HOGESCHOOL GENT
FACULTY OF INDUSTRIAL SCIENCES
DEPARTMENT OF ELECTRONICS

Academic year of 2005-2006

PSYCHOACOUSTIC AUDIO CODING ON SMALL ARM CPUs

Gian-Carlo PASCUTTO

Supervisor:
Ing. P. Van Torre (Hogeschool Gent)

A thesis presented in fulfillment of the requirements for the degree of
INDUSTRIEEL INGENIEUR ELEKTRONICA OPTIE ONTWERPTECHNIEKEN

Abstract

Psychoacoustic audio codecs have become the tool of choice for many embedded, portable audio applications. Likewise, the ARM processor architecture has become common in applications where high performance and low power consumption are required.

Although currently available codecs provide high performance, they generally require a high implementation effort, and have large RAM and ROM requirements, which makes them unsuitable for smaller embedded systems.

We take a look the peculiarities and capabilities of the ARM architecture, currently employed ARM implementations, and study the current state of the art in general audio coding. This allows us to construct a new audio codec with high performance, minimal processing requirements and a low implementation complexity, which is suitable for usage on small embedded ARM CPUs.

We design a reference playback platform and demonstrate that the proposed solution is a practical replacement for existing codec designs.

Acknowledgments

This work was made possible due to the contributions and help of several people. First, I have to thank Mr. Patrick Van Torre, who proposed the subject, was receptive to suggested changes, and provided a lot of suggestions and corrections when they were most dearly needed, as well as providing motivating positive feedback. Likewise, Mr. Luc Colman proved to be a great help by reviewing several proposed PCB designs, and contributed directly to the hardest parts of getting the final design soldered together. He also acted as an excellent guide in the never-ending quest of locating the right tools and equipment in Hogeschool Gent's electronics labs. The technical personnel deserves mention as well: they did an amazing job in producing a working PCB even when faced with very strict specifications.

Sebastian Gesemann from the University of Paderborn corrected various misunderstandings in our knowledge of the audio coding field, and contributed several ideas and suggestions which we were able to make good use of. Ivan Dimkovic and Menno Bakker from Nero A.G. assisted in locating hard-to-find papers and provided many hints as to what approaches would be most fruitful. Mr. Peter Pawlowski contributed a high-quality library for the handling of WAV files, and made a plugin for the foobar2000 audio player, which significantly aided our quality testing. Mr. Christopher Montgomery took the time to personally explain several intricate details of the Vorbis algorithms, and his free publication of the Vorbis codec was what originally drew me to the audio coding field.

Lastly, my girlfriend, family and friends had to endure a full year of middle-of-the-night audio quality testing, keyboard hammering and general disinterest in anything that did not involve time-frequency trade-offs, and nevertheless still kept encouraging me. For that, I thank them.

Contents

1 The ARM architecture **7**

1.1 Introduction 7
1.2 Overview 7
1.2.1 History and origin 7
1.2.2 Evolution of the ARM architecture 9
1.2.3 Future development 11
1.3 ARM peculiarities 13
1.4 The ARM instruction set architecture 14
1.4.1 Register file 14
1.4.2 Processor modes 15
1.4.3 Current Program Status Register 15
1.4.4 Conditional Execution 16
1.4.5 Barrel shifter 16
1.4.6 Loading and Storing 17
1.4.7 Branching and subroutines 18
1.5 DSP capabilities 18
1.5.1 Fixed point arithmetic and load-store 18
1.5.2 ARM7TDMI 19
1.5.3 ARM9TDMI 19
1.5.4 ARM9E 19
1.5.5 ARM10, ARM11 and later 19
1.5.6 XScale 20
1.5.7 DSP-specific extensions 20
1.6 Coding for the ARM 20
1.7 ARMs with DSPs 21
1.8 Conclusions & outlook 22

2 Psychoacoustic audio coding **23**

2.1 Introduction 23
2.2 The human auditory system 23
2.3 Psychoacoustic principles 24
2.3.1 Critical bands & the Bark scale 24
2.3.2 Simultaneous masking & tonality 26

2.3.3 Absolute Threshold of Hearing 27
2.3.4 Spreading 27
2.3.5 Temporal effects 27
2.3.6 Noise and noise 28
2.3.7 Binaural effects 28
2.4 Toward the frequency domain 28
2.4.1 Quadrature Mirror Filters 28
2.4.2 Subbanding 29
2.4.3 Wavelet filterbanks 29
2.4.4 FFT 30
2.4.5 Modified discrete cosine transform 31
2.5 Quantization 32
2.5.1 Normalization 32
2.5.2 Linear quantization 32
2.5.3 Power-law quantization 32
2.6 Lossless coding 33
2.6.1 Direct coding 33
2.6.2 Huffman coding 33
2.6.3 Arithmetic coding 33
2.6.4 Vector Quantization 33
2.7 Stereo coding 34
2.8 Binaural Cue Coding 34
2.9 Practical audio codecs 34
2.9.1 MPEG 1 Layer 1 & 2 34
2.9.2 MPEG 1 Layer 3 35
2.9.3 MPEG 2 & 4 Advanced Audio Coding 36
2.9.4 Ogg Vorbis 37
2.9.5 Wavelet Audio Codecs 38
2.10 Quality evaluation 39
2.11 Conclusions & outlook 40

3 Bacon: a low-complexity audio codec 41
3.1 Design considerations 41
3.2 Transformation 42
3.2.1 Avoiding block switching 42
3.2.2 The Modified Discrete Wavelet(like) Transform 42
3.2.3 Transform switching 45
3.2.4 Temporal Noise Shaping 53
3.2.5 Fast MDCT implementation 54
3.2.6 Spectrum partitioning 56
3.3 Psychoacoustic model 56
3.4 Quantization 59
3.4.1 Mide/Side Coding 59
3.4.2 Quantizer 59

3.4.3 The 2-loop system 60
3.4.4 Scalefactor Estimation 61
3.5 Lossless coding 61
3.5.1 Differential encoding 61
3.5.2 Huffman encoding 62
3.6 Additional tools 62
3.6.1 PNS 62
3.6.2 LastVal 63
3.7 Fixed point version 63
3.7.1 Number classes 63
3.7.2 Smart multiplying 64
3.8 Performance evaluation 65
3.8.1 Processing requirements 65
3.8.2 Memory requirements 67
3.8.3 Quality 68
3.9 Conclusions & future research 70

4 An ARM-based Bacon audio player 72
4.1 The high end - TI OMAP5912 OSK 72
4.1.1 About the OSK 72
4.1.2 The ARM Linux kernel 73
4.1.3 Host software requirements 73
4.1.4 Targeting the OSK 74
4.1.5 Buildroot 74
4.1.6 Audio output with OSS 75
4.1.7 Some remarks on difficulty 75
4.2 The low end - Atmel AT91SAM7S 75
4.2.1 Choosing a CPU 76
4.2.2 Getting the input 77
4.2.3 Playing the output 77
4.2.4 Programming the ARM 78
4.2.5 Circuit design 79
4.2.6 PCB design 80
4.2.7 Finishing & testing the PCB 83
4.3 Conclusions & evaluation 84

List of Figures

3.1 Transition band requirements for QMF filters 45
3.2 Filter properties of Johnston 64 tap QMF 46
3.3 Filter properties of Yu 32 tap QMF 47
3.4 Filter properties of Johnston 16 tap QMF 48
3.5 Filter properties of 6 tap 1.3 bi-orthogonal wavelet 49
3.6 c44.wav (Castanet impulses sound sample) 50
3.7 c44.wav encoded with a 2048 point MDCT 51
3.8 c44.wav encoded with a 2048 point MDWT 52
3.9 c44.wav encoded with a 2048 point MDCT + 16 tap TNS . . 55
3.10 PEAQ Basic Model quality test results 71

4.1 Wiggler compatible JTAG schematic 78
4.2 AT91SAM7S Bacon player schematic 81
4.3 PCB Front 82
4.4 PCB Back 82
4.5 PCB Soldermask 83

List of Tables

1.1 Condition Codes 16
1.2 Barrel shifter 16
1.3 Indexing Methods 17
1.4 Multiple Load-Store Indexing Methods 17
1.5 MAC timings per ARM core, in cycles 18

2.1 Bark scale (Traunmüller) 25
2.2 NMT/TMN Formulas 26

3.1 ERB Scale 57
3.2 Bacon spectral partitions (SFB's) at 44.1kHz 58
3.3 PC encoder speeds for 128kbps CBR 65
3.4 PC decoder speeds for 128kbps CBR 66
3.5 ARM decoder speeds for 128kbps CBR 67
3.6 ARM decoder memory usage 68
3.7 Quality test samples 69
3.8 PEAQ Basic Model quality test results 70

4.1 The AT91SAM7 range 77

Chapter 1

The ARM architecture

1.1 Introduction

The ARM architecture is a 32-bit advanced RISC design. The use of a simple design and a highly orthogonal, yet powerful instruction set allows it to attain good processing speeds at a low power consumption.

As the demand for greater processing power in embedded devices keeps increasing, so does the popularity of ARM. The architecture is licensed by a large number a microprocessor manufacturers, leading to a wide choice and availability of ARM based CPUs and microcontrollers.

Current ARM based CPUs are powerful enough to allow a large variety of DSP processing tasks to be performed without the use of a coprocessor. We will take a look at both the ARM architecture and its application to DSP tasks.

1.2 Overview

1.2.1 History and origin

The origins of ARM date back to the 1980's and the British computer manufacturer Acorn Computers. Acorn had enjoyed some success in the British market, getting large sales with it's "BBC Micro" computer. The BBC Micro was built by Acorn for the British Broadcasting Corporation's "BBC Computer Literacy Project", with the goal of educating the British population about what was considered the inevitable up-and-coming microcomputer revolution.

The BBC Micro was, for it's time, an advanced machine with good expandability and a rugged design – a necessary feature since it was widely used for teaching computer skills in classrooms. At it's hearth was a 2Mhz MOS Technology 6502A. Over 1 million of these machines were sold for prices ranging from 300 to 400£ [1], allowing Acorn to expand it's business.

Meanwhile, the IBM PC had been launched, and began making headway in the business world. The Apple Lisa had also brought the first windowing system to the market. It was time for Acorn to move away from the 6502 CPU to a more capable alternative. There was however, one problem: Acorn's engineers had tested all 16 and 32-bit processors on the market, and none of them met their requirements. Notably, the Motorola 68000 and National Semiconductor 16032[1] processors were rejected because they had too big interrupt latencies, or were to slow and complex, respectively. Even at higher clock speeds, the very advanced NS 16032 was actually slower than the much simpler 6502.

Acorn decided to do the processor design themselves. They had sufficient cash reserves, meaning the project could take a while if needed, and some quite skilled research staff. What they didn't have was the experience nor the equipment or manpower[2] to tackle a large design [2]. At about the same time, the first formal research papers on RISC architectures started appearing from UC Berkeley and Stanford. This served as an encouragement: if a group of graduate students could come up with competitive new CPU designs, then Acorn surely had a good chance of succeeding.

A team consisting of Steve Furber, Roger Wilson and Robert Heaton set to work on the project, basing themselves on their large experience with writing system software for the 6502 and – from looking at the competing designs – the knowledge of how it should *not* be done.

Work started in 1983 and finished in 1985, a mere 18 months and 5 man-years later. The Acorn RISC Machine 1 (ARM1) consisted of only 25 000 transistors. Much of the required tools were custom written in BASIC, to be run on the BBC Micro. In return, the first ARM prototypes were used as coprocessors for the Micro's, so the design tools for the complete system could finish faster.

The first experiences with writing system software for the ARM pointed to some inefficiencies, and those were resolved quickly in the ARM2. A multiply and multiply-accumulate instruction were added, both instructions which were usually the first to be thrown out in RISC designs. Putting them back in allowed the Acorn to do some real-time signal processing and have minimal sound capabilities, which was estimated to be important for a home and educational computer.

While the design of the CPU for the next-generation Acorn computer was a success, the rest of the system was more problematic. Both the surrounding peripherals as the operating system took large amounts of time to finish, delaying the system by as much as 2 years. Meanwhile, the IBM PC had managed to set itself more or less as the standard, and the ARM's

[1]Later renamed the 32016 to emphasize the 32-bitness.

[2]While Acorn could have hired more engineers or bought more equipment, manager Hermann Hauser famously refused to, giving the engineers "no money and no people".

appearance with a new processor, and new operating system, and no software, was not met with much enthusiasm. Neither were Acorn's in-between systems based on the NS16032, leading to a financial crisis which allowed Olivetti to buy Acorn, mostly to acquire it's share in the UK home computer market.

Acorn continued both the development of the ARM and the systems built on it, gradually allowing more software to appear, and boosting the performance of the ARM processors. The ARM3 added on-chip cache, which boosted the numbers of transistors to 300 000, still a tiny amount compared to other designs.

Some success was made selling the ARM CPU to other companies, for example as an embedded processor. One of the companies to buy the ARM was Apple, which built a new kind of device around it: the Apple Newton Personal Digital Assistant (PDA). The new device was not very successful – at the time.

