by observing the magnitude of the data cache misses. This seemed like a not unreasonable thing to do as the pattern of memory access, for data at least, would hopefully be fairly similar when running the same programme built with the same compiler.

The results of this brief investigation are shown in Table 7.1. From these I concluded that when running the Ogg Vorbis encoder, the optimal configuration for the data cache, in terms of hardware size efficiency, was to have a line size of 32, a set associativity of 2 and the bigger the total cache, the better. Unfortunately it would not really be sensible to make such judgements about the instruction cache using 'Cachegrind' as the architectures

Cache Size (bytes)	Assoc.	Line Width (bytes)	Data misses
16384	1	32	42,727,216
16384	1	16	67,754,124
16384	2	32	27,917,457
16384	2	16	47,835,524
16384	4	32	23,555,348
16384	4	16	41,748,896
32768	1	32	32,783,266
32768	1	16	51,554,189
32768	2	32	17,656,235
32768	2	16	30,335,432
32768	4	32	17,519,694
32768	4	16	29,717,605
65536	1	32	26,328,751
65536	1	16	41,224,479
65536	2	32	8,938,079
65536	2	16	13,693,105
65536	4	32	8,250,883
65536	4	16	12,836,180

Table 7.1: Data cache misses running the Ogg Vorbis Encoder in Cachegrind.

of the Pentium and Leon are so different. So for lack of a better idea, I just duplicated the data cache configuration for the instruction cache.

7.3 A Hardware 'Noise Masker' Data Processor

Before we continue, I should first admit that despite having spent quite some time looking at implementing the functions which make up the Vorbis "Noise masker" in hardware I still have very little idea of how it actually performs this task. There is little in the way of hints in the source code for someone with as little experience in DSP/Psychoacoustic models of audio as myself to aid in the understanding of this function. So when attempting to implement it in hardware, my perspective was entirely from that of a person trying to optimise based on only the source code available.

As I did not understand the functions in an abstract sense, I had to attempt to achieve a speed-up by maximising levels of parallelism within a hardware data path equivalent to the code being examined. It would not be enough to simply translate the code directly into an equivalent hardware representation, as this would likely end up being slower than the sequential processor

version. This would be due to the increased latency when accessing memory without the aid of a full cache, the overhead of initialising and collecting results from an external data processor, and Leon + the GRFPU's ability to complete one whole instruction per cycle under ideal conditions already.

The noise masker itself is made up of two functions: `bark_noise_hybridmp` and `_vp_noisemask`. As the operation of these is fairly independent, I shall discuss how I implemented them in hardware separately in the following two sections.

7.3.1 `bark_noise_hybridmp`

It is in this function that most of the actual work of the noise masker takes place. Its basic task is to take as input an array of spectrum energies, and generate a corresponding energy mask that can be applied to that spectrum to remove the effects of noise[2]. An overview of its concrete flow of operation is shown in Figure 7.3, and a more detailed view can be found in the source code itself, which has been included and annotated with reference to this diagram in Appendix F.

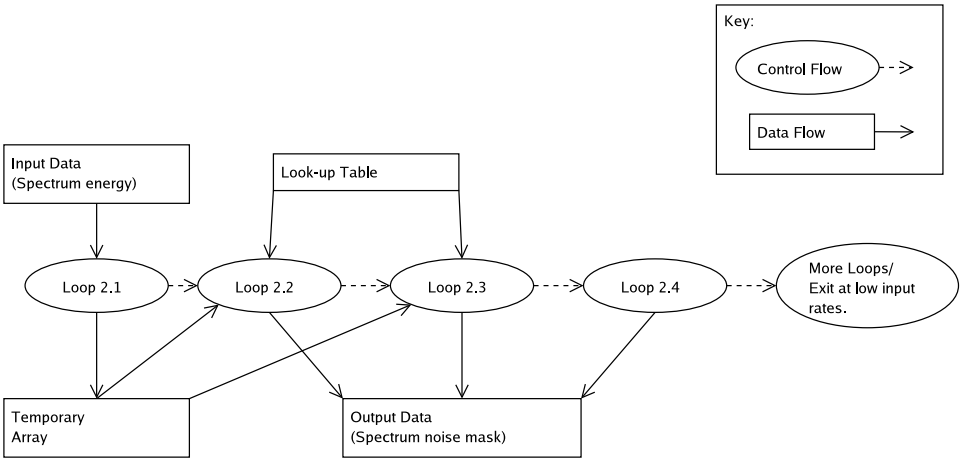

Figure 7.3: Data and control flow in `bark_noise_hybridmp`.

When implementing this function using Handel C, I produced an implementation which basically matches the same flow of sequential operation as illustrated in the diagram. Although as is suggested, I did not implement the entire function in hardware as roughly half of it is not needed at the data input rates I am targeting. Any actual speed gains come in the form of

[2]Or at least this is what I have inferred from the parameter and variable names.

parallelism introduced within the Handel C versions of the loops themselves, so I shall discuss these individually.

Loop 2.1

In this loop, an array is generated which is used by loops 2.2 and 2.3. This array is accessed in a non-linear fashion (based on the input look-up table), and is constructed cumulatively using the input energy spectrum provided. These features make it difficult to re-factor this array as a function which can be used by the other loops.

The main points in this loops where parallelism is gained are:

- Over-lapping the memory access of the input data with the arithmetic operations required to generate the corresponding entry in the array.
- Performing as many of the arithmetic operations in parallel using multiple real number units wherever possible.

It is possible to quantitatively compare my hardware implementation of this loop with the software version by estimating the number of cycles taken in the core of the loop for each. For my hardware version the core executes in exactly 6 cycles per iteration. For the software version if we assume optimal conditions with no cache misses, results always being available when needed, repeated expressions are re-used and one instruction is completed per cycle we have approximately 21 cycles per iteration. A minimum speed up of 3.5x.

Loop 2.2 and Loop 2.3

These two loops use the array generated by Loop 2.1 to generate the output mask. They are also almost identical, the only difference being their exit condition which is based on the input look-up table shown in Figure 7.3. Using this similarity I created a single functional unit to implement both of these loops, which appropriately selects which exit condition to use when invoked.

The main constraint on the maximum speed these loops can be executed when using the unmodified Celoxica floating point library from Section 6.3, is a division operation which occurs at the end of loop. Almost all operations in the loop can be overlapped with this, which leads to a minimum of N + 8 cycles for each iteration where N is the size of the mantissa used in bits. This

can again be compared to the software version running in ideal conditions which has an approximate minimum rate of 39 cycles per iteration. If we assume a mantissa size of 10 (which as we shall see in the next chapter gives a reasonable output quality) we have a minimum speed up of 2.2x.

Loop 2.4

Loop 2.4 basically just consists of a division operation similar to that in Loops 2.2 and 2.3. The core of this loop executes in N + 6 cycles. At first glance this appears to be worse than the software version which under ideal conditions would take approximately 7 cycles. Fortunately it is still possible for the hardware version to run faster than the software version, as the optimal conditions can never be reached in Leon using the GRFPU. The reason for this is that the GRFPU also only has a single division unit, which itself takes 14 cycles to complete, and so is similarly constrained.

Despite the limited potential speed up for this loop, I have still implemented it in hardware as it allows the whole of `_vp_noisemask` to be implemented without the overhead of switching between hardware and software contexts. Also, my own investigations revealed that on average, this loop is only executed a small number of times in comparison to the other loops.

Storing the Temporary Array

As has already been mentioned Loop 2.1 generates a temporary array which is then used by Loops 2.2 and 2.3. When investigating implementing this in hardware, I had two possible options for dealing with this array. I could either use my memory interface to store and access the array from the system RAM, or alternatively generate my own RAM just for this function using the hardwired block RAMs available on the FPGA. The reason for considering these alternatives is the way that the array is accessed in Loops 2.2 and 2.3, which can be expressed abstractly as:

```
A = ARRAY[i]   +/- ARRAY[j]
B = ARRAY[i+1] +/- ARRAY[j+1]
C = ARRAY[i+2] +/- ARRAY[j+2]
D = ARRAY[i+3] +/- ARRAY[j+3]
E = ARRAY[i+4] +/- ARRAY[j+4]
(where each element in ARRAY is 32bits wide)
```

A single access to an external SSRAM bank on the RC2000 takes 3 cycles, and the maximum number of memory banks which can be used with my custom memory interface[3] on the RC2000 is 4. So with a maximum of 128bit reads in any one go, the maximum speed this code can execute is 13 cycles = 3 cycles read + 3 cycles read + 3 cycles (read + first set of additions A - D) + 3 cycles read + 1 cycle final addition (E). The main drag being the fact that actually 2x160bits are needed which requires an extra two memory accesses.

Using the block RAM approach, the above code can be executed in 3 cycles with parallel access to 5 independent memory blocks, a considerable saving. The disadvantage of this is that the array size would have to be known in advance in order to allocate the RAM blocks in the hardware design. For the low sampling rates that I was targeting, I found than this size was invariably constant, but for higher sampling rates this size could increase.