The collaboration with Apple, as well as the financial situation of Acorn, lead to the splitting off of the Advanced Research and Development section of Acorn into the newly founded Advanced RISC Machines Ltd. ARM Ltd. focused mainly on the design of the processor IP cores. The actual silicon production is handled by the licensees.

While the sales and licensees of ARM Ltd. kept going up, Acorn's value kept going down until, eventually, it's share in ARM Ltd. was worth more than the company itself. It underwent a series of restructurings and repositioned itself as an IP developer in the DSP world, with the new name of Element 14. They later bought up core parts of Alcatel's DSL development, before being acquired by Broadcom.

1.2.2 Evolution of the ARM architecture

A note on naming

Since ARM refers to both the architecture design as well as the processor IP cores implementing it, the naming can be confusion. The architectures are referred to as ARMvX with X being the version, and the cores just as ARMX, with X being the version.

ARMv1, ARMv2 and ARMv2a

Introduced in 1985, the ARM1 was the first incarnation of the ARMv1 architecture, a 32 bit architecture, using 26 bit addressing. The ARMv2 and ARM2 added multiplication and multiply-accumulate capability, as well as coprocessor support.

The ARM3 (using ARMv2a) added a 4k on-chip code and data cache.

ARMv3 – ARM6 and ARM7

ARMv3 moved the ARM architecture toward full 32 bit addressing, and also allowed for chips to optionally contain a Memory Management Unit (MMU), allowing the chip to run full operating systems.

ARMv3M added optional long multiplication support (32 x 32 = 64 bits), marked by an M in the processor type name

ARMv4 — ARM7TDMI-S and Thumb

The ARM4 is a further evolution of the ARM3 design, adding support for 16 bits load and stores, and dropped support for 26-bit addressing. It was introduced in 1995.

A major step forwards in the ARM architecture was made by the introduction of Thumb in ARMv4T. Thumb is a new instruction set for the ARM architecture, using 16 bit instructions. It represents a significant enhancement to the ARM architecture, because it allows many operations to be performed at a significantly reduced code size. It is very useful in situations where bus or memory bandwidth or space is limited. The Thumb instructions are slightly less capable than the ARM counterparts, notably lacking conditional execution, shift/rotation support, and not allowing access to the entire register file. Because of the loss of orthogonality and cleanness, Thumb code is generally not written by hand.

Although Thumb reduces the size of each instruction by half, performing the same operations in Thumb usually requires some more instructions as in ARM mode. On average, one can expect a savings of 30%.

It is possible to mix ARM routines with Thumb routines, allowing the performance critical parts of the code to be written in ARM code, and the rest of the system in Thumb, maximizing performance and minimizing code size.

The ARMv4T architecture is very widespread, because the ARM7TDMI core, which implements it, was widely licensed and adopted both in various microcontrollers as well as portable devices (such as the iPod or the Game Boy Advance). The ARM7TDMI includes wide (M)ultiply and (T)humb decoding, as well as the (D)ebug JTAG and embedded (I)n-circuit emulator features.

The ARM7TDMI core is also available in a (S)ynthesizible version, allowing it to be compiled into other designs.

ARMv5 — ARM9EJS

The ARMv5 makes the optional Thumb and wide multiply features of the ARMv3 the default, and adds the following:

- Some extra instructions for smoother switching between ARM and Thumb states.
- Enhanced DSP support, including faster 16 bit multiplies and saturating arithmetic, and a faster MAC. Marked by an (E) in the chip name.
- Jazelle, allowing the ARM core to execute some Java bytecode instructions in hardware. Jazelle is an essential component in the Java support included in many mobile phones. Marked by a (J) in the chip name.

The ARM9 extends the 3 stage pipeline of the ARM7 to a 5 stage pipeline, allowing higher clock speeds. While most ARM7 cores peak around 80Mhz operation, ARM9 allows clock speeds in excess of 220Mhz. The design also switched from a Von Neumann (shared data and instruction buses) to a Harvard architecture (separate data and instruction buses). It was released in 1997.

The ARM9EJ-S, released in 2000, was the first part to include Jazelle and is currently immensely popular in mobile phones and PDA's.

ARMv5 — ARM10

The ARM10 is an evolution of the ARMv5 architecture aimed at maximal performance. It extends the pipeline to 6 stages, and includes support for a floating point coprocessor (VFP). It is projected to run at speeds of around 260Mhz.

ARMv6 — ARM10, ARM11

The ARMv6 adds memory management features, a new multimedia instruction set, and a longer pipeline (8 stages), allowing even higher clock speeds, up to 335Mhz. The new multimedia instructions are SIMD[3] and resemble the MMX, SSE and SSE2 instructions sets as used on common PCs. ARMv6 was introduced in 2001.

Neither the ARM10 nor ARM11 have had significant uptake in the market yet, and are not nearly as widely available as the ARM7TDMI and the ARM9EJS. Of course, this may change as time passes.

1.2.3 Future development

While the ARM10 and ARM11 focused on higher performance, particularly in the high-end DSP area, ARM also continued to develop toward the lower end of the spectrum, introducing Thumb–2.

[3]Single Instruction, Multiple Data

Thumb–2 capitalizes on Thumb's strength (32 bit processing with half-sized instructions) and fixes some shortcomings in the original design. It allows Thumb code to be freely mixed with ARM code, eliminating the need to switch modes, and adds several new instructions [3]:

- Move a 16 bit constant into a register. This reduces the need for the programmer to store immediate constants in memory.
- Bitfield handling instructions, allowing single bits to be set or checked in a single instruction.
- Table branching, allowing switch statements to be expressed more compactly and faster.
- If—Then instructions, which conditionally execute the next 1 to 4 instructions.

ARM claims Thumb–2 attains 98% of ARM's speed with a 26% better code density.

It remains to be seen whether Thumb–2 really delivers on these promises when used in practice. We can make several observations about it, though:

- Thumb–2 is essentially meant to be generated by a compiler, just like Thumb–1. This means that the compiler must be smart enough to make use of all features. While ARM seems to have accomplished this with their own compiler, third party manufacturers may not follow so quickly, which will reduce support for the architecture.
- A strength of the ARM architecture was originally it's simplicity and orthogonality. Thumb–2 appears to be opposed to this, introducing more complex instructions. But it's important to remember that the simplicity of the ARM wasn't so much an intentional design feature, but rather born out of necessity. This evolution is further made possible by the advancement of processor design tools in the last 20 years [4]. Furthermore, a little added complexity can sometimes prevent the need for adding much more elsewhere — conditional execution allows the ARM to do away with advanced branch prediction.
- Most new additions to the ARM architecture are not very open. This applies to Jazelle as well as Thumb–2. This may be resolved in the future, but without the wide third-party support the current ARM architecture enjoys it remains to be seen if it can be as successful. It's hard to say whether ARM's acquisition of Keil helps or hurts in this aspect.

Nevertheless, Thumb–2 takes the ARM's strong points and makes them even better. Thumb–2 is used on the ARM Cortex cores, which are expected to begin production shortly.

Aside from Thumb–2, ARM will introduce a more advanced SIMD instruction set, called NEON, and security enhancements called TrustZone. Not much specifics about both technologies are known about this time.

StrongARM and XScale

In 1995, Digital Equipment Corporation (DEC), in cooperation with ARM, developed a new, faster CPU based on the ARM architecture, named the StrongARM. The StrongARM has a 5-stage pipeline, separate data and instruction caches[4], and could be clocked up to 206Mhz. The chip devision of DEC was later sold to Intel, which gave them access to the ARM line. Intel adopted the design as a replacement for their own, less successful RISC chips, and later improved it to form the Intel XScale. Intel added SIMD multimedia and DSP instructions, and quickly ramped up the clock speed of the XScale, first to 400Mhz, later to 624Mhz, and recently demonstrated the capability of the latest generation (Monahan) to run up to 1.25Ghz. The XScale uses the ARMv5TE architecture, and is probably the fastest, widely available ARM chip, being used on high-end PDAs. It should be noted that Intel's SIMD and DSP extensions (named Wireless MMX) are different and incompatible with the SIMD extensions in the new ARMv6 architecture.

1.3 ARM peculiarities

While the ARM architecture looks like a RISC design, it nevertheless has it's own set of peculiarities. We already touched upon one earlier: the ARM has a native multiply and multiply-accumulate instruction (MAC), which is not found on 'pure' RISC designs.

RISC features:

- Load/store architecture: Except for the load and store instructions, all instructions operate on registers.
- Large register file. The ARM has 16 registers of each 32 bits wide, of which 14 are available during normal usage.
- Single-cycle instructions: Most instructions finish in a single cycle, allowing heavy pipelining and fast clock speeds.
- Fixed width instructions: all instructions are 32 bits wide. This means that for some instructions, much space is wasted whereas for others,

[4]This actually causes the StrongARM to be incompatible with the ARM in the case of self-modifying code. This is rare enough that it does not cause problems in practice.

there is not enough room. For example, the common instruction to increment a register with 1 takes a full 32 bits. Conversely, one generally cannot use a 32 bit immediate constant in an ARM instruction. The constant has to be stored separately in memory.

- Orthogonal instruction set: All instructions support all conditional, shifting and addressing modes and can access any register.

Peculiar ARM features:

- Advanced barrel shifter and rotation possibilities available in every instruction: while this follows quite directly from most data path designs, it is not available on most other architectures, and can provide good speedups for addressing or fixed point computations.
- Large set of condition codes available on every instruction: similar remarks as for the barrel shifter apply. The ARM even allows the programmer to indicate for every instruction whether it should modify the condition codes or not. This allows well-written code to do away with numerous branches and replace them by conditional execution.
- Pre- and post indexing: instructions accessing memory can increment or decrement their index register before or after executing. This can save an instruction almost every time anything is ready from or written to memory. (Practical experience shows that compilers have problems with this if the code treats the data like an array, while they usually have no problems with shifting pointers. Unfortunately, readability and most existing code favors the former, so code needs special care to make use of this feature.).
- Uni-byte-sex capable: most ARM cores can be reconfigured to work either in little or in big endian mode. This is in stark contrast to other architectures which are generally strictly little (Intel x86) or big (Motorola 68x) endian with all related portability issues.

1.4 The ARM instruction set architecture

Note that we only look at the parts of the ISA that are really specific to the ARM. For a full ISA reference, many sources are available, such as [5].

1.4.1 Register file

The ARM has 16 visible 32-bit wide general purpose registers, numbered *r0* to *r15*, and 2 status registers, named *cpsr* and *spsr* (current and saved program status register). Registers *r13*, *r14* and *15* have a special function:

- r13: Stack pointer (sp)

- r14: Link register (lr). This stores the return address on a subroutine call.
- r15: Program counter (pc)

Whether or not *r13* can be used for general usage depends on the operating system. If one is present, it will generally require *r13* to point to a valid stack frame.

1.4.2 Processor modes

The ARM has 7 processor modes, which determine which subset of the register file is visible and what operations are allowed on *cpsr*.

1. abt: Abort. Triggered on a failed memory access.
2. fiq: Fast Interrupt Request. Triggered on a high priority interrupt.
3. irq: Interrupt Request. Triggered on a normal interrupt.
4. svc: Supervisor. The mode that the operating system works in.
5. sys: System. Like user, but allows full cpsr access.
6. und: Undefined. Triggered on an invalid instruction.
7. usr: User. The normal mode for applications.

When switching modes, the ARM will "bank" part of the register file, usually *r13, r14* and *cpsr*, and replace them with the corresponding stored registers for that mode. In total, there are 37 registers, with only 18 visible at any given time.

1.4.3 Current Program Status Register

The *cpsr* is divided into four 8 bit fields: flags, status, extension and control. Extension and status are not used by current ARM cpus. "Control" stores the current processor mode (ARM, Thumb, Jazelle), as well as which interrupts are enabled.

"Flags" stores the condition flags:

- Q: Saturation. The result overflowed or saturated.
- V: Overflow.
- C: Carry flag.
- Z: Zero flag.
- N: Negative. Bit 31 of the result is set.

1.4.4 Conditional Execution

All ARM instructions can be made conditional by setting the appropriate condition code (see 1.1). If the condition matches, the instruction is executed normally. A condition miss executes in 1 cycle. It is up to the programmer

Condition Code	Flags
EQ (Equal)	Z
NE (Not Equal)	$\tilde{Z}$
CS or HS (Carry Set/unsigned Higher-or-Same)	C
CC or LO (Carry Clear/unsigned LOwer)	$\tilde{C}$
MI (MInus/Negative)	N
PL (PLus/Positive or Zero)	$\tilde{N}$
VS (oVerflow Set)	V
VC (oVerflow Clear)	$\tilde{V}$
HI (HIgher)	C and $\tilde{Z}$
LS (Lower or Same)	$\tilde{C}$ and Z
GE (Greater or Equal)	N = V
LT (Less Than)	N = $\tilde{V}$
GT (Greater Than)	(N = V) and $\tilde{Z}$
LE (Less or equal)	(N = $\tilde{V}$) or Z
AL (Always)	N/A

Table 1.1: Condition Codes

to specify which instructions are allowed to modify the condition flags, by marking them with an extra "S" suffix.

1.4.5 Barrel shifter

Most data processing instructions allow the simultaneous use of the barrel shifter to preprocess the data before it enters the ALU. 1.2 lists the possi-

Mnemonic	Result	Shift Amount
LSL (logical shift left)	$x << y$	0–31 or Rs
LSR (logical shift right	$x >> y$	1–32 or Rs
ASR (arithmetic shift right)	$x >>> y$	1–32 or Rs
ROR (rotate right)	$(x >> y \vert x << (32 - y))$	1–32 or Rs
RRX (rotate right and extend)	$(C << 31 \vert x >> 1)$	N/A

Table 1.2: Barrel shifter

bilities. Note that the shift amount can be a register, although care must be taking for scheduling conflicts if this possibility is used.

1.4.6 Loading and Storing

Single tranfers

The ARM provides several indexing methods for addressing memory: postindex, preindex and preindex with writeback. They are summarized in table 1.3. Offsets can be either a register, a scaled register using the barrel shifter,

Indexing method	Data from	Addr. reg. after	Syntax
Postindex	addr	addr + offset	LDR r0, [r1], #offset
Preindex	addr + offset	addr	LDR r0, [r1, #offset]
Preindex with writeback	addr + offset	addr + offset	LDR r0,[r1, #offset]!

Table 1.3: Indexing Methods

or an immediate value.

Multiple transfers

There is an additional possibility to transfer multiple words in a single instruction. In such a case, a set of register must be specified as the source or destination operand. The possible indexing methods are listed in 1.4. Multiple transfer can both be used for moving blocks of data as for stack

Adressing mode	Start address	End address	Raddr!
IA (Increment After)	Raddr	Raddr + 4 * N − 4	Raddr + 4 * N
IB (Increment Before)	Raddr + 4	Raddr + 4 * N	Raddr + 4 * N
DA (Decrement After)	Raddr − 4 * N + 4	Raddr	Raddr − 4 * N
DB (Decrement Before)	Raddr − 4 * N	Raddr − 4	Raddr − 4 * N

Table 1.4: Multiple Load-Store Indexing Methods

operations.

```
; 32 byte block transfer, from address in r9 to address in r10
LDMIA r9!, {r0-r7}
STMIA r10!, {r0-r7}
```

For stack operations, several aliases are defined, which map the ARM indexing modes to (A)scending or (D)escending and (F)ull or (E)mpty stacks respectively. ARM standard usage defined the stacks as being Full Descending, meaning that LDMFD and STMDF (aliases for LDMIA and STMDB) should be used for stack access.

```
; store r0 to r4 on the stack
STMDF sp!, {r0-r4}
; pop r0 to r4 from the stack
LDMFD sp!, {r0-r4}
```

1.4.7 Branching and subroutines

The ARM has 4 branch instructions. B (Branch) is an unconditional jump, similar to JMP in other architectures. BL (Branch and Link) stores the current *pc* in the *lr* register, similar to a CALL. Returning can be done by either moving *lr* to *pc*, or via a (multiple) store. Below is an example of how a typical function prologue and epilogue would look.

```
; subroutine prologue
STMFD sp!, {r1,r3,r4,r5,lr}
; code
...
; subroutine epilogue, we return by popping lr into pc
LDMFD sp!, {r1,r3,r4,r5,pc}
```

1.5 DSP capabilities

1.5.1 Fixed point arithmetic and load-store

Given that the ARM has no floating point capability, all DSP related processing needs to be done with fixed point arithmetic. Luckily, the 32 bit registers, fast multipliers, combined with the availability of the barrel shifter in each instruction mean that high processing performance is still possible. In fact, for many applications the ARM is fast enough that the need for a second, dedicated DSP processor will disappear.

An important difference between the ARM and DSPs is that the ARM is a pure load/store architecture and cannot simultaneously fetch from memory while doing an arithmetic operation. This means that, where possible, we will try to process data in small blocks, so as much data as possible can be cached in the register file.

The performance of any given ARM chip for DSP applications is mainly limited by the speed of it's multiplier and the speed by which it can retrieve and store results from memory.

Processor	16 x 16 = 32 bit MAC	32 x 32 = 64 bit MAC
ARM7	approx. 12	approx. 44
ARM7TDMI	4	7
ARM9TDMI	4	7
StrongARM	2-3	4-5
ARM9E	1	3
XScale	1	2-4
ARM11	0.5	2 (upper half)

Table 1.5: MAC timings per ARM core, in cycles

In table 1.5 [6] we can see that cores before the ARM7TDMI are slow, particularly for high precision arithmetic, due to the lack of the advanced multiplier. Getting results in higher precision entails a performance penalty on all cores. Note that for high quality audio, we want to have at least 13–14 correct bits [5] in the result, meaning that 16-bit processing will not be adequate for most computations. Additions are almost always single-cycle, with some exceptions on the more pipelined ARMs if heavy use of the barrel shifter is made.

There are some peculiarities with how different instructions interact that are relevant to DSP, but which differ from core to core. We will provide a short overview.

1.5.2 ARM7TDMI

Load instructions take 3 cycles to load a single value, whereas the load multiple instructions only require 1 extra cycle per additional word transferred. This means that it is advantageous to load 16 bit data in 32 bit pairs.

Multiply is 1 cycle faster than multiply-accumulate. This means that we can often prefer a multiply with a separate addition (which can, for example, make additional use of the barrel shifter) over a MAC.

1.5.3 ARM9TDMI

Load instructions are single-cycle, but the result is not available for the next 2 cycles. 16 Bit data is thus most quickly processed by loading it with separated 16 bit loads.

Multiply and MAC are the same speed. Using MAC saves an additional cycle over a separated multiply plus add.

1.5.4 ARM9E

The ARM9E can load and unpack 16 bit halves from 32 bit words natively, so the corresponding instructions should be used, if possible.

The ARMv5E architecture contains specific instructions for very fast 16 bit multiplies, which are single-cycle, although the result is not available for one extra cycle (except as accumulator input to another MAC instruction).

1.5.5 ARM10, ARM11 and later

Loads and stores multiples run in the background, meaning that proper scheduling can make them essentially free. For aligned accesses 64 bits can be transferred per machine cycle, so data should be aligned whenever

[5] This is about the resolution of a standard on-board sound card, and about the limit of perception for most untrained listeners.

possible. 16 bit MAC's take an extra cycle over 16 bit multiplies, so it makes sense to split them up if one can make use of the barrel shifter.

1.5.6 XScale

64 bit transfers are possible in a single cycle, with proper scheduling and alignment. The result of a multiply is not available for 1–3 cycles, depending on how many bits are used in the multiplicand.

1.5.7 DSP-specific extensions

Because DSP related work has been some common, the ARM architecture has been extended several times to cope better with DSP related tasks. The ARMv5E was the first revision to contains some additions to the instruction set, which are specifically aimed at DSP processing.

There are 3 classes of new instructions:

- CLZ (Count Leading Zeros): Generally used for normalization of integers, which in turn is useful for divisions, floating point conversion, logarithms and implementing priority decoders.
- QADD, QDADD, QSUB, ..: Saturating additions and subtractions. Very useful for DSP algorithms that need to clip outputs.
- SMLA, SMLAL, SMUL, ...: Signed multiply and MAC instructions, which provide more flexibility when working with 16 bit values. They are all single-cycle.

The ARMv6 adds a set of SIMD instructions, treating 32 bit values as $4 * 8$ bits or $2 * 16$ bits. Because there is no support for 32 bit arithmetic, nor is ARMv6 widespread, it makes not much sense to investigate them in detail at this point. They are mostly useful for video processing. The one noteworthy addition for audio processing is the SMMLA, SMMLS and SMMUL set, which provides 32 x 32 = 32 bit multiplications and MACs, retaining the upper 32 bits of the result. They are a natural fit for 32 bit fixed point arithmetic and do speed up audio related algorithms.

1.6 Coding for the ARM

These small but performance-critical differences between the ARM chips mean that optimal usage of a particular ARM chip can only be attained by writing specific versions for that chip. For many applications, coding specifically for a single chip in assembly language is not a very desirable proposition. This is particularly true if one wants to gain performance by usage of Thumb–1 or Thumb–2 along the way. Because of this, our preferred way of coding for the ARM will be to write the code in C, emphasizing the use of clear and well-optimizable code. Handling all the peculiarities of a certain ARM chip can then be left to the compiler, while we still have the option to look at the generated assembler code, and see if the compiler can use a little bit of help for the most critical sections.

There are a number of precautions we can use the help the compilers job:

- Use the `int` data type whenever possible. The ARM is a 32-bit architecture, and using 32 bit operations on 32 bit data will avoid needless casts.
- Use loops that count downwards. The end of the loop can be checked with a comparison to zero, which is always faster or more compact than comparing against an arbitrary constant or variable.
- Always store often used data in variables. The compiler will optimize these extra stores away whenever appropriate, but we will have informed it that they cannot be modified by unrelated functions or pointer dereferences, which avoids needless extra memory fetches.
- ARM will pass up to 4 function parameters in registers. Use functions with 4 or fewer arguments if possible.
- Don't divide if at all possible. The ARM has no native division, and it will generate a slow library call. Division by small constants can often be replaced by a series of adds, shifts and subtractions, so those don't count.

Note that several of these remarks are not ARM specific, but will speed up code on any architecture.

Some references like [6] recommend the use of loop unrolling to further speed up ARM code. We would advise caution: loop unrolling often hurts code readability, and most modern compilers can do this optimization itself, particularly so when profiling feedback is available. Leaving the unrolling decision up to the compiler allows it to better account for the specifics of the targeted chip, and will avoid pessimization if the code has to be changed afterwards.

1.7 ARMs with DSPs

A noteworthy development is that of ARM chips which are combined with DSPs on the same chip. Texas Instruments is a great pusher of this kind of architecture, combining the ARM9E or ARM11 architecture with their TMS320 DSP architecture. The idea is to have the ARM do the general purpose processing, and offload all the heavy DSP work to the dedicated

DSP processor. The OMAP is used in almost all high-end mobile phones and a large number PDA's.

The Apple iPod is a variation on this theme, using a dual ARM7TDMI together with some audio decoding acceleration hardware.

1.8 Conclusions & outlook

The ARM architecture has succeeded in establishing itself as a leader for any embedded application where both power consumption and good performance are paramount. Extensions and improvements to the architecture have continuously boosted processing speed, while retaining a low power consumption. Specific attention has been payed to the usage of ARMs for DSP processing tasks, through the addition of extra instructions specifically meant for DSP work.

The ARM7TDMI-S is widely used in practice as a high-performance yet very-low power-consumption microcontroller, used in cheap devices with modest demands. Likewise, the ARM9EJ-S is widely used in applications with slightly higher processing demands, such as high end portable devices. We will investigate the performance of audio coding algorithms on both of these chips.

In the future, we can expect ARM to continue to expand, both in the low end through the speedups brought by Thumb–2, as well as the high end, by usage of the new ARM10 and ARM11 designs. Meanwhile, Intel has been improving it's high-performance ARM offerings, which take a different path from those of ARM itself. The success of ARM depends partly on it's support by third party tools and suppliers, and it remains to be seen how soon and how well supported the new ARM architectures will be.

Chapter 2

Psychoacoustic audio coding

2.1 Introduction

The goal of psychoacoustic audio coding is to exploit our knowledge of the human hearing system in such a manner that we can exploit more redundancy and irrelevancy in sound and music signals, than would be normally available via standard, general purpose coding methods.

We'll provide a very short overview of the physical composition of the human hearing, and then look more in depth at the known effects which this produces and which we can exploit. Next, we'll look at practical algorithms that allow us to use these effects for audio coding purposes.

2.2 The human auditory system

We can divide the physical part of the human auditory system in 3 parts:

1. Outer ear
2. Middle ear
3. Inner rear

The outer ear consists of the visible part of the ear, called the pinna, and the ear canal. The shape and folds of the pinna will diffract and reflect the sound, providing the rest of the hearing system with with additional directional information. Given that all humans are physically different but similar on average, the same applies to this effect, and it can be modeled via so-called Head Related Transfer Functions (HRTF) [1].

[1]For more information about the current knowledge regarding (spatial) audio perception, please see [7].

The ear canal is a tube that runs from the outer ear to the middle ear. It protects the middle and inner ear, and also provides a 10dB gain around 1 to 4kHz[2], depending on the on the individual.

The middle ear consists of the eardrum plus the ossicles (3 small bones) and connects to the cochlea. It provides impedance matching between the air in the ear canal and the fluids in the cochlea, and is further believed to serve as both an amplifier with overload protection, as well as to increase temporal resolution at high frequencies, by serving as a high pass filter.

The inner ear consists of the cochlea, a spiraling hollow bone, the Organ of Corti, and several membranes. At the start of the cochlea lies the oval window (fenestra ovalis). When the middle ear transmits vibrations, the oval window translates them into waves traveling along the so-called basilar membrane. On the basilar membrane are series of sensory cells, the inner and outer hair cells. The outer hair cells act as a series of tunable amplifiers, transmitting vibrations to the inner hair cells. The inner hair cells then translate the vibration to an electrical signal, which is sent to the brain.