I tried implementations using both approaches, and eventually decided to go with the block RAM method.

7.3.2 `_vp_noisemask`

Once I completed my hardware version of `bark_noise_hybridmp` I then went on to implement most of `_vp_noisemask` in hardware as well. Fortunately the control flow of this function is considerably less complex. An annotated version of its source code can be found in Appendix E and can be summarised as follows:

1. Run `bark_noise_hybridmp`.
2. **Loop 1.1**: Perform a vector subtraction.
3. Run `bark_noise_hybridmp`.
4. **Loop 1.2**: Perform a vector subtraction.
5. **Loop 1.3**: Adjust the final result array with a look-up table.

The specifics of what these operations achieve is not important here, the thing to note is that Loops 1.1 and 1.2 are implemented in hardware and after the second call to `bark_noise_hybridmp` is completed, control is returned to software for the final loop.

[3]The number of memory banks must be a power of two, to allow selection of a RAM bank based on a minimal fixed number of low bits in the address.

Loops 1.1 and 1.2 show the advantages of my custom memory interface. They are both almost identical vector subtractions operating on two independent arrays. In hardware the loop kernel of each operates in the equivalent of 2.5 cycles (as 4 iterations are grouped together for parallel operation). In optimal conditions this same operation in software would be about 6 cycles per iteration.

7.3.3 Software Support

To support the hardware version of the 'noise masker', I replaced the software `_vp_noisemask` function with an equivalent function that calls the hardware data processor appropriately. Initialisation and activation of the hardware is fairly straight forward and is performed by filling in registers on the APB bus as described in Section 4.1. A quick check is also performed to ensure that the parameters are within the bounds of what the hardware version can do, and if not the original software version of the function is called instead.

After the hardware data processor is complete, an important step before returning control to the normal program flow is to update the processor cache, which will now be out of sync with the contents of system RAM. This is done by explicitly invalidating the relevant addresses in the cache using special SPARC assembly instructions.

7.4 Summary

In this chapter I described how I attempted to accelerate the Ogg Vorbis encoder software by adding custom hardware to a SoC using the framework created earlier. I began by profiling the algorithm, and identified the two busiest functions in the encoder software. I discovered the busiest of these could be disabled at the expense of increasing file size output and implemented the other as a hardware data processor.

I did not make use of the custom instruction interface that I had developed as I couldn't find any obvious way to use it to provide significant benefits over Leon and the GRFPU.

In the next chapter I will present an analysis of how the accelerated encoder SoC performs in terms of processing speed and quality of output.

Chapter 8

Testing and Results

In this chapter I will present tests of the accelerated Ogg Vorbis encoder. I will begin by discussing the goals that are intended to be achieved with testing in the context of the accelerated Ogg Vorbis Encoder hardware in Section 8.1. Then I will provide details of experimental constants used in testing in Section 8.2 and present the results of tests in following categories:

- The speed-ups gained by the acceleration hardware out of the context of the full encoding process (Section 8.3).
- Performance and quality of the full accelerated and un-accelerated encoding process (Section 8.4).
- Completeness of the implementation (Section 8.5).
- Generated hardware size and speed characteristics (Section 8.6).

Where appropriate, a summary of observations is included for each testing category. An evaluation of these results, and the designs from this project as a whole will be presented in the next chapter.

8.1 Testing Goals and Issues

In the case study of the Ogg Vorbis encoder, I attempted to use the framework created in this project to achieve a speed increase when executing the encoding software on an embedded system on a chip. To judge the success of this I considered:

- The performance benefits which accrued from this design.
- Sacrifices in the quality of output caused.
- Hardware resources utilised.

Using the implementation described, quality trade-offs are introduced by varying the precision of the floating point hardware used, and by disabling certain features of the encoder. To try and observe these effects I considered the speed and quality of the system when varying the following parameters:

- Number of bits in the mantissa used by the floating point hardware.
- Using the modified or unmodified floating point libraries discussed in Chapter 6.
- Encoding with or without the 'tone masking' functionality of Vorbis enabled. In Chapter 7, I suggested this feature could be disabled with apparent effects only on the size of the encoded stream, and not its quality.

As precision of the floating-point hardware is varied, it is likely that I will find the size of the output stream will change as well, as the accelerated portions of the encoder are involved with identifying information that can be removed from a signal. Altering the size of the output stream is also likely to cause a change in overall processing overhead. This will undoubtedly lead to a fairly complex relationship between the precision/size of hardware and the resulting performance benefits in terms of both quality and speed.

To try and separate the effects of performance changes due to fluctuations in the amount of data being analysed or hardware acceleration, I considered the acceleration of procedures both in the context of the full encoding process and independently of it.

Finally I will have a look at the characteristics of the hardware used in these tests with regards to area of an FPGA utilised, and the maximum clock speeds that can be attained by the design.

8.2 Experimental Constants

8.2.1 Audio Clips Used

In the test experiments performed, two audio clips were used:

- **Clip A** 'Set the World Afire' by the band: Megadeth, of the genre: 'Thrash Metal', length 5:48 minutes.
- **Clip B** 'Doodlin' ' by the band: Horace Silvers and the Jazz Messengers, of the genre: 'Blues', length 6:44 minutes.

These two clips were chosen due to their starkly contrasting audio styles. The thrash metal clip is high-tempo and has high levels of noise, whereas the jazz/blues clip is relatively slow and tonal in nature.

When profiling the Ogg Vorbis encoder, it quickly became apparent that only low sampling rates would be possible to achieve in real time using the tools available. As a result of this the acceleration hardware I have produced only targets encoding audio with low sample rates and with only a single channel. So unless otherwise stated, the experiments presented here all use versions of the audio clips that have been pre-down-sampled to a rate of 8000Hz in monotone.

8.2.2 Vorbis Quality Settings

By default Vorbis uses a very simple quality level system. Quality is specified by a floating point number ranging from -0.1 (Bad) to 1.0 (Good). This results in a 'Variable bitrate' output stream, however it is also possible to constrain the output bitrate by enabling a much slower option called 'bitrate management mode'.

For the sake of reducing the number of experiments that needed to be performed, I did not use bitrate management, and fixed the quality value at a level of 0.5 for all tests (unless otherwise stated).

8.2.3 Hardware Variations

For testing, 7 different hardware designs were compiled with varied real number implementations:

- **Hardware A5** - Original Celoxica Floating Point Library with a mantissa size of 5 bits.
- **Hardware A10** - Original Celoxica Floating Point Library with a mantissa size of 10 bits.

- **Hardware A14** - Original Celoxica Floating Point Library with a mantissa size of 14 bits.
- **Hardware A23** - Original Celoxica Floating Point Library with a mantissa size of 23 bits (conforming to IEEE 754 specifications).
- **Hardware B5** - Modified Celoxica Floating Point Library with a mantissa size of 5 bits.
- **Hardware B10** - Modified Celoxica Floating Point Library with a mantissa size of 10 bits.
- **Hardware B14** - Modified Celoxica Floating Point Library with a mantissa size of 14 bits.

These variations were chosen in order to give a reasonable spread of possible precision levels within limitations of the design allowed ranges. Versions of hardware using fixed point arithmetic units were not used, as even at the maximum level of precision allowed by my libraries the output was distorted beyond all usefulness. Using the original Celoxica fixed-point libraries with suitable precision levels, produced hardware that could only run at clock frequencies too low to be stable on the RC2000 board.

8.3 Testing Speed-up out of Context

In this testing category I observed the absolute speed-up of just the functions of the encoder that were implemented in hardware. To do this I ran the software and hardware versions of just the hardware accelerated functions in a tight loop executed 100,000 times. I then computed the speed-up by looking at the ratios of the time taken for the software only version versus the different hardware versions. The results are shown in Table 8.1.

Hardware Used	Time taken in seconds @ 12.5MHz	Relative Speed-up
Software Only	696	–
A5	170	4.09x
A10	192	3.63x
A14	208	3.35x
A23	244	2.85x
B5	205	3.40x
B10	221	3.15x
B14	238	2.92x

Table 8.1: Absolute speed-up of hardware accelerated functions.

It should be noted that in this Test, the time taken for the hardware functions to execute include the time for the CPU cache to be re-synchronised, which is done using a software subroutine.

Observations

Looking at the results of these tests for the absolute speed-up of functions, it is clear that the hardware implementation does run faster than the software version. This is a useful statistic, as it proves that the hardware functions do actually provide some form of benefit, and that performance increases observed later in the context of the full encoder are likely to be attributed to hardware acceleration.

It can be clearly seen that varying the hardware floating point mantissa has a direct impact on the speed-up achieved. This is regardless of data processing overhead changes due to inaccuracies, as the number of operations performed by these functions is dependent only on the input data, and will not be affected by any resultant imprecision as may be the case in the full encoder context.

Switching between the modified and unmodified Celoxica libraries also has an effect on the processing time, increasing it within a range of 12-16%. The effect of this speed decrease however, reduces with mantissa size increase. This is to be expected as the division operation time length is the same for both implementations and is dependent on mantissa size. As the mantissa size increases, the division operations presumably further dominate processing time.

8.4 Testing Speed-up in Context

Having tested the speed-up provided by creating hardware versions of some of the encoding functions, I now considered the speed up when these functions were used in the context of the full encoding process. From this I hoped to attained quantitative values for quality and performance fluctuations caused by the design.

In theory it should be fair to say that in a completely self-contained embedded system, every test run of the encoder with identical input data should execute in the same amount of time, and produce precisely the same output. However in the testing environment used, while given the same input data output will be identical, it is not necessarily the case that the tests

will run in the same time. The reason for this, is that the SoC part of the system is reliant on a host PC for the transfer of data. The host PC used was running a standard desktop install of the Linux operating system, and so is a non-real-time multi-processing environment which can introduce a non-deterministic time overhead during data transfers.

During development I observed that this non-deterministic element could occasionally be quite significant. To compensate for this I ran separate tests for determining quality and speed of the algorithm.

In the speed tests, I uploaded a 512Kbytes sample from the beginning of the audio clip being tested completely to the SoC. This sample was then encoded 6 times (to increase the timing resolution), and the total time taken recorded. Using this method eliminated any non-deterministic elements caused by delays during processing introduced by the host PC. However as there is no output, it did not allow me to test for quality.

In the quality tests, the entire audio clips were encoded from start to finish using the support of the host PC to transfer data throughout. Tests for quality were then be performed on the output data.

In Section 8.4.3 I have attempted to combine the results for quality and speed together to give an overall system view.

8.4.1 Testing for Speed

The results of testing for speed are shown in tables Table 8.2, Table 8.3, Table 8.4 and Table 8.5. These cover the results for clip A and B, with and without tone masking functionality. The columns represent the following:

- **Hardware** – Indicates the hardware precision used for the test using the variations defined in Section 8.2.3. The 'software only' tests were also performed on the Leon based hardware.
- **Time** – Time taken in seconds to encode the uploaded sample 6 times when the hardware was running at 12.5MHz.
- **Speed Up** – The speed-up introduced when using the custom hardware compared to the software only version.
- **Speed-up (Relative to tone-masked software version)** – This is an additional column for tests where tone-masking was disabled. It gives the overall speed up through the combined effect of hardware acceleration and disabling tone masking. This is done by comparing

the time taken for that test to the time taken using only software with tone-masking enabled.

Hardware	Time in Seconds @ 12.5MHz	Speed-up
software only	461	–
A5	394	14.53%
A10	412	10.63%
A14	418	9.33%
A23	421	8.68%
B5	393	14.75%
B10	412	10.63%
B14	419	9.11%

Table 8.2: Speed analysis for audio clip A, with tone masking.

Hardware	Time in Seconds @ 12.5MHz	Speed-up
software only	466	–
A5	392	15.9%
A10	415	10.9%
A14	420	9.9%
A23	427	8.4%
B5	394	15.5%
B10	414	11.2%
B14	421	9.8%

Table 8.3: Speed analysis for audio clip B, with tone masking.

Hardware	Time in Seconds @ 12.5MHz	Speed-up (Relative to non-tone masked software version)	Speed-up (Relative to tone masked software version)
software only	350	–	24.08%
A5	278	20.57%	39.70%
A10	305	12.86%	33.84%
A14	308	12.00%	33.19%
A23	312	10.86%	32.32%
B5	279	20.29%	39.48%
B10	305	12.86%	33.84%
B14	309	11.71%	32.97%

Table 8.4: Speed analysis for audio clip A, no tone masking.

Hardware	Time in Seconds @ 12.5MHz	Speed-up (Relative to non-tone masked software version)	Speed-up (Relative to tone masked software version)
software only	353	–	24.2%
A5	279	20.1%	40.1%
A10	307	13.0%	34.1%
A14	310	12.1%	33.5%
A23	314	11.0%	32.6%
B5	282	20.1%	39.5%
B10	307	13.0%	34.1%
B14	312	11.6%	33.0%

Table 8.5: Speed analysis for audio clip B, no tone masking.

Observations

- Despite their starkly contrasting styles, processing times are highly similar for both audio clips. As the input data size in these tests is exactly the same for both clips, this suggests that there is probably a rough correspondence with input data size and processing time regardless of content.
- The effect of turning off tone masking functionality alone, provides a significant speed increase; more than the acceleration hardware. This appears to be in the region of about 20-24%.
- The speed-up caused by the hardware acceleration varies between 8-13% (if we ignore the A5 and B5 hardware results, which as we shall see in the next section produce so much quality degradation as to be considered useless).
- The combined effect of disabling tone masking and using hardware acceleration provides a performance increase in the region of around 33%.
- In the context of the full encoder, the effects of switching between the modified and unmodified Celoxica floating-point libraries appears to have a negligible effect on the relative speed-up. However this may be misleading, as we shall see in the tests for quality, this switch also reduces the output file size. This will also cause a decrease in processing overhead, which might be responsible for compensating for a reduced speed-up.

8.4.2 Testing for Quality

Quantitative testing of the output quality from a psychoacoustic audio encoder is implicitly difficult. The quality of the signal is always degraded as information is lost, but the observed quality degradation is subjective to a human listener. To try and gauge quality here, I have used a simple Signal-to-Noise Ratio (SNR) comparison between the resulting decoded output and the original input signal. I found that subjective quality degradation roughly correlated with an equivalent decrease in SNR value. For quality purposes, I also recorded the compression ratios achieved as ultimately, the task of the encoder is to maximise compression and minimise the impact on audio quality.

The results for testing for quality are shown in tables Table 8.6, Table 8.7, Table 8.9 and Table 8.10. These cover the results for clip A and B with and without the tone-masking feature. The columns represent the following:

- **Hardware** – Indicates the hardware precision used for the test (as explained in Section 8.2.3). The 'software only' tests were also performed on the Leon based hardware.
- **SNR** – Indicates the strength of the transcoded signal compared to the original signal, in decibels (dB).
- **Compression Ratio** – Size of the encoded stream compared to the original signal data size.
- **File size change compared to software only version** – Another expression of compression ratio, this time directly relating the fluctuations in file size caused by varying the hardware used.
- **File size change compared to software only version, with tone-masking** – As above, but compared to the software only version with tone-masking.

Hardware	SNR (dB)	Compression Ratio	File size (Relative to software version)
software only	19.476	4.67:1	–
A5	10.744	8.00:1	58.4%
A10	15.94	5.41:1	86.3%
A14	19.022	4.92:1	94.9%
A23	19.995	4.72:1	99.0%
B5	10.739	8.21:1	56.9%
B10	13.337	5.89:1	79.3%
B14	18.446	5.01:1	93.2%

Table 8.6: Quality analysis for audio clip A, with tone masking.

Hardware	SNR (dB)	Compression Ratio	File size (Relative to software version)
software only	24.302	4.96:1	–
A5	13.05	9.97:1	49.7%
A10	20.597	5.64:1	87.9%
A14	23.453	5.22:1	95.0%
A23	24.315	4.98:1	99.6%
B5	13.369	9.99:1	49.6%
B10	18.994	5.98:1	82.9%
B14	22.983	5.31:1	93.4%

Table 8.7: Quality analysis for audio Clip B, with tone masking.

Hardware	SNR (dB)	Compression Ratio	File size (Relative to software version)
software only	24.302	4.96:1	–
A5	13.05	9.97:1	49.7%
A10	20.597	5.64:1	87.9%
A14	23.453	5.22:1	95.0%
A23	24.315	4.98:1	99.6%
B5	13.369	9.99:1	49.6%
B10	18.994	5.98:1	82.9%
B14	22.983	5.31:1	93.4%

Table 8.8: Quality analysis for audio Clip B, with tone masking.

Hardware	SNR (dB)	Compression Ratio	File size (Relative to software version)	File size (Relative to tone masked software version)
software only	18.366	4.02:1	–	116.1%
A5	10.666	6.70:1	60.0%	69.7%
A10	15.393	4.08:1	98.5%	114.3%
A14	17.043	3.91:1	102.7%	119.5%
A23	17.631	3.87:1	104.0%	120.8%
B5	10.609	6.69:1	60.1%	69.8%
B10	13.487	4.19:1	96.0%	111.4%
B14	16.958	3.92:1	102.7%	119.3%

Table 8.9: Quality analysis for audio clip A, no tone masking.