The position of the receptors along the basilar membrane is peculiar and important: the traveling waves will peak at regions with a resonant frequency close to their own, and decay rapidly thereafter, activating different groups of receptors based on their frequency. This means that effectively a frequency-to-position transformation takes place, and that the cochlea can be seen as a series of overlapping bandpass filters.

2.3 Psychoacoustic principles

2.3.1 Critical bands & the Bark scale

The structure of the cochlea as described in the last paragraph is a defining element of how we look at the human hearing from a psychoacoustic point of view. The excitation of the hair cells will occur in groups which all belong to a particular band of frequencies, meaning that, for example, if we add an extra stimulus to an already excited group of cells, this will have noticeably less of an effect than if this stimulus were to happen in another group of cells, or translated back, another auditory band. It also means that the processing of signals is somewhat independent between the bands, and that any auditory phenomenon is best expressed in a scale which matches these bands. Hence we arrive at the “Bark scale” or the “critical bands”. The critical bands are about 100Hz wide on the low end, and widen up to 1/3th of an octave at higher frequencies. Their exact position and width is somewhat open to dispute, and several formulas exist. We use the one due to Traunmüller [9]. A similar but newer system is “Equivalent Rectangular Bandwidth” (ERB). While ERB is a newer paradigm, and not so widely

[2]Some sources claim up to 12kHz, see for example [8].

Bark	starting f (Hz)	ending f (Hz)	width (Hz)
1	20	107	87
2	108	214	106
3	215	300	85
4	301	408	107
5	309	516	107
6	517	645	128
7	646	774	128
8	775	925	150
9	926	1097	170
10	1097	1269	172
11	1270	1485	215
12	1486	1722	236
13	1723	2002	279
14	2003	2325	322
15	2326	2712	386
16	2713	3164	451
17	3165	3703	538
18	3704	4392	688
19	4393	5275	882
20	5276	6416	1140
21	6417	7708	1291
22	7709	9409	1700
23	9410	11864	2454
24	11865	15654	3789
25	15655	$\pm$20000	$\pm$4345

Table 2.1: Bark scale (Traunmüller)

Reference	NMT	TMN
Johnston '88 [11]	$(14.5 + Bark)$dB	5.5dB
Johnston '93 [12]	$(19.5 + (18/26)Bark)$dB	$(6.56 - (3.06/26)Bark)$dB
ISO 11172-3 (MP3)	29dB	6dB
ISO 14496-3 (AAC)	18dB	6dB

Table 2.2: NMT/TMN Formulas

used or understood as the Bark scale, it should be a prime contender for further research, notably given some of the observations in [10].

2.3.2 Simultaneous masking & tonality

Due to the limitations of the hearing system, it is possible for a sound to prevent another sound from being heard. This phenomenon is called "masking" and is a prime actor in the workings of a perceptual audio codec. Our main goal is to determine the SNR that is required for a particular part of the signal, so that the reproduced signal masks the quantization noise we wish to introduce. We will first investigate what happens if two sounds play simultaneously (simultaneous masking).

As explained previously, the structure of the cochlea is such that a strong signal in one bark will prevent another weaker signal in the same bark from being detected. The difference in intensity between both signals, for one signal to be fully masked by the other, is dependent on the structure of the signals. For this reason, we will define two masking distances, Noise-Masking-Tone (NMT) and Tone-Masking-Noise (TMN).

- Noise-Masking-Tone: This is the ability of a noise-like signal to mask away a less powerful tone-like signal. Common values for NMT are around 6dB.
- Tone-Masking-Noise: the ability of a tonal signal to mask away a less powerful noise-like signal. Common values for TMN are around 18–24dB.

All practical signals will fall in between these two extremes, by an amount that is determined by how noise-like or tone-like the signal is, called the "tonality" of the signal[3]. The NMT and TMN are actually slightly Bark dependent, and different formulas can be found in the literature, see for example table 2.2.

The large difference between NMT and TMN (up to 33dB worst-case) necessarily means that the determination of the tonality per Bark is one of

[3]More correct is: the temporal instability of the signal. The concepts of tone and noise are more familiar to work with and an acceptable simplification, so they are used instead.

the must crucial steps in the psychoacoustic model. Different algorithms for this are available; two common are the Spectral Flatness Measure as explained in [11], and spectral unpredictability (chaos measure) as used in the normative parts of recent ISO standards. Several other methods are known, but the jury is still out as to what is the best method.

2.3.3 Absolute Threshold of Hearing

The Absolute Threshold of Hearing (ATH) is based on the fact that there is an absolute minimal intensity level below which a sound is not audible in any circumstance. Notably, the ATH drops off at very steeply at very low and at very high frequencies, expressing that we cannot reasonably hear sounds below 20Hz and above 16–19Khz[4].

The main problem with applying this knowledge to audio coding is that we do not know in advance what the actual playback level will be. In fact, it might very well be variable if the user turns the volume knob. So in practice we will have to do with some reasonable, heuristic assumptions.

This issue is not limited to the ATH alone, in fact almost all psychoacoustic effects have some relation to the playback level. However, their variability is in general not of such a large extent that we have to explicitly take this into account in order to have a working system.

2.3.4 Spreading

Masking also occurs between critical bands; very loud signals in one band can cancel much quieter ones in adjacent bands. While the exact amount is once again a disputed issue, with varying formulas available to determine the amount of masking that must be spread between different bands, we can use the following as a good average:

- Spreading from a Bark to the next higher Bark: 30dB of masking
- Spreading from a Bark to the next lower Bark: 15-20dB of masking

2.3.5 Temporal effects

In addition to simultaneous masking, the ear also exhibits temporal masking effects, whereby a quiet sound can be masked by louder sounds shortly in front (forward masking) or behind it (backward masking).

Forward masking runs into excess of 20 ms, while backward masking can be less than 1 ms for a trained listener.

Exploiting temporal masking effects is possible, but seriously complicated by the natural interdependency between frequency and time resolution. In general, the *lack* of backward masking causes serious problems that

[4]Depending on listener age and other factors.

any audio codec must specifically address to obtain a reasonably high quality result. The cause of this is that we do our computation in the frequency domain, and any quantization will spread out in the time domain when the reverse transformation is applied. If this spread is larger than the backward masking, our quantization noise will no longer be masked, resulting in very audible artifacting (pre-echo).

2.3.6 Noise and noise

In the case of pure or almost pure white noise, the hearing system is not able to make a differentiation between an original noise signal and another signal with the same energy contents, as long as there are no pathological phase interactions, and the temporal envelope is sufficiently similar.

This means that in these circumstances, we don't actually have to try to code the fine detail of the signal, but can suffice with transmitting just it's energy level and generating random noise of the same energy level on playback.

2.3.7 Binaural effects

The above phenomena so far have been described in the context of mono signals. In the case of stereo signals, we have to be slightly more careful, and watch out for so called "stereo unmasking" effects, where the combination of the signals in both ears causes differences in the way we perceive the sound.

The Binaural Masking Level Difference (BMLD), sometimes also called Binaural Masking Level Depression, describes the fact that for lower frequencies, the hearing system can detect level and phase differences between the signals at both ears, and provide a spatial separation of the sound that can lower the needed SMR level for a signal by up to 20–25dB. A formula for worst-case behavior in terms of frequency can be found in [12].

2.4 Toward the frequency domain

The psychoacoustical effects described in the previous section act and are described for the most part in the frequency domain. This means that to properly exploit their properties, our audio codec will also need to have some way to operate in the frequency domain.

2.4.1 Quadrature Mirror Filters

A Quadrature Mirror Filter (QMF) consists of a combination of a real-valued FIR highpass and a lowpass filter, following certain mathematical properties, which when downsampled by a factor of 2, form a filterbank which splits the signal into two equally wide bands. The design of the filter also ensures that the aliases produced by the analysis filters are cancelled on synthesis, producing an aliasing-cancellating filterbank. There is a large degree of freedom in the design of these filters, however to be practically useful for audio coding we are generally interested in certain properties: no phase distortion, no amplitude distortion, good stopband and transition band characteristics. Unfortunately, the above characteristics do work against each other in the filters' design.

Classical wavelets such as the Daubechies, Coiflet and Haar wavelets satisfy a QMF relation and are hence also QMF filters. They have no amplitude or phase distortion and thus provide perfect reconstruction, however, they generally do not have good frequency selectivity, particularly for smaller orders.

Another classical series of QMF filters are those proposed by J. D. Johnston in [13]. They do not achieve perfect reconstruction, but trade off excellent frequency selectivity with minimal amplitude distortion, and no phase distortion.

2.4.2 Subbanding

The traditional approach used in older audio coding systems is the use of subbanding. Subbanding splits the signal into a series of equally-wide frequency bands, which are downsampled by the same amount as there are bands. The number of bands determines both the time and frequency resolution, with more bands trading time resolution for frequency resolution, and vice versa.

Traditionally, this was implemented via cascaded Quadrature Mirror Filterbanks: if we apply a QMF filterbank to the output of a QMF filterbank, we can keep increasing the number of bands and our frequency resolution. Later on these were replaced with equivalent but computationally more efficient Polyphase Quadrature Mirror Filterbanks (PQMF), typically splitting the signal into 32 bands.

A problem in this approach can be seen by referring to the Bark table mentioned earlier; at the lower end of the spectrum Bark bands are about 100Hz wide. At a sampling rate of 44100Hz as used for example in CD audio, extremely large filters are required to attain a sufficient frequency resolution, while still maintaining good reconstruction and bandpass characteristics.

2.4.3 Wavelet filterbanks

The usage of wavelets-style filterbanks for perceptual audio compression is based on the observation that the increased frequency resolution is only needed in the lower parts of the spectrum, whereas the increased time resolution needed for avoiding pre-echo is only needed in the higher parts of the spectrum. It is possible to make a tree of stacked QMF filters that give a decomposition which roughly corresponds to the Bark scale, thereby seemingly meeting all goals, and many papers have been consequently published about this system (Wavelet Packet Decomposition).

Nevertheless, this approach has some issues. Firstly, the depth of the QMF stack and the order of the QMF filters has to be rather large to achieve a reasonable accuracy and good frequency separation characteristics, which entails a rather serious performance disadvantage.

One widely tried improvement to this scheme is to make the decomposition dynamic, trying different ones with different base wavelet filters, and to pick the one which leads to the most compact representation of the resulting signal.

But if we want to vary the decomposition, we need to split the input in blocks, and we encounter the issue of how to treat the block boundaries so as to avoid aliasing artifacts. We know of 3 approaches to this problem:

- Overlap the blocks. This is not a very good solution, since our filterbank is suddenly no longer critically sampled[5], which causes us to lose coding efficiency. There is also the question of how much to overlap the blocks, to which there is no definite answer.
- Employ special filters near the block boundaries, for example using Gram-Schmidt orthogonalization. The disadvantage is that the filter characteristics near the block boundaries will not be optimal any more.
- Use an interpolation scheme between blocks. This is also unattractive, because it should not be possible to find a scheme that can eliminate all artifacts all of the time.

2.4.4 FFT

Another approach to audio coding is to use the FFT to obtain a complete time-frequency transformation. This entails that we obtain a very high frequency resolution, at the cost of a total loss of time resolution.

The outstanding characteristic of this approach is that for most common audio sources, which are largely consisting or more-or-less stationary sinusoidal signals, we will achieve a high coding gain since most information will be centralized into just a few significant coefficients.

Moreover, the high frequency resolution also means that we can divide the resulting FFT coefficients up into groups corresponding very tightly to a Bark scale.

There are only 2 problems with using an FFT, firstly, the question of how to deal with aliasing at block boundaries. This can be solved by using overlapping, however, exactly the same remarks as for wavelets apply.

[5]Meaning, we are getting exactly as much data out of it as we are putting in.

Secondly, the total loss of time resolution means that we will cause large temporal artifacts when the signal is not stationary.

2.4.5 Modified discrete cosine transform

The Modified Discrete Cosine Transform (MDCT), introduced in [14] is an orthonormal filterbank, based around the type–4 Discrete Cosine Transformation (DCT–4).

The MDCT is peculiar in the sense that it will only produce n outputs for every $2n$ inputs, however, because it requires a 50% overlap between consecutive blocks, the end result is exactly a critically-sampled filterbank.

It achieves complete aliasing cancellation by exploiting the boundary conditions of the DCT–4. A very nice explanation of why and how this "Time Domain Aliasing Cancellation (TDAC)" works can and should be read in [15]. It is interesting to note is that this system does not require consecutive blocks to have the same size, and still works in the presence of a windowing operation.

Calculation of the MDCT boils down to performing a DCT–4 and applying the overlapping together with a butterfly operation[6]. The butterfly operation is a trivial set of multiplications and efficient algorithms for the DCT–4 are known, based on a similar decomposition as used for decomposing a DFT into a FFT. So, efficient algorithms exist for the computation of the MDCT.

Putting it all together, the MDCT has the following properties:

- Complete elimination of blocking effects.
- Critically sampled filterbank.
- Has the DCT–4 as the basis function, a transformation with a very high coding gain.
- Efficient computation.
- Possibility to trade off frequency/time resolution on a block-by-block basis, by switching between block sizes.