Hardware	SNR (dB)	Compression Ratio	File size (Relative to software version)	File size (Relative to tone masked software version)
software only	24.16	4.11:1	–	120.6%
A5	13.37	8.46:1	48.6%	58.6%
A10	21.113	4.41:1	92.2%	112.5%
A14	23.057	4.21:1	97.6%	117.8%
A23	23.749	4.07:1	100.9%	121.9%
B5	13.696	8.07:1	50.9%	61.4%
B10	19.557	4.53:1	90.7%	109.5%
B14	22.847	4.24:1	96.9%	117.0%

Table 8.10: Quality analysis for audio clip B, no tone masking.

Observations

- Disabling tone masking causes an output file size increase in the region of 10-20% depending on the precision of hardware used.
- Decreasing the precision of the hardware floating point representation caused a decrease in both file size and SNR. This suggests that decreasing precision generally causes the output signal to be over filtered thus leading to a reduced file size and quality.
- Switching between the modified and unmodified Celoxica floating point libraries causes a small decrease in quality and file size. This will be due to the imprecision introduced by removing the INF and NaN checks.
- Subjectively I found that with a mantissa precision of 5 bits, audio output was severely degraded. 10 bits had some noticeable, but very minor effects, and for 14bits and above the output was indistinguishable from the software only version.

8.4.3 Combining the Results

In order to more clearly illustrate the trade-off between speed and quality, I have combined the results for each clip into a single graph. These are shown in Figure 8.1 and Figure 8.2.

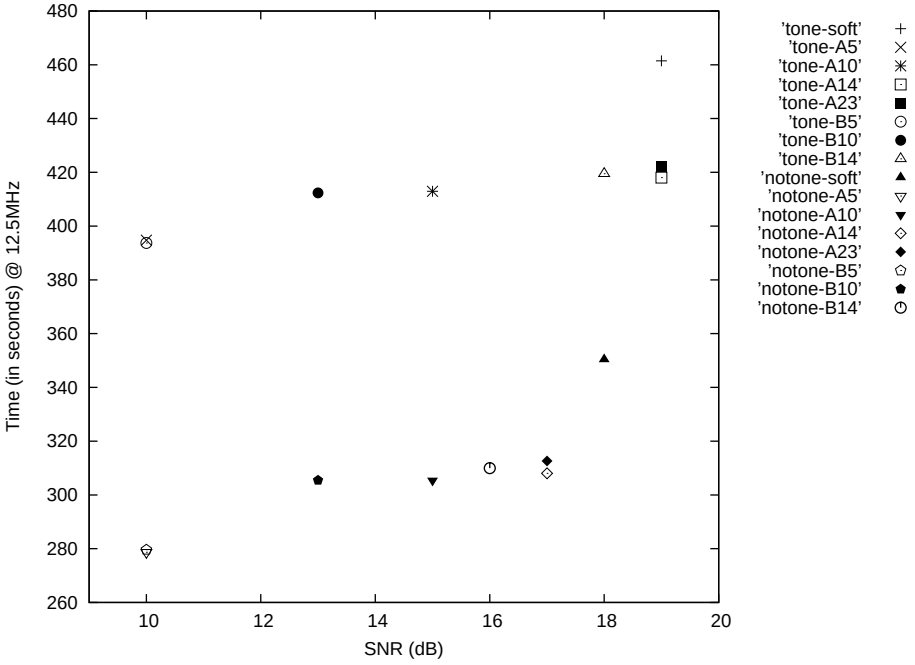

Figure 8.1: Combined quality and speed results for Clip A.

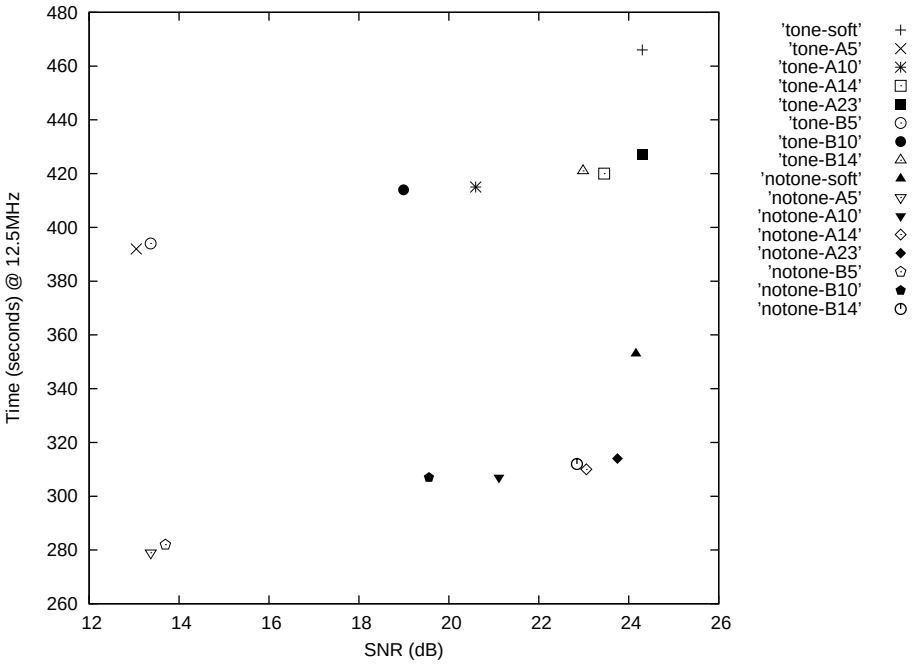

Figure 8.2: Combined quality and speed results for Clip B.

Observations

These graphs do not reveal much new information. However, they do provided a much clearer view of the observations made in the previous sections. Particularly we can see the similarity in encoding times for the two different clips and the jump in time caused by toggling tone masking. Based on the SNR results, we can also see an approximate ordering in terms of precision vs. quality.

8.5 Completeness of the Implementation

So far the tests presented have only been targeting low input sampling rates and monotone audio as this was the target input quality for optimisations. However it would also be re-assuring to know that the modifications made to the encoder have not hindered its functionality when processing samples outside of this range.

As a simple test of the completeness of the encoder, audio clip A was encoded at its originally sampled rate of 44.1KHz with a Vorbis quality setting of 1.0 . I found that in this mode, the hardware accelerated functions were not used at all, but the clip was encoded correctly with the software only versions of functions automatically used as a fall back. The time taken to encode this clip when running the hardware at 12.5MHz was roughly 5 hours.

8.6 Hardware Characteristics

In addition to the performance and quality trade-offs, I also considered the trade-offs in hardware resources. In this project I have primarily been concerned with hardware size and maximum clock frequency.

Table 8.11 shows approximate hardware sizes in terms of FPGA resource utilisation.

Unfortunately during development I encountered many problems when trying to determine the maximum safe clock rate that could be achieved by the various hardware designs. The main problem seemed to be that Handel C generated too many constraints for the Xilinx synthesis tools to handle. This caused many of them to be ignored and even then there were other problems, for example the 'place and route' process failing due to insufficient memory despite running on a machine with 2 Gb of RAM.

Hardware	Virtex-II slices	RAM Blocks	Multipliers
No Acceleration	14014	42	17
A5	16903	57	17
A10	17835	57	17
A14	18738	57	17
A23	20924	57	17
B5	16776	57	20
B10	17629	57	20
B14	18304	57	20

Table 8.11: FPGA hardware resource utilisation.

However I was able to ascertain that without the custom acceleration hardware, a maximum clock frequency of approximately 27MHz was achievable. With custom acceleration hardware using the modified floating-point units with 10bits of precision for the mantissas, 25MHz was achievable.

It was for this reason the hardware designs used in these tests were synthesised without timing constraints applied, and run at what I assumed to be a safe, low-speed of 12.5MHz.

Observations

- For all the different hardware variations the bulk of the logic resources are clearly consumed by Leon, the GRFPU, the extensible framework, and the RC2000 support hardware. For a mantissa size of 10 bits with the modified Celoxica floating point libraries we see that the acceleration hardware causes a roughly 20% overall size increase compared to system with no acceleration hardware. Even with a hardware mantissa size of 23 bits, there is still only a 33% increase total logic resource usage.
- If we combine the limited timing results collected with the results from the speed tests, it is possible to predict that when running at 25MHz, with a mantissa size of 10 bits, and with tone masking disabled the encoding system should be able to process data in real-time with an input sampling rate of 8000Hz in monotone.

8.7 Summary

In this chapter I presented tests for an audio encoding SoC built with my extensible hardware framework based on the Ogg Vorbis algorithm. Primarily I discussed the quality/speed/hardware trade-offs associated with varying the floating point implementation used, as this provided the main way of controlling quality.

From my observations I believe I can summarise the three most most important points as:

- With 10bits of precision in the mantissa, audio output quality had minimal degradation over a software only produced version. The hardware area increase associated with this precision level compared to a design with no hardware acceleration was about 20%.
- The overall effect of turning off tone masking and using acceleration hardware provided on average a 33% increase in speed at the expense of an increase in file size output of about 16%.
- From the test results, it would seem that the hardware produced is quite capable of encoding low rates of input in real-time, and if time is not an issue then any input quality can be encoded.

Chapter 9

Conclusion

I began this project with two main aims:

1. Create an extensible framework that can be used for the investigation of 'System-on-a-Chip' based implementations of audio encoders.
2. Use this framework to produce a working implementation of a digital audio encoder, by applying it to the Ogg Vorbis algorithm. This should be optimised to achieving real-time processing rates at as high quality possible using prototyping hardware available for this project.

In this chapter I shall evaluate how well these two goals have been met in Sections 9.1 and 9.2, present a final review of this project in Section 9.4 and suggest possibilities for further work in Section 9.5.

9.1 Evaluating My Hardware Framework

The first goal of this project was to produce an extensible framework that could be the basis for an application specific 'System-on-a-Chip'. The idea being that this could be used to investigate accelerating audio encoding applications using the technique of software and hardware co-design. In Chapter 3, I gave a more detailed description of exactly what this intended to achieve, including a concise list of requirements. These were as follows:

1. The architecture should be independent of any base sequential processor used. It would be preferable if the framework could completely insulate the extension units from the underlying base SoC/CPU.

2. Adding or removing custom instructions or data processors should be simple, i.e. require few steps with limited distributed alterations.

3. Custom data processors should have a simple and fast interface for accessing shared system RAM.

4. Custom instruction interfaces should be as flexible as possible, with variable input and output parameter numbers and sizes.

5. There should be support for real number processing in hardware, and this should be modular in nature to allow exploration of different real number representations.

In Chapters 4 and 6, I attempted to realise these by implementing a framework based around Leon, written in the Handel C hardware description language.

Of these requirements those pertaining to adding custom data processors have been most successfully met. As described, in order to add a data processor, all that needs to be done is to add an entry in an array, fulfilling requirement 2. Fast access to RAM is provided by the custom memory controller added to Leon which allows uninterrupted direct and parallel access to available RAM banks, fulfilling requirement 3.

This design however is still not perfect, and there are two main limitations. Firstly due to the mutually exclusive RAM access required by this design, the main processor is effectively halted during data processor operation. Fortunately this is not much of a problem for implementing processing stages of an audio encoder in hardware, when they are sequential in nature. For Leon this problem could also have been avoided if necessary by using the interface developed for the high speed AHB bus instead of the bespoke memory access system. It may even be possible to turn the AHB interface into a more abstract interface suitable for accessing other type of buses found in other soft-processors, although this was not attempted in this project.

The second limitation is that the amount of RAM that can be accessed in parallel is defined by the number of RAM banks present. As such, execution on a different hardware system may require modifications to points which access RAM in my current design. This can however be mitigated with careful design of data processors where the RAM is accessed so that parallelism can be easily varied.

Apart from this second limitation, the data processors themselves are not tied to the underlying SoC architecture in any way. In theory, by adapting

the framework to handle a different base CPU, for example Xilinx MicroBlaze, it should be possible to re-use data processing code with no modifications. Thus fulfilling requirement 1.

Although an attempt was made to create a general custom instructions interface, not much use was made of it in this project. Despite this the extension point developed allows a fairly general form of instruction within the constraints of having a maximum of two 64bit input parameters and one 64bit return value. Adding a custom instruction is also fairly easy, again only requiring an entry into an array. This implementation quite neatly satisfies requirements 1 and 2, although the satisfaction of requirement 4 is limited.

The main reason for not making use of custom instructions when implementing the Ogg Vorbis encoder, was although several repeated forms of arithmetic operation could be seen in parts of the source code, it was difficult to see how providing custom instructions for these would give much advantage over the high speed GRFPU. This was particularly due to a rather restrictive co-processor interface provided by Leon, which only allows operations to take place on a segregated set of registers. This would have made sharing data between the co-processor and FPU slow, as all transfers would have to occur via system RAM.

In order to fulfill the need for real numbers support, I made use of the premade Celoxica floating point libraries. In their original form, these would have limited the maximum clock rate of the overall hardware system. Therefore an attempt was made to improve the maximum clock rate attainable by utilising FPGA hardwired multipliers in the multiplication units, and effectively pipelining of the operation of the addition unit. From the results of testing for speed, the slow down caused by these functions seems to cause a fairly negligible overall change in speed when used in the context of the full Ogg Vorbis encode process.

Unfortunately due to the problems ascertaining the maximum clock speed of the hardware design mentioned in the testing chapter, it is difficult to say how much this allowed the maximum clock speed to be improved in the context of the full encoder hardware. However it should be noted that whereas it was possible to obtain a clock speed estimate using the modified libraries, the Xilinx tools were unable to provide a value when using the original libraries due to excessive memory usage. They do also cause a clear hardware resource usage decrease.

I also made attempts to use fixed point real number units. However for this project they proved ineffective due to the large dynamic range required by the specific acceleration hardware produced. But what They do demonstrate

that the real number interface is reasonably modular. They may also become useful when attempting to accelerate other portions of the Ogg Vorbis, or other encoders which do no require such large dynamic number ranges.

9.2 Evaluating the Ogg Vorbis Encoder SoC Implementation

The second goal of this project was to use the framework created to produce a self-contained Ogg Vorbis system on a chip encoder. At the most basic level this has been successfully fulfilled.

The results of testing from the previous chapter show that my final implementation is capable of producing correct Ogg Vorbis streams from PCM input on the RC2000 FPGA prototyping board. This is even capable of encoding in real time, when the input sampling rate is 8000Hz in monotone, tone-masking is disabled, and running at 25MHz. When not attempting to run in real-time, encoding using the full capabilities of Ogg Vorbis is possible.

The resulting system however is probably of not much commercial value as it stands. The acceleration hardware produced typically only allows a 10-12% speed up when tone-masking is disabled. As such it would probably be more cost-effective to use a pre-made CPU/DSP solution running at a high clock speed, which would not require a custom chip to be developed.

The reason for having only achieved a modest speed-up is that the Ogg Vorbis encoding runtime is fairly evenly spread over quite a number of large complex functions. It is infeasible to implement all of these in hardware, as they would likely produce such large designs that it would negate the overall benefits by causing a lower clock speed and requiring a very power-hungry chip. At the same time implementing just a few of them does not justify the speed up gain relative to having to produce a custom chip.

On a more positive note, what the Ogg Vorbis encoder implementation provides is validation that the hardware framework produced can be used to accelerate a software process. In another application, possibly not even audio-related, where there is just one or two dominant run-time consuming functions, it should be relatively straight forward to implement hardware data processors using the same framework.

9.3 An Alternative Audio Encoding Algorithm

The LAME project [35] is another open source audio encoder, which produces output in the MP3 format. A detailed evaluation of this was not performed as part of this project. However it was briefly considered, and a software-only version to run on the RC2000 + Leon was created.

Appendix B shows a call-graph overview of where run-time is spent in LAME when encoding a monotone 8KHz audio file using the lowest possible quality setting. In this mode LAME runs approximately 2.5x faster than Vorbis working in its lowest quality setting. I have estimated that it should be possible to encode 24KHz mono audio with LAME on the same hardware if the MDCT algorithm could be accelerated by a factor of 2x using custom hardware or otherwise.

The primary reason for my not having considered MP3 based encoders in much detail in this project, is that the MP3 standard is covered by patents which require royalty fees [5], whereas Ogg Vorbis is completely free.

9.4 Final Project Review

The goal of this project was to investigate the production of a general framework for creating a self-contained audio encoding system on a chip, and applying it to the Ogg Vorbis lossy audio encoder. In order to do this I began by hypothesising a plan to try and address the high computational element of audio encoders using system on a chip design. From this I produced a list of requirements for an extensible framework that would allow rapid and easy exploration of hardware/software partitions. This was then realised by producing such a framework based around the open source soft-core Leon processor.

Once this framework was completed it was then successfully applied to a reference implementation of Ogg Vorbis allowing me to produce a self-contained audio encoding system. This was then run and tested on the RC2000 FPGA-based prototyping board. The resulting hardware is able to encode raw PCM audio data at real-time speeds using low quality inputs, and when not functioning in real-time is able to encode input of any quality.

Now this work is complete, I believe I can identify 5 distinct achievements:

1. Creating the outline for an extensible framework that can be used for creating application specific SoCs.

2. Producing an implementation of such a framework based around the Leon soft-core processor.
3. Getting this system to run on the RC2000 FPGA-prototyping board using a combination of hardware and software to overcome the difficulties of booting and debugging software executing in this environment.
4. Investigating hardware designs for using real numbers including both floating point and fixed point implementations.
5. Producing an accelerated version of the Ogg Vorbis encoder using the real numbers and extensible framework hardware created.

In my evaluation I suggested that the Ogg Vorbis based system on a chip audio encoder, was probably of limited value on its own due to only a modest speed-up achieved. However the designs produced to facilitate this implementation were validated by this case study, and due to their highly generic nature, should have use in other applications for example the LAME MP3 decoder.

9.5 Further Work

Here is a brief list of items, that I feel may be areas of interest for further work.