It should come as no surprise at this point that the MDCT is essentially the holy grail of audio coding and is consequently used in all recent major audio coding standards: MP3, AAC, AC3, ATRAC, Vorbis and WMA.

[6]Multiplication of the samples from the different blocks in the shape of cross or butterfly.

2.5 Quantization

2.5.1 Normalization

The values coming out of the time to frequency transformation can have a wide dynamic range. Most perceptual audio coders will hence apply a normalization step on the resulting spectral values, whereby they set up a partitioning in those output values, and divide them partitionwise by some kind of scaling factor, resulting in residual values with a smaller, controllable dynamic range.

Two examples of how this can be done:

- Split the spectrum in Barks (or half-Bark, 3^{rd} Bark, ...) wide partitions (forming so called scalefactor bands) and apply a scalefactor with a resolution of 1.5dB per partition. Used by MP3 and AAC.

- Split the spectrum in Bark partitions, store a 0.5db, 1.1dB or 2.2dB resolution scalefactor for each partition boundary, and divide all spectral values by a new scalefactor that results from linearly interpolating the scalefactors at the corresponding boundaries. Used by Vorbis.

2.5.2 Linear quantization

A simple form of quantization that nevertheless can be successfully used for audio coding is simply rounding the values obtained from the normalization step to integers.

Note that we can control the quantization accuracy by controlling the scalefactors. The larger the scalefactor, the lower the resulting values going into the quantization layer, and the lower the resulting accuracy and signal to noise ratio is.

2.5.3 Power-law quantization

A problem with linear quantization occurs when we are trying to obtain high signal to noise ratios, and hence we are dealing with large resulting values. This can be seen from the following observation: say that the value of 15.5 comes into our quantizer. We round this to 16, giving a quantization error of 3%. Now, the value of 1.5 comes into our quantizer. We round to 2.0, giving a quantization error of 33%. So clearly, a linear quantizer will cause relatively larger errors to occur for smaller values going into it, not giving a constant SNR over all the values in it's input range. A logarithmic quantizer does give a constant quantization error for every value in its range. In fact, neither is entirely desirable, since masking effects do cause larger quantizer errors for smaller values to be less well heard.

A practical solution is to use something which treads in the middle of those two systems, and for this reason, some audio codecs use power-law quantizers instead. The formula of

$$x = x^{3/4}$$

is used in both MP3 and AAC.

2.6 Lossless coding

2.6.1 Direct coding

The simplest method to store the quantized spectral values would be to just store them directly. This has the advantage of minimal complexity and fast processing, but does not allow us to use our knowledge of the distribution of the spectral values for further redundancy removal.

2.6.2 Huffman coding

Huffman coding is a well known entropy coding scheme which constructs prefix-free codes, given a known symbol distribution. It will perform optimally when the symbol probabilities can be expressed as negative powers of 2. Decoding is low complexity and only requires walking a binary tree for each symbol that we are trying to decode.

When the symbol probabilities are not optimally distributed, as can easily happen with small symbol alphabets, some efficiency can be gained by encoding blocks of values at once. The resulting probability distribution will often be better. This is specifically true if there is correlation between the values in a block, since in that case Huffman coding will further extract the redundancy between those values.

2.6.3 Arithmetic coding

Arithmetic coding is an improvement over Huffman coding in the sense that it will still generate optimal output regardless of the distribution of the symbol alphabet. However, it attains this at an increased cost in computing power.

Because the inefficiency of Huffman coding can mostly be circumvented by the above mentioned tricks, the use of arithmetic coding has not found wide application in current audio codecs codecs.

2.6.4 Vector Quantization

The idea behind vector quantization (VQ) is to not encode single scalar values, but to group them in vectors, store those vectors in a codebook, and only transmit the codebook indexes, which can be entropy encoded themselves. We can store multiple indexes for the same block we are trying to transmit, and combine the vectors in some way. There is a large degree of freedom in how we try to construct the VQ codebooks.

Blocked Huffman coding as described above can be considered a special case of Vector Quantization.

2.7 Stereo coding

In the case where we are encoding a stereo signal, we can further increase our compression by exploiting the redundancy between the channels, if present. A simple but very effective method is the usage of Mid/Side stereo coding. In this method, the signal is split up into a common and a difference channel. If there is a high redundancy between both channels, the Mid channel will contain all the entropy and the information in the Side channel will be negligible.

Further improvement is possible by doing this transformation in the frequency domain and by choosing bandwise whether or not we want to use Mid/Side coding for this particular frequency range.

2.8 Binaural Cue Coding

Binaural Cue Coding takes stereo or multichannel coding a step further by the observation that humans really only have two ears, and that the perception of stereo effects is being generated by the differences between what those two ears perceive.

The working is based on combining a number of different channels into a single downmixed channel, coding this channel, and adding side channel information related to the time, level and phase differences of the signal arriving at each ear, as well as the correlation between both signals.

A poor man's version of this tool is Intensity Stereo coding, which only transmits the level differences between the channels.

2.9 Practical audio codecs

We now turn to a number of audio coding systems used in practice and investigate how the above techniques are used in practice.

2.9.1 MPEG 1 Layer 1 & 2

The ISO/IEC Motion Picture Experts Group's standard for perceptual (CD-quality) audio coding was finalized in 1992, after a collaboration between researchers from a variety of companies and research institutions. The standard provided for 3 separate audio encoding systems, forming layers of increasing complexity and increasing performance. The first two layers, MPEG 1 Layer 1 and MPEG 1 Layer 2, are sufficiently similar that we will treat them as a whole.

Both codecs take the input signal and put it through a PQMF subband filterbank, decomposing the signal into 32 750Hz wide frequency bands, each containing 12 samples. The resulting spectral samples are normalized via scalefactors to have a maximal value of 1.0 per band. Per band, the scalefactor is transmitted in 6 bits, together with the used quantizer (4 bits). The quantizer to use is determined by the output of the psychoacoustic model. No entropy coding or stereo redundancy removal is used, although Intensity Stereo can be used.

Layer 2 works almost identically, but considers groups of 3 times 12 samples, and transmits either 1, 2 or 3 scalefactors per group, depending on the temporal characteristics of the signal. The number of possible quantizers is increased. It is still widely used in practice for the Digital Audio Broadcasting (DAB) standard.

2.9.2 MPEG 1 Layer 3

MPEG 1 Layer 3 is widely known as the "MP3" codec, achieved worldwide recognition and is probably the most used audio codec in existence.

It works by partitioning the input signal into an alternation of 576 or 3×192 sample blocks (depending on the required temporal resolution of the block being encoded) putting the input signal through a PQMF subband filterbank, decomposing the signal into 32 frequency bands in the same way as Layer 1 and 2. On each of the 32 subbands, an MDCT is performed. (The filterbank is hence a hybrid between subbanding and transforms)

The spectral output samples are divided into scalefactor bands, roughly corresponding to Barks. Per scalefactor band the psychoacoustic model is used to select a scalefactor, and the samples are sent to a fixed power-law quantizer[7].

The resulting quantized values are divided in 3 groups, zeros, small values (-1, 0 and 1) and big values (up to 8206), and a variety of block Huffman codebooks can be used to optimally code each range.

Mid/Side stereo is possible on a frame by frame basis, and Intensity Stereo is also supported.

There are some provisions for allowing the number of bits used to vary from frame to frame as the demand from the psychoacoustic model varies, even if the encoder is targeting a fixed bitrate, via the use of a bit reservoir.

[7]Note that this means that unlike Layer 1 and 2, the scalefactor actually determines the quantization accuracy, since the quantizer itself is fixed. The advantage is that we do not have to transmit which quantizer we are using.

2.9.3 MPEG 2 & 4 Advanced Audio Coding

Goals

The ISO/IEC improved upon their MPEG 1 standard, culminating in the finalization of the a new MPEG 2 audio coding standard in 1997. MPEG 2 extended the use of the MPEG 1 codecs into the multichannel area, but also introduced a new perceptual audio coder, originally named MPEG 2 Non Backwards Compatible (MPEG 2 NBC), but is now commonly known as MPEG 2 Advanced Audio Coding. The main interest at the time of its introduction was to achieve "EBU indistinguishable"[8] quality at 384kbps for 5.1 audio, however the eventual result exceeded the design goals and met the requirement at 320kbps for 5.1 channel audio, and 128kbps for stereo.

Design

MPEG 2 AAC improved upon its predecessors by doing away with the PQMF and hybrid filterbanks altogether, and relying entirely upon a single MDCT applied to blocks of either 2048 or 8×256 samples. Two different windows can be used for the MDCT, trading bandpass width versus stopband rejection on a block by block basis.

The resulting spectral values are split into 49 scalefactor bands (very roughly corresponding to half-Barks). From the output of the psychoacoustic model, the decision whether to use normal stereo, Mid/Side or Intensity stereo encoding is done on a per scalefactor band basis, the right scalefactors are selected and the samples are sent through a fixed power-law quantizer.

The quantized spectral values are divided into a number of groups with the same Huffman codebook and the encoder finds the optimal grouping strategy which combines an optimal selection of block Huffman codebooks with the lowest amount of groups.

In the case that we are encoding a group of 8×256 sample blocks, the encoder groups the scalefactor and spectral information, so a minimal number of codebook groups or scalefactor information must be transmitted.

AAC contains a number of other tools to improve the coding efficiency, such as the possibility to code isolated pulses in the frequency domain separately and Temporal Noise Shaping (TNS)[9].

AAC further defines several profiles, which expand the base layer (Low Complexity or LC profile) with a backwards prediction system giving further coding gain (Main profile), a special filterbank design to minimize coding delay (Low Delay or LD profile), or a simple form of scalability (Scalable Sampling Rate or SSR).

[8]Roughly speaking, this means that the codec did not produce an annoying distortion on any test signal, and was indistinguishable for the majority.

[9]We will treat this more in depth in a later section.

Extensions

In 1998, the MPEG 4 standard expanded AAC further, adding the PNS tool (which allows replacing a noisy scalefactor band entirely by just coding the energy in that band) to the LC and Main profiles, and a forwards prediction system (Long Term Prediction or LTP profile) to replace the computationally inefficient backwards predictor from the Main profile.

The resulting standard can safely be said to represent the state of the art in general audio coding to this day, and is still periodically being expanded to keep tracking new advances in audio coding, generally layering improvements on the Low Complexity AAC base layer. Examples of recent improvements are High Efficiency AAC (adding bandwidth extensions with Spectral Band Replication or SBR), High Efficiency AAC version 2 (adding Binaural Cue Coding for stereo signals), Scalable Lossless Coding or SLS (adding a correction layer that allows the codec to scale up to lossless quality) and Spatial Audio Coding (Binaural Cue Coding for multichannel scenarios).

High Efficiency AAC version 2 is capable of delivering high quality audio at bitrates as low as 20kbps, and is used for digital shortwave radio systems such Digital Radio Mondiale and XM Satellite Radio.

Popularity

Despite all of this, AAC did not achieve general recognition or support until being chosen as the format of choice for the Apple iPod. The performance of High Efficiency AAC is notable enough that it is now also becoming a standard format for new mobile phones.

2.9.4 Ogg Vorbis

Ogg Vorbis started out as a project in 1998 to provide the world with a patent-free audio codec, and was designed and programmed by Christopher Montgomery.

The design of the audio codec has a lot of similarities with AAC, and has similar performance characteristics.

It splits the signal into a sequence of blocks of either 2048 or 256 samples [10], depending on the temporal characteristics. The MDCT is applied with a custom, fixed window function. The spectral values are split into bands which roughly correspond to a Bark division, and at each band division a scalefactor is computed. The spectral values are normalized by linearly interpolating[11] between the two scalefactors at the ends of their Bark bands.

[10]The specification allows any combination of 2 separate powers of 2, but the values listed are the ones used for normal CD stereo audio.

[11]An earlier version of Vorbis used a quantized LPC filter, which is more computationally intensive and less stable.

The spectral values quantized with a linear quantizer. The quantized spectral values are then sent through a square polar mapping function. For an explanation of square polar mapping, see [16]. Note that square polar mapping depends on the entropy encoding step to actually remove the redundancy between the channels. The current implementations in the Vorbis encoder correspond very closely to Mid/Side Stereo coding and Intensity Stereo coding.

The resulting values are coded via Vector Quantization. The current usage of VQ in Vorbis is such that this can be considered equivalent to a block Huffman encoding, although one important difference is that Vorbis provides the required Huffman codebooks at the start of each stream, while they are fixed in AAC.

2.9.5 Wavelet Audio Codecs

Wavelets have become something of a buzzword in the compression world, and we can find many publications about perceptual audio coding with wavelets, all of which claim excellent results. One example is [17] which claims “almost transparent” coding at 110-126kbps as early as 1993.

It is interesting to note that the MPEG ISO AAC verification tests[12] in 1997–1998 concluded that none of the codecs under test achieved statistical transparency[13], although they did achieve “EBU indistinguishable quality”. Despite the fact that this EBU recommendation and proper testing procedures have been known at least since 1991, essentially all wavelet audio coding papers we encountered carefully sidestepped the issue by making claims of “nearly transparent” or “almost transparent” coding, without any further formal definition.

As a result, we are rather sceptical of any such claims of excellent results, partly because in general the evaluation of quality appears to be based on very rough tests, the statement of the resulting performance is so broad as to be useless, and the results are impossible to verify (usually many details of the test procedure are missing, let alone working software published). There is, to the best of our knowledge, no wavelet based audio coding scheme is in practical use today, let alone a standardized one, nor is any wavelet audio encoding software available that could be used for a performance comparison. This also complicates the evaluation of any new wavelet audio coding method. What are we going to compare it against?

[12]For a reference, see [18].

[13]In fact, the author is not aware of any such result even in the year 2006.

2.10 Quality evaluation

Subjective tests

The evaluation of the quality of a perceptual codec is a tricky matter, because it relies inherently on the way the human brain processes information, a process that is not entirely mapped out and understood in a foolproof way. Therefore, a correct evaluation will always require a human factor. This causes a lot of complications, because not all humans are or hear the same, and not even the same human will have the same perception all of the time. The performance of the algorithm will also naturally vary between different source materials.

There are further pitfalls. Humans are sensitive to placebo effects. The knowledge that they are listening to an audio signal that is degraded in some way will often make the listener believe that he can hear those differences, even if this is in fact not the case due to psychoacoustic reasons.

Bringing it all together, the correct way to grade the quality of an audio coding method is to set up a double-blind test with a sufficient, representatively chosen number of items under test, and a sufficient, randomly chosen listening audience performing the test under adequate conditions. Statistics will then allow us to make sound decisions from the data gained from the various listeners.

This is the situation in an ideal world. In practice, when we are developing a codec, it is not feasible to set up such a test for each and every improvement or variation of parameter that is tried. So, we have to compromise, and do much smaller, perhaps even single-person tests, on just a few samples. In the case that the audible distortions are large, we could also drop the requirement for the test to be blind. These are engineering decisions; they can cause us to make judgement errors, but they at least allow a reasonable amount of experiments to be performed. The end result can then still be judged by a more sound method.

Objective tests

Despite the need we have for them, the above limitations of Subjective Testing are well known, and methods have been sought to "objectively" assess the quality of codecs, if possible even in a fully automated manner without human intervention.

Because of the nature of the perceptual codecs, it is vital for any such method to take into account the nature of the human hearing system. The ITU standardized one such method in ITU Recommendation BS.1387, also called PEAQ.

The PEAQ tool takes a reference file and a file-under-test, and feeds it to a model of the human hearing, which produces a series of Model Output Variables (MOV's). Examples of MOV's include the average Signal To

Masker Ratio and audio bandwidth. The output of the MOV's is sent to a neural network, which is tuned with the results of previous listening tests performed by the ITU.

The output of the tool is finally an Objective Difference Grade (ODG) ranking, which equates to the score that a listener would give the sample compared to the reference, on a scale from 5.0 to 1.0.

There exist both simple and advanced versions of the PEAQ tool. One implementation of the simple model is provided by the McGill University of Montreal, and can be found at:

`http://www-mmsp.ece.mcgill.ca/Documents/Software/`

Both the simple and advanced model are very useful tools for development and evaluation, since they present an automated way to get feedback on the codec, and are absolute measures that can be perfectly reproduced. However, relying on their results blindly is very dangerous, as the tools do have flaws, and they do sometimes give totally wrong results.

2.11 Conclusions & outlook

We have presented a short but fairly thorough overview of the current knowledge about the human auditory system, and an overview of currently deployed perceptual audio coding methods.

While many of the presented methods are very powerful, they do suffer from the fact that sometimes during their standardization not all design aspects were properly understood. In some cases, the design was based on a necessary consensus between a large variety of contributors, which lead to designs that can hardly be called elegant. The two differing normalization systems in the Vorbis codec, the hybrid filterbank in the MPEG 1 Layer 3 codec, or the large number of variants for Advanced Audio Coding are clear examples of this issue.

With the benefit of hindsight it is possible to design a much simplified coder with competitive performance, and investigate some additional techniques for potential improvement.

Chapter 3

Bacon: a low-complexity audio codec

3.1 Design considerations

The design of the codec has the following goals in mind:

- The codec should be as simple as possible, both from a design point of view, as in terms of implementation.
- The resulting performance should still be equal or superior to (a high quality implementation of) an MPEG 1 Layer 3 codec.
- The decoder should be able to run in fixed point mode on contemporary small ARM chips, with a minimal requirement of RAM and ROM.

The last requirement in particular is driven by the current state of ARM chips. Cheap Flash-based ARM chips with high performance are easily available, but the cost per unit goes up very steeply with the amount of ROM, and especially RAM, that is required. For example, for less than 5 Euros a piece it is possible to obtain an ARM chip capable of running at 55Mhz, but the same chip will only have 16k RAM and 64k ROM available. For a chip with 32k RAM, the price already rises to 7.5 Euros.

Furthermore, by designing our own codec, we increase our understanding of current audio coding technologies and experience from the front line what practical issues one encounters during the design and implementation phase.

Lastly, by simplifying our design we can make a baseline that is easily adaptable to further experimentation with new ideas in the audio coding field.

3.2 Transformation

3.2.1 Avoiding block switching

One of the root causes of complexity in audio codecs is the need to deal with block switching. As explained in the section about the MDCT, block switching allows a codec using the MDCT to use shorter blocks whenever higher temporal resolution is needed. However, this does cause a large number of complications:

- Because of the overlapping structure, we need to know in advance if the next block is going to be a short one, so we can adjust the window and overlap that is used in the current block.
- The size of the blocks we are using is no longer constant, which may require us to add a lot of code to handle the two cases in all the following processing.
- Short blocks have a lower frequency resolution, a lower coding gain, and require relatively more side information to be stored. (A lot of complexity in AAC comes from the regrouping to reduce the latter problem).
- We need to store the required windows and fixed point data tables for both sizes of blocks, and also for the case of the transition between them. (In the case of AAC where different window functions are possible, it gets even worse)

So, if at all possible, we wish to avoid block switching and only use a single window size. However, we must have a way to deal with pre-echo situations if we want to reach an acceptable quality level. We tried 2 approaches to this problem, firstly, to use new style of wavelet filterbank, and secondly, to use Temporal Noise Shaping.

3.2.2 The Modified Discrete Wavelet(like) Transform

As noted in the previous chapter, a wavelet filterbank can give us improved temporal resolution particularly in the high frequencies[1], which is exactly what we need to improve our performance in transient situations.

Conversely, for the case where we are mostly dealing with sinusoidal signals, this extra temporal resolution will work against us, because we must now store a series of coefficients (one for each time “unit”) instead of a single one.

[1]Because of the time and frequency duality, high temporal resolution for low frequency signals makes little sense.

It is therefore tempting to look for a way to switch between an MDCT based filterbank and a Wavelet based one. One such scheme was explored in [19], which uses the observation that the MDCT is a combination windowing butterflies and a DCT–4 orthogonal transform. We can replace the DCT–4 by another transform, and we are particularly interested in the case where this is wavelet filterbank. Some other considerations, notably regarding the need for low-overlap windows, can be found in the same paper. A problem of the proposed method is the need for transition filters at the end of blocks, to avoid block aliasing artifacts[2].

A new method is now proposed which avoids the need to employ transition filters, which we will refer to as the MDWT.

MDCT boundary conditions

The basis vector for a DCT–4 is even at the left end and odd at the right end[3], i.e. it corresponds to alternating odd/even boundary conditions. This means that if the input to the DCT–4 is a series of samples

$$(a, b, c, d)$$

it corresponds to an extended signal:

$$(d, c, b, a), (a, b, c, d), (-d, -c, -b, -a)$$

It is this particular boundary behavior that makes time-domain aliasing-cancellation work, because the samples fold back in the correct manner to cancel each other in the overlap operation of the MDCT.

If we can substitute the DCT–4 by another transform with the same boundary behavior, the aliasing-cancellation of the MDCT will still work, and there is no need to employ transition filters to get rid of block boundary problems.

Even order, linear phase

We note that an even order, linear phase QMF filter consists of components of the following form:

Analysis lowpass: $(a, b, c, d), (d, b, c, a)$
Analysis highpass: $(e, -f, g, -h), (h, -g, f, -e)$
Synthesis lowpass: $(e, f, g, h), (h, g, f, e)$
Synthesis highpass: $(-a, b, -c, d), (-d, b, -c, a)$

[2]It's interesting to note that the authors of that paper managed to patent the whole concept of switching between a wavelet and an MDCT transform, regardless of the method used.

[3]See for example [20] and [15].

Applying the filters to a signal as extended in the above explained manner leads to a filtered result with the following structure:

Lower band: $(p, o, n, m), (m, n, o, p), (-p, -o, -n, -m)$
Upper band: $(-l, -k, -j, -i), (i, j, k, l), (l, k, j, i)$

It can be seen that:

- The boundary conditions of the output fulfill the above requirement, or can be trivially made to do so by some sign changing.
- The output only has as many unique samples as the signal before extension, meaning we only need to transmit those to be able to reverse the filter operation, which in turn means that we are still critically sampled.

The application of this kind of QMF filter can be repeated on the newly obtained lower bands, thereby allowing further subdivision of the signal in frequency.

Suitable filters

To be applicable for audio coding, there are several requirements on the usable QMF filters:

- Stopband attenuation should be high enough so that the alias of one band does not noticeably show up in the other band, otherwise we might end up quantizing it with an inappropriate precision. For the first level split we want to perform (0–11khZ and 11-22kHz), level differences of > 50dB are typical in modern pop music, so we require an attenuation sufficiently in excess of this.
- QMF filters will alias from one band to another (and cancel this again on reconstruction). We want to prevent this aliasing from propagating further down our cascade, which would cause issues with differing quantization accuracy. This requires the transition band to not span over more than $\frac{1}{3}$ of the spectrum. For an illustration, see figure 3.1.
- Filter order should be as low as possible, to minimize computing requirements.
- The combination of analysis and synthesis filters should obtain either perfect or near-perfect reconstruction.

Unfortunately, these requirements are in direct conflict with each other. In practice, for a lossy audio codec, perfect reconstruction is the easiest

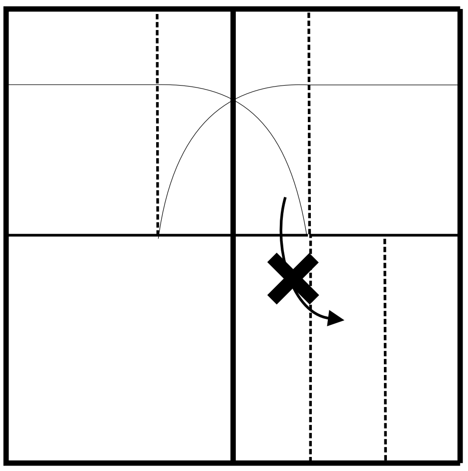

Figure 3.1: Transition band requirements for QMF filters

one to give up on, although we can't really strictly speak of a "wavelet" filterbank any more in this case.

In Bacon, we implemented a tree structure based on the Johnston QMF filters found in [13], the slightly improved versions by Yu found in [21], and a 1.3 Bi-orthogonal wavelet. The smaller filters are used when the number of samples to split in two bands is smaller than the order of the filter itself.

3.2.3 Transform switching

Complexity considerations

Instead of solving artifacts caused by lack of time resolution problems by switching the block size of our codec, we can now switch the used time to frequency transformation instead. This is of much reduced complexity, because outside of the transformation block no code needs to be aware of the fact that something has changed. Such a thing is not possible if the size of the block, and hence, the resulting amount of samples to encode, is different, let alone if we try to group multiple short blocks to improve our coding efficiency.

Switching decision

The decision when to switch transforms can be based on the same kind of techniques used for normal block switching, such as detecting a surge in high frequency energy, however we opted for a simpler method: we determine which of the two transformations results in the largest amount of coefficients that are smaller than $\frac{1}{8}$ of the largest coefficient value. This gives a good impression of the energy compaction attained by the transform in question.