- **Investigate implementing the Vorbis decoder using my framework** – It would interesting to see how much easier it is to investigate the hardware/software partition using Handel C rather than VHDL as was used in the 'Ogg-on-a-chip' project. It may also be possible to better their performance increase with the same hardware partition by taking advantage of the higher memory bandwidth provided by my custom interface. It was specifically suggested in Section 4.2.2.3 of the 'Ogg-on-a-chip' report that memory access speed was one of their limiting factors.
- **Investigate running other audio encoders/decoders using my framework** – The tools and framework created in this project are all of a highly generic nature. It would be good to find a better example to demonstrate their effectiveness than the Ogg Vorbis encoder. One particular area of interest might be producing an accelerated version of the [I]MDCT transforms. This could then be used in many lossy audio CODEC algorithms, all built to run on the same hardware allowing it to be both a recorder and a player.

Another possibility is to create an adaptable audio encoder, which uses different algorithms depending on the type of input data and output quality desired. Xiph.Org hosts two other audio CODECs: Speex which is specifically optimised for compressing speech, and FLAC which performs lossless compression of audio data. By combining these with Ogg Vorbis, on a single embedded audio encoder a user could choose how most effectively to use the storage media available to them. It may even be possible to create an embedded system based around an FPGA, which can reconfigure the acceleration hardware available appropriately for the selected algorithm.

- **Evaluating the power consumption characteristics of hardware acceleration using this framework** – Power consumption is another hardware characteristic along with size and speed, which has not been discussed in this work. This is an area that needs to be considered before implementations made using an extensible SoC framework can be considered viable for use in real portable embedded systems.

- **Modify the Leon memory controller to support parallel RAM bank access** – Currently the Leon memory controller does not deliver data at the maximum rate allowable by the AHB bus. This is due to the delays involved when using sequential access to RAM banks. It might be possible to improve this using a striped memory layout as was used in my custom memory controller.

- **Create an abstract bus access system which can be used by the framework** – For applications where it is not desirable to block CPU access when a data processor is active, an interface to a traditional arbitrated bus would be more advantageous. This could for example seamlessly allow access to the AHB in Leon or CoreConnect when using a Xilinx MicroBlaze or PowerPC processor, without having to change any code in a data processor implementation. It may even be possible to still achieve a high bandwidth by simply widening the bus.

- **Generalise the framework to work with other soft-core processors** – Doing this would allow not only the investigation of custom hardware to accelerate an application, but also different CPUs. This would allow observations of trade-offs in different sequential processor designs, without having to modify custom hardware created.

- **Improve the custom instruction addition infrastructure in the framework** – The custom instruction extension point developed was

heavily driven by the co-processor interface provided by Leon. By taking a wider look at the custom instruction extension features of other processors, it might be possible to create a more portable method for creating custom instructions.

If this proves successful, it might then be possible to combine this with a compiler which can also automatically infer and generate custom instructions given a program listing. An example of a compiler which can already perform a similar task for a specifically made CPU can be found in [34].

Appendix A

Floating-Point Refresher

Lest the specifics of floating point numbers not be at the forefront of the reader's mind, here is a quick refresher.

Floating point number representation is one of the most widespread ways of representing fractional numbers in digital systems. Numbers are represented in three binary parts: a one bit sign, an exponent which represents the magnitude of a number as a power of two, and a mantissa which represents and constrains the dynamic range of numbers. This can be expressed mathematically as:

$$(-1)^{sign} * mantissa * 2^{exponent}$$

When I refer to floating point numbers in this report, I am specifically referring to the IEEE 754 form of floating point. This defines that numbers are stored in a normalised form, where the mantissa is represented in fixed-point binary with a one bit integer part and the most significant bit of the mantissa is always one. As the MSB of the mantissa is always one it is considered to be implicit and not actually stored. It also stipulates that the exponents are to be stored as a 'biased' value, i.e. the stored value has a constant offset from the actual value.

IEEE 754 specifies several different classes of accuracy commonly used to store floating point numbers. The Ogg Vorbis encoder mostly uses the 'single precision' accuracy format where 8 bits are used for the exponent with a bias of -127, and 23 bits are used for the mantissa. Furthermore there are special reserved bit patterns that can be used to represent floating point exception values for example Infinity (INF) and Not-a-Number (NaN).

Appendix B

LAME MP3 Encoder Callgraph

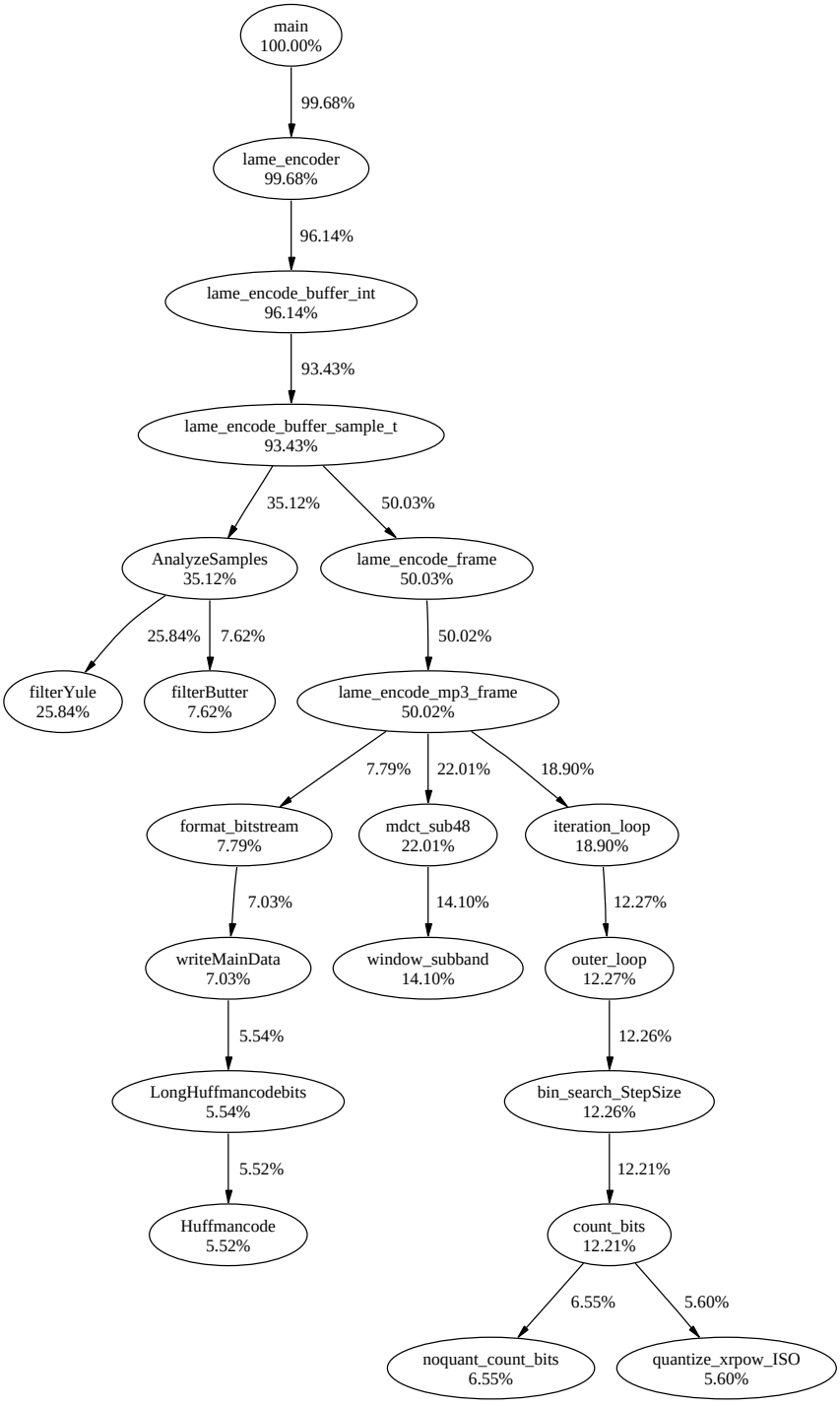

Figure B.1: Callgraph from LAME using a low quality setting on an 8000Hz mono sample.

Appendix C

Floating Point Adder Data-flow

In order to partition the Celoxica floating point adder into two sequential cycles, I hacked together Perl script to generate a rough data-flow/dependency graph. Using this I could determine which signals needed to be buffered across cycles. This would have been quite difficult to do without aid, due to the shear number of variables as the graph shows.

The graph is shown in Figure C.1. Nodes represent variable signals, and arrows indicates that computation of a variable depends on the result of the variable pointed to.

Using this graph, I chose to make the division by moving the computation of: `addmanttoflow`, `delmosubmant` and `eelmosubmant` to the second cycle. This may look a slightly unbalanced division, however there is also another part of the function implemented in the second cycle which is not represented on the graph.