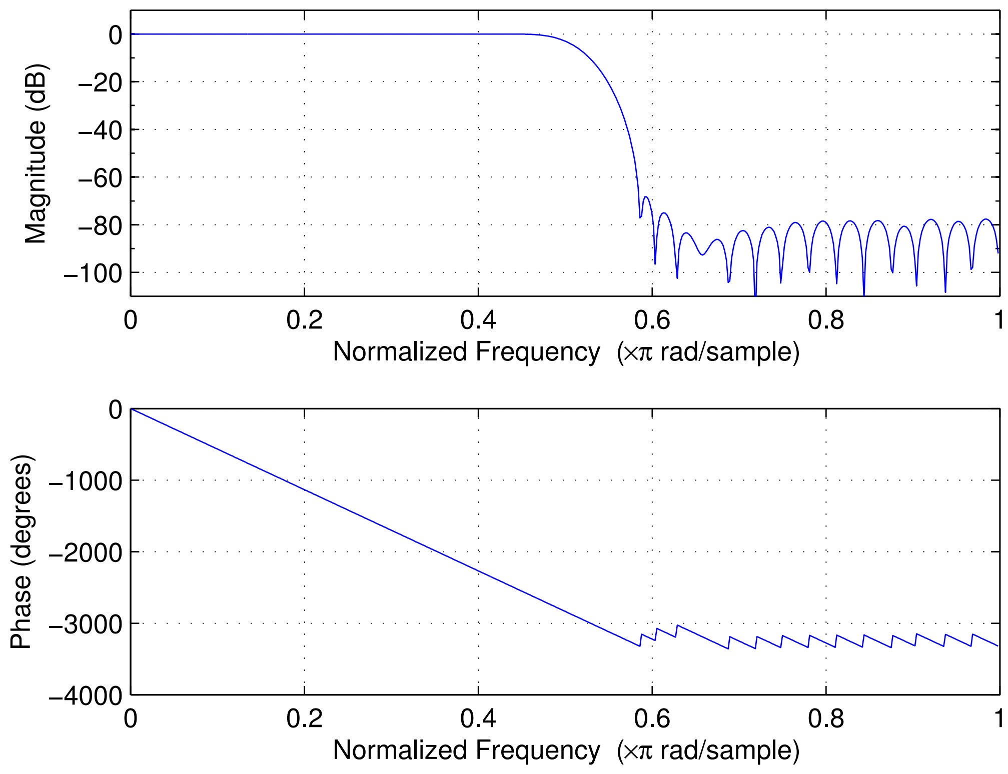

Figure 3.2: Filter properties of Johnston 64 tap QMF

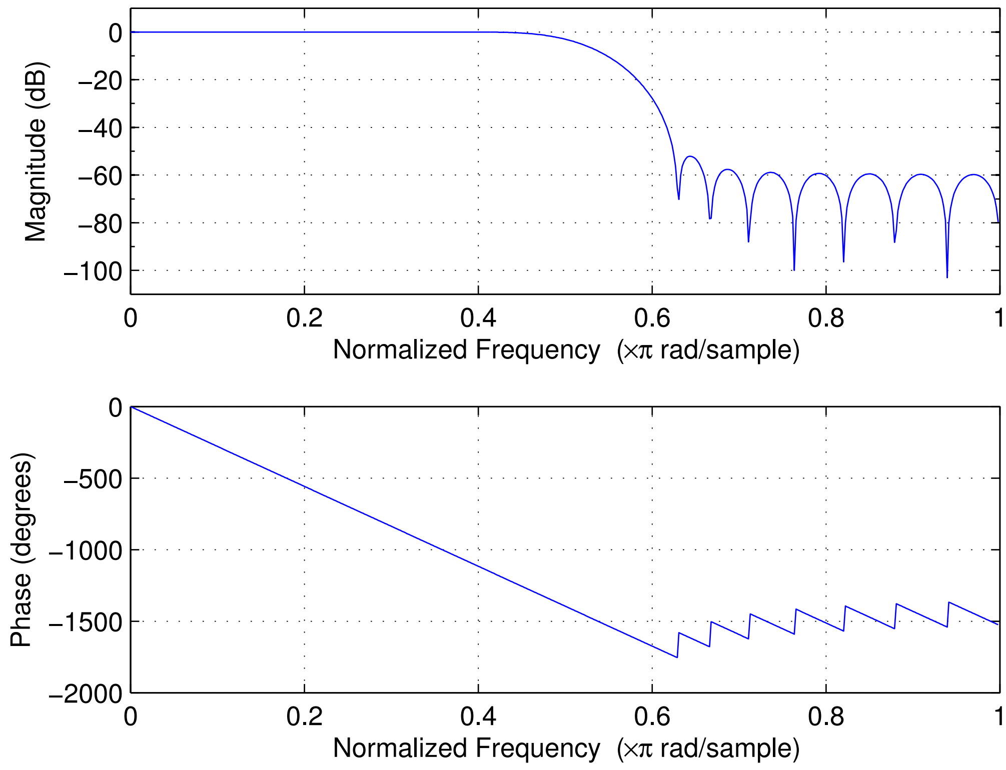

Figure 3.3: Filter properties of Yu 32 tap QMF

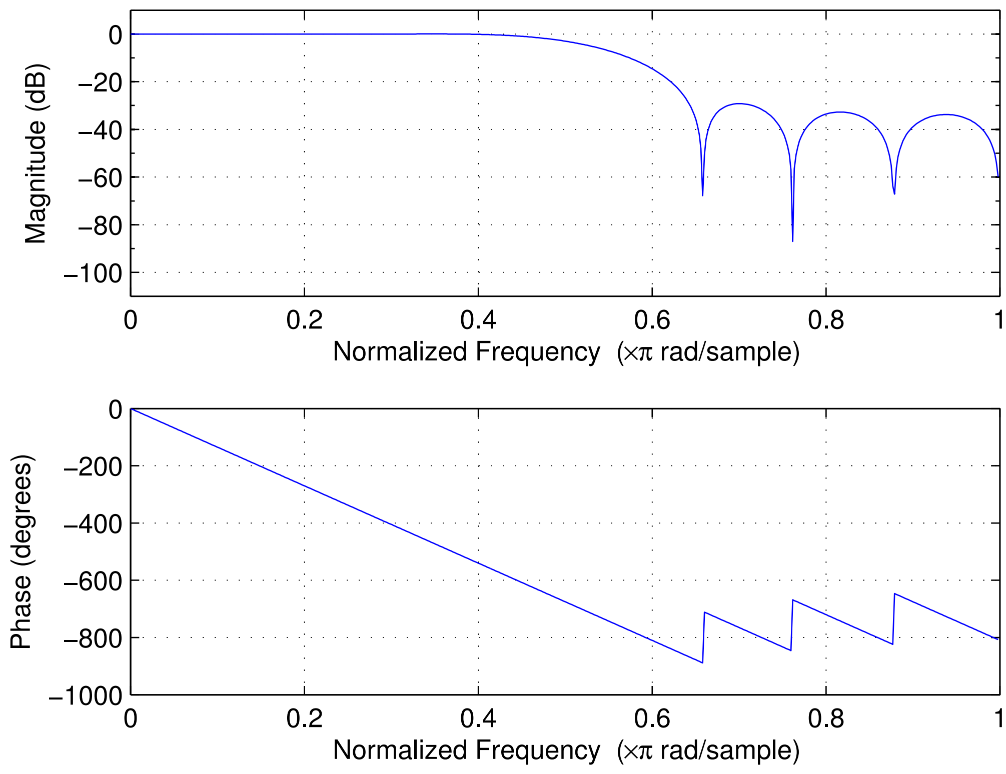

Figure 3.4: Filter properties of Johnston 16 tap QMF

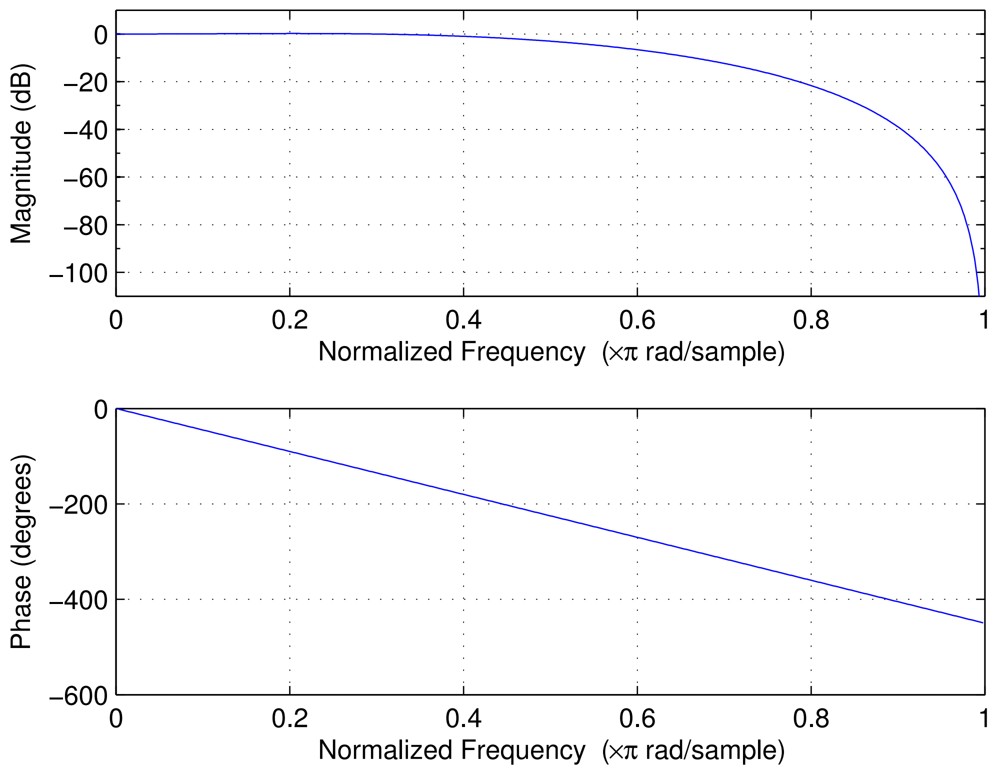

Figure 3.5: Filter properties of 6 tap 1.3 bi-orthogonal wavelet

Test results

We ran a series of tests on critical signals, switching between a 2048 point MDCT and MDWT, to see if this method reached its goals. The results are well-illustrated by figures 3.7 and 3.8, which represent a signal consisting of a few castanet claps encoded by both methods.

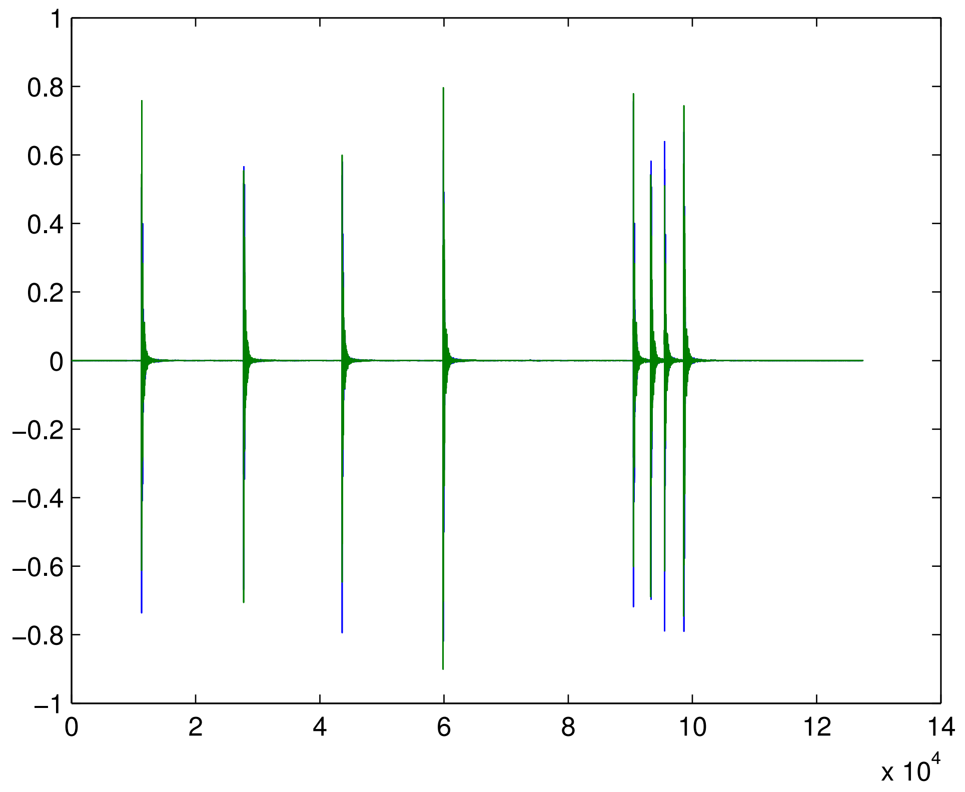

Figure 3.6: c44.wav (Castanet impulses sound sample)

It can be clearly seen that the MDCT encoded signal suffers very heavily from pre-echo, and this is in fact clearly audible. The MDWT performs much better and is only detectable by a trained listener.