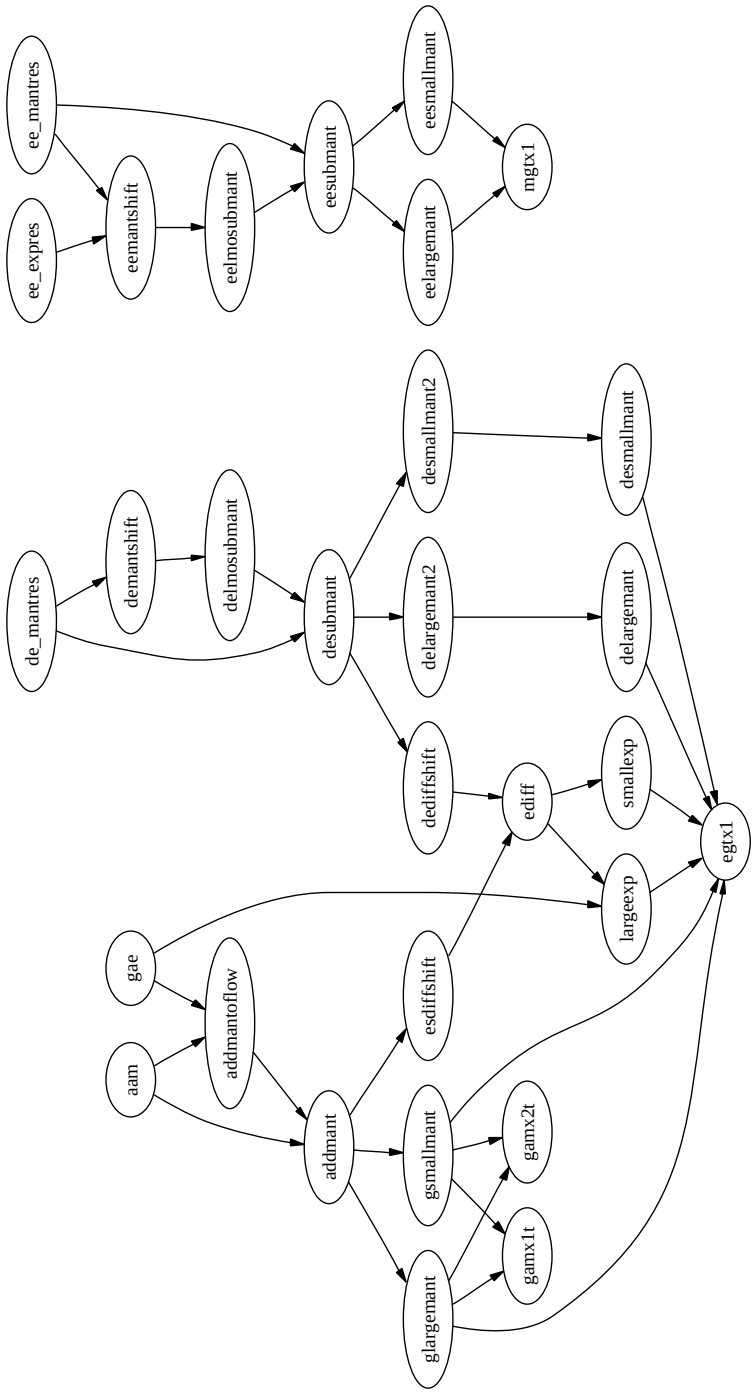

Figure C.1: Rough data dependencies in part of the Celoxica Floating Point adder.

Appendix D

Control Client Command Line Options

Command line options for the RC2000 + Leon control programme.

Parameter	Use
`--bitfile filename`	FPGA Bitfile containing LEON.
`--indata filename`	Input data file.
`--outdata filename`	Output data file.
`--freq int`	Frequency to run design in Hz (default = 12000000).
`--card int`	FPGA card index (default = 0).
`--help`	Display command line options help.
`--srec filename`	S-Record file to load program image from.
`--progressmode`	Disables text output and displays a progress bar instead. Progress is calculated based on the amount of the input data file read.
`--rate int`	Specifies the input PCM sampling rate in Hz (default = 8000).
`--channels int`	Specifies the number of audio channels in the input PCM data (default = 1).
`--quality int`	Specifies the quality of audio encoding as an integer 0-100 (default = 50).
`--accel`	If this option is supplied then hardware acceleration is used in the encoding process.
`--tonemasking`	If this option is supplied then tonemasking is enabled.

Appendix E

_vp_noisemask annotated source code

```
void _vp_noisemask(vorbis_look_psy *p,
                   float *logmdct,
                   float *logmask){

  int i,n=p->n;
  float *work=alloca(n*sizeof(*work));

  /* First call to bark_noise_hybridmp */
  bark_noise_hybridmp(n,p->bark,logmdct,logmask,
                      140.,-1);

  /* Loop 1.1 */
  for(i=0;i<n;i++)work[i]=logmdct[i]-logmask[i];

  /* Second call to bark_noise_hybridmp */
  bark_noise_hybridmp(n,p->bark,work,logmask,0.,
                      p->vi->noisewindowfixed);

  /* Loop 1.2 */
  for(i=0;i<n;i++)work[i]=logmdct[i]-work[i];

  /* Loop 1.3 */
  for(i=0;i<n;i++){
    int dB=logmask[i]+.5;
    if(dB>=NOISE_COMPAND_LEVELS)dB=NOISE_COMPAND_LEVELS ↩
        -1;
    if(dB<0)dB=0;
    logmask[i]= work[i]+p->vi->noisecompand[dB];
  }
```

```
}
```

Appendix F

bark_noise_hybridmp annotated source code

```
/* n is the size of data to be processed
 * b is the lookup table from the diagam
 * f is the input spectrum
 * nosie is the output mask
 */
static void bark_noise_hybridmp(int n,const long *b,
                                const float *f,
                                float *noise,
                                const float offset,
                                const int fixed){

/* These 5 seperate arrays are grouped together as one
 * temporary array in the diagram
 */
  float *N=alloca(n*sizeof(*N));
  float *X=alloca(n*sizeof(*N));
  float *XX=alloca(n*sizeof(*N));
  float *Y=alloca(n*sizeof(*N));
  float *XY=alloca(n*sizeof(*N));

  float tN, tX, tXX, tY, tXY;
  int i;

  int lo, hi;
  float R, A, B, D;
  float w, x, y;

  tN = tX = tXX = tY = tXY = 0.f;

  /* Initialisation of Loop 2.1 */
```

```
y = f[0] + offset;
if (y < 1.f) y = 1.f;

w = y * y * .5;

tN += w;
tX += w;
tY += w * y;

N[0] = tN;
X[0] = tX;
XX[0] = tXX;
Y[0] = tY;
XY[0] = tXY;

/* Loop 2.1 */
for (i = 1, x = 1.f; i < n; i++, x += 1.f) {

  y = f[i] + offset;
  if (y < 1.f) y = 1.f;

  w = y * y;

  tN += w;
  tX += w * x;
  tXX += w * x * x;
  tY += w * y;
  tXY += w * x * y;

  N[i] = tN;
  X[i] = tX;
  XX[i] = tXX;
  Y[i] = tY;
  XY[i] = tXY;
}

/* Loop 2.2 */
for (i = 0, x = 0.f;; i++, x += 1.f) {

  lo = b[i] >> 16;
  if( lo>=0 ) break;
  hi = b[i] & 0xffff;

  tN = N[hi] + N[-lo];
  tX = X[hi] - X[-lo];
  tXX = XX[hi] + XX[-lo];
  tY = Y[hi] + Y[-lo];
  tXY = XY[hi] - XY[-lo];
```

```
  A = tY * tXX - tX * tXY;
  B = tN * tXY - tX * tY;
  D = tN * tXX - tX * tX;
  R = (A + x * B) / D;
  if (R < 0.f)
    R = 0.f;

  noise[i] = R - offset;
}

/* Loop 2.3 */
for ( ;; i++, x += 1.f) {

  lo = b[i] >> 16;
  hi = b[i] & 0xffff;
  if(hi>=n)break;

  tN = N[hi] - N[lo];
  tX = X[hi] - X[lo];
  tXX = XX[hi] - XX[lo];
  tY = Y[hi] - Y[lo];
  tXY = XY[hi] - XY[lo];

  A = tY * tXX - tX * tXY;
  B = tN * tXY - tX * tY;
  D = tN * tXX - tX * tX;
  R = (A + x * B) / D;
  if (R < 0.f) R = 0.f;

  noise[i] = R - offset;
}

/* Loop 2.4 */
for ( ; i < n; i++, x += 1.f) {

  R = (A + x * B) / D;
  if (R < 0.f) R = 0.f;

  noise[i] = R - offset;
}

/* This test always results in an exit
 * when using low input data sampling rates.
 */
if (fixed <= 0) return;


/*
 * The following loops are not implemented in hardware.
```

```
    * But as we can see their similarity to the previous
    * loops would make them quite easy to implement as
    * well
    */

  for (i = 0, x = 0.f;; i++, x += 1.f) {
    hi = i + fixed / 2;
    lo = hi - fixed;
    if(lo>=0)break;

    tN = N[hi] + N[-lo];
    tX = X[hi] - X[-lo];
    tXX = XX[hi] + XX[-lo];
    tY = Y[hi] + Y[-lo];
    tXY = XY[hi] - XY[-lo];


    A = tY * tXX - tX * tXY;
    B = tN * tXY - tX * tY;
    D = tN * tXX - tX * tX;
    R = (A + x * B) / D;

    if (R - offset < noise[i]) noise[i] = R - offset;
  }
  for ( ;; i++, x += 1.f) {

    hi = i + fixed / 2;
    lo = hi - fixed;
    if(hi>=n)break;

    tN = N[hi] - N[lo];
    tX = X[hi] - X[lo];
    tXX = XX[hi] - XX[lo];
    tY = Y[hi] - Y[lo];
    tXY = XY[hi] - XY[lo];

    A = tY * tXX - tX * tXY;
    B = tN * tXY - tX * tY;
    D = tN * tXX - tX * tX;
    R = (A + x * B) / D;

    if (R - offset < noise[i]) noise[i] = R - offset;
  }
  for ( ; i < n; i++, x += 1.f) {
    R = (A + x * B) / D;
    if (R - offset < noise[i]) noise[i] = R - offset;
  }
}
```

Appendix G

Obtaining the Project Source Code

The source code to the hardware and software designs created as part of this project can be found at the following web address:

http://www.doc.ic.ac.uk/∼jab00/project/

Appendix H

Glossary

AMBA bus	'Advanced Microcontroler Bus Architecture', a bus standard created by ARM.
AHB	'Advanced High-speed Bus', part of the AMBA bus standard.
APB	'Advanced Peripheral Bus', a low-speed bus, part of the AMBA bus standard.
Audio encoder	A system for capturing and representing analogue audio signals in a digital format suitable for storage and/or distribution.
Custom Instruction	A basic programmable operation that can be performed by a software CPU which targets optimising a specific application.
CODEC	A COding DECoding algorithm, see also **Audio encoder**.
Data-path	When used in the context of hardware, refers to circuit linking together registers using combinatorial logic to perform a task.
(Hardware) Data processor	A specific instance of a **data-path** which performs sequential data processing.

Fixed point	A binary form of fractional numbers with a fixed position radix.
Floating point	See Appendix A.
FPGA	'Field-Programmable Gate Array', A re-programmable hardware logic device. See also **LUT**, **Slices**, and **Virtex**.
GRFPU	'Gaisler Research Floating Point Unit', a high-speed floating point unit for use with Leon.
Handel C	A hardware description language made by Celoxica.
Leon	An open-source soft-core CPU produced by Gaisler Research implementing the SPARC v8 standard. See also **soft-core processor**.
LUT	'Look-Up Table', a type of FPGA resource for performing combinatorial logic. See also **FPGA**.
Macro instruction	A specific form of a custom instruction which is formed by merging together several other processor instructions/ operations, and running them in parallel. See also **Custom Instruction**.
MDCT	'Modified Discrete Cosine Transform', a mathematical transformation converting a signal from the time domain into the frequency domain.
Noise masking	Part of the psychoacoustic audio profiler used in Ogg Vorbis. The specifics of its operation are not well defined.
Soft-core processor	A hardware CPU supplied in an abstract or source code form.
System on a Chip (SoC)	A highly integrated logic device, often containing the majority of components found in a complete computing system.
Tone masking	Part of the psychoacoustic audio profiler used in Ogg Vorbis. The specifics of its operation are not well defined.

Slices	A division of logic resources used by the Virtex range of FPGAs. See also **Virtex** and **LUT**.
VHDL	'Very large scale integration Hardware Description Language', a language that can be used to describe and model digital hardware.
Virtex	A brand of FPGA produced by Xilinx. See also **FPGA**.

Bibliography

[1] M. J. Geier - *Movies on the move*, IEEE Spectrum, pg 57-58, June 2004.

[2] Xiph.Org Foundation - *Vorbis.com - Open, Free, Audio.*, http://www.vorbis.com/, 2004.

[3] Xiph.Org Foundation - *List of Vorbis hardware from the Xiph Wiki*, http://wiki.xiph.org/VorbisHardware/, 2004.

[4] L. Azuara, P Kiatisevi - *Ogg-on-a-Chip Project*, http://oggonachip.sourceforge.net/, 15^{th} July 2002.

[5] Thomson - *mp3licensing.com*, http://www.mp3licensing.com/, 2004.

[6] Wikipedia - *Modified discrete cosine transform*, http://en.wikipedia.org/wiki/Modified_discrete_cosine_transform, 2004.

[7] M. F. Davis - *The AC-3 Multichannel Coder*, http://www.dolby.com/tech/ac-3mult.html, October 1993.

[8] K. Tsutsui, H. Suzuki, O. Shimoyoshi, M. Sonohara K. Akagiri, R. M. Heddle - *ATRAC: Adaptive Transform Acoustic Coding for MiniDisc*, http://www.minidisc.org/aes_atrac.html, October 1992.

[9] Xiph.Org Foundation - *Theora.Org*, http://www.theora.org/, 2004.

[10] Open Source Initiative OSI - *The BSD License*, http://www.opensource.org/licenses/bsd-license.php, 2004.

[11] Xiph.Org Foundation - *Games That Use Vorbis*, http://wiki.xiph.org/GamesThatUseVorbis, 2004.

[12] Xiph.Org Foundation - *Vorbis I Specification*, http://www.xiph.org/ogg/vorbis/doc/Vorbis_I_spec.html, 2003.

[13] JCraft - *JOrbis - Pure Java ogg Vorbis decoder*, http://www.jcraft.com/jorbis/, 2004.

[14] M. Coleman - *Vorbis Illuminated*, http://www.mathdogs.com/vorbis-illuminated/, 2001.

[15] Computer50.org - *50th Anniversisary of the Manchester Baby computer*, http://www.computer50.org, 2004.

[16] Wikipedia - *Intel 4004*, http://en2.wikipedia.org/wiki/Inteli_4004, 2004.

[17] Apple - *Apple - iPod*, http://www.apple.com/ipod/, 2004.

[18] PortalPlayer Inc. - *PP5002 Digital Media Management System-on-a-chip*, http://www.portalplayer.com/products/PP5002_Brief.pdf, 2003.

[19] Celoxica - *DK Design Suite*, http://www.Celoxica.com/products/tools/dk.asp, 2004.

[20] Celoxica - *RC200 Development Board*, http://www.celoxica.com/products/boards/rc200.asp, 2004.

[21] Celoxica - *RC2000 Development Board*, http://www.celoxica.com/products/boards/rc2000.asp, 2004.

[22] D. Driessens, T. Tierens - *Overview of Embedded Processors used in Programmable SOC*, http://emsys.denayer.wenk.be/empro/Overview%20of%20Embedded%20Processors.pdf, 2003.

[23] Gaisler Research - *LEON2 Processor*, http://www.gaisler.com/leonmain.html, 2004.

[24] Yahoo! Groups - *Leon Sparc*, http://groups.yahoo.com/group/leon_sparc/, 2004.

[25] Sparc International Inc. - *SPARC Architecture V8*, http://www.sparc.com/standards/V8.pdf, 2004.

[26] ARM Ltd. - *AMBA Specification Rev 2.0*, http://www.arm.com/products/solutions/AMBA_Spec.html, 1999.

[27] Prof. G. Z. Yang - *Multimedia Lecture Notes - Sound and Audio*, 2004.

[28] K Projects - *Valgrind*, http://valgrind.kde.org/, 2004.

[29] K Project - *KCachegrind and Calltree*, http://kcachegrind.sf.net/, 2004.

[30] E. Montnemery, J. Sandvall - *Ogg/Vorbis in embedded systems*, http://www.sandvall.nu/thesis.pdf, February 2004.

[31] Motorola - *The S-record file format*, http://www.wotsit.org/download.asp?f=srec, March 1984.

[32] Alpha Data - *ADM-XRC-II User Manual*, http://www.alpha-data.com/pdf/ADM-XRC-II%20User%20Manual.pdf, 2004.

[33] W. Chu, R. Dimond and W. Luk - *Customising EPIC processor: architecture and tools*, *Proc. Design, Automation and Test Europe (DATE'04)*, IEEE, 2004, http://www.doc.ic.ac.uk/∼wl/papers/date04chu.pdf, 2004.

[34] R. Dimond - *Compilation and Simulation Support for Custom Instruction Processors*, http://www.doc.ic.ac.uk/∼rgd00/essay_submit.pdf, 2003.

[35] LAME - *The 'Lame Ain't an MP3 Encoder' Project*, http://lame.sourceforge.net/, 2004.