However, further testing revealed issues when an impulse sample falls near the boundary of a block. In this case, the impulse might be coded with the MDWT, but due to the overlapping structure, it also risks being partially coded by an MDCT in the next block, still causing audible pre-echo. This is preventable by giving the encoder some lookahead, letting it switch to the MDWT in advance, and delaying the switch back to the MDCT, but this also means that if the signal surrounding the impulse is sinusoidal, we will lose coding efficiency. Furthermore, adding lookahead will add a significant

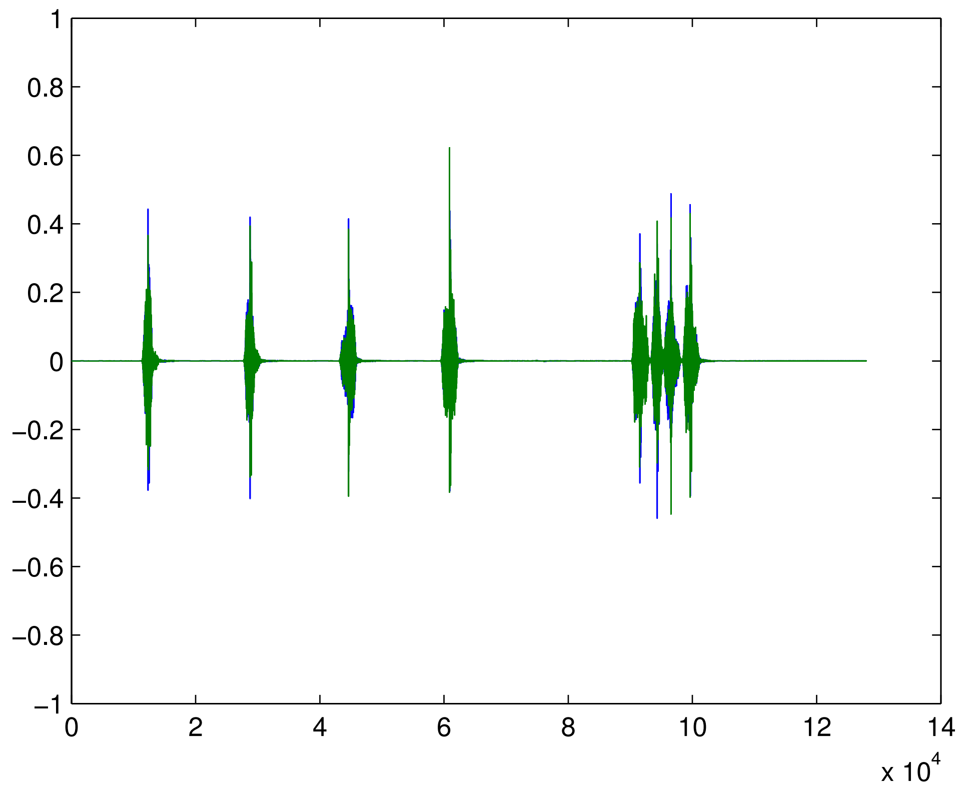

Figure 3.7: c44.wav encoded with a 2048 point MDCT

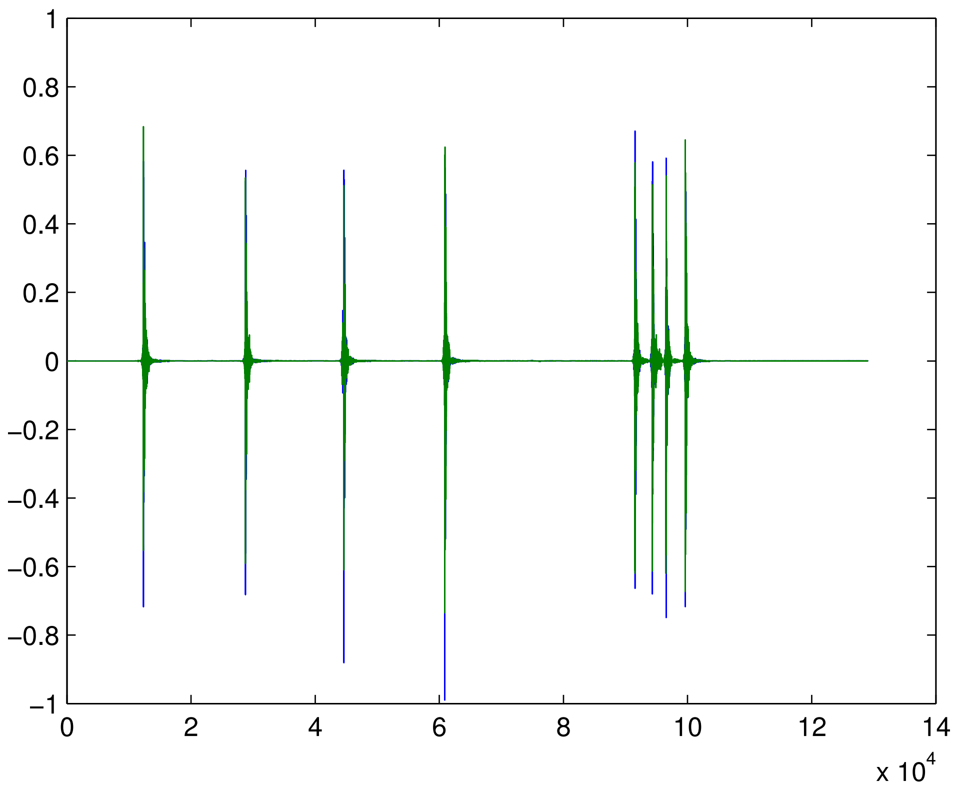

Figure 3.8: c44.wav encoded with a 2048 point MDWT

amount of complexity to the encoder.

A more serious problem is that to reach a 1 Bark resolution in the lower frequencies, we may need to split a 44.1kHz sampled signal up to 8 times ($22kHz \mapsto 11kHz \mapsto 5.5kHz \mapsto 2.7kHz \mapsto 1.4kHz \mapsto 700Hz \mapsto 350Hz \mapsto 170Hz \mapsto 86Hz$), causing the effective filter length for each sample to go up into the hundreds of taps, and causing the effective performance of the MDWT filterbank to be noticeably slower than that of an MDCT.

The number of MDWT blocks is low in typical music, so this problem could theoretically be solved without a large increase in required computing power by buffering a few blocks. Unfortunately, such buffering would increase the RAM requirements and is hence not really compatible with our design goals.

Another alternative is to simply not try to attain 1 Bark resolution, however this does mean that we cannot take full advantage of our psychoacoustic model, and so it is also not a very attractive option.

3.2.4 Temporal Noise Shaping

Temporal Noise Shaping was introduced in 1996 during the standardization of AAC as an alternate approach to block switching for handling the preecho problem.

AAC relies on the duality of the time and frequency domain representations in the DCT–4[4]. A sine signal in the time domain will show up as an isolated peak (pulse) in the frequency domain. Conversely, a pulse in the time domain will show up as a sinusoidal signal in the frequency domain. Sinusoidal signals can be efficiently predicted by linear prediction systems.

TNS makes use of this phenomenon by using Linear Prediction [22] in the frequency domain. An LPC [23] filter predicting the frequency coefficients is calculated via the Levinson-Durbin method, and this filter is slid over the high-frequency range of the MDCT spectrum, producing a residual signal. The resulting coefficients (the residue signal) are then treated normally by the remainder of the encoding process.

The goal of this procedure is not to attain a reduction of total quantization error, since we quantize the predicted coefficients with the same precision as the normal ones, but to exploit the fact that the quantization noise will now be shaped in time according to the energy envelope of the input signal[5]. By shaping the quantization noise this way, we ensure that low energy parts before an impulse will have very low quantization noise, whereas the quantization noise will be lot stronger during the impulse, but there it can be masked by the impulse itself. Note that we are using LPC in the opposite way in which it is typically used (e.g. in lossless or speech codecs). Normally, we use LPC prediction on the time domain samples to

[4]The same can be observed in a regular DFT.

[5]The derivation of this result can be found in [24].

predict a sinusoidal signal. In TNS, we use LPC prediction on the frequency domain coefficients to predict an impulse signal.

TNS is very flexible because we can select the part of the spectrum which we wish to shape, by simply limiting the range over which we apply the LPC filter. We can enable the filtering at any time we think it can produce a gain (typically detected by looking at the coding gain the LPC filter would produce). We just need to transmit whether or not if TNS is enabled, and if so, the filter length and quantized coefficients. A lot of theory regarding optimal LPC coefficient quantization exists due to its widespread application to speech, and we transmit the coefficients in their PARCOR/reflection coefficient form. The filter length is limited to 16 taps, as experimentation shows there is seldom a gain from using a higher order.

The application of TNS on the decoding side is limited to translating the reflection coefficients into their direct form, and running an all-pole IIR filter of a small order over the MDCT coefficients. No other part of the decoder needs to be aware of TNS.

The results of TNS on a strongly impulse sample can be seen in figure 3.9. It should be noted that typical music samples don't generally consist solely of impulses, but often have sinusoidal signals at the same time as well. In this case, the MDWT will suffer from a reduction of coding gain, while no such effect exists when using TNS, since we can limit it's application precisely to the frequencies where it can provide a gain. The only overhead is a small amount of side information that must be transmitted.

Because of all these advantages, and the low complexity, we chose to prefer TNS over the MDWT in the eventual implementation of our low-complexity codec.

The choice of the block length to apply the MDCT+TNS on is a trade-off between memory usage, processing requirements, coding gain, the amount of side information to transmit per second, and temporal resolution. Based on the results in [10] a block size of 2048 (1024 with 50% overlap) was initially used, but later lowered to 1024 samples to cut down as much as possible on the memory requirements. This was the smallest size that does not result in a large quality drop according to PEAQ testing.

3.2.5 Fast MDCT implementation

For our MDCT implementation, we chose a Kaiser-Bessel-Derived (KBD) window function with an alpha parameter of 5.0. The KBD window is used in the AC3 codec and is also one of the windows used in AAC, and gave marginally better performance in PEAQ testing as compared to sine windows.

For the computation of the MDCT, we follow the method given in [25], which decomposes a size N MDCT into a combination of N/2 pre-multiplications, a size N/4 complex FFT, and N/2 post-multiplications. The

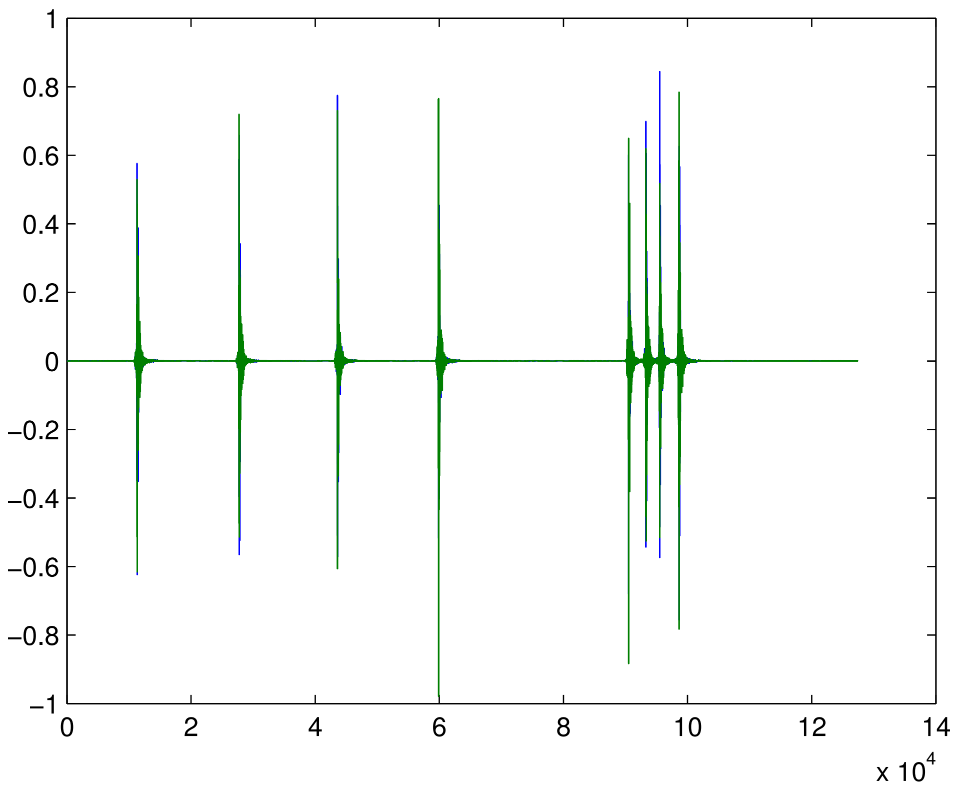

Figure 3.9: c44.wav encoded with a 2048 point MDCT + 16 tap TNS

problem of computing the MDCT quickly is hence reduced to computing an FFT quickly, which is a well understood problem.

For the computation of the FFT, we use a highly optimized Split-Radix FFT, based on the description in [26], including the optimization of separating the trivial (times 1 and –1) butterflies, and an additional optimization for the last pass, where we have eliminated most of the control flow.

For the bit reversal pass, we do not use the algorithm given in the above paper, but use a faster one as given in [27].

3.2.6 Spectrum partitioning

The most straightforward way to treat the spectrum in a manner consistent with the critical band scale is to partition up the spectrum in 1 critical band wide slices, or some multiple or division thereof ($\frac{1}{2}$ or $\frac{1}{3}$ Bark wide bands). For Bacon, we took advantage of the existence of the newer ERB scale and divided up the spectrum in bands that are consistent with ERB's instead of the older Barks.

Using a smaller subdivision allows a more precise control of our quantization process, but also causes a larger number of bits needed to transmit the side information for each band. We tried several alternatives at a fixed bitrate of 128kbps and eventually settled on bands that are 2 ERB wide.

Furthermore, we merged the bands above 16-17kHz since we will generally not code anything in that area anyway. The resulting partitioning (including the effect of rounding to the 512 spectral coefficients) is shown in table 3.2. The best configuration turned out to have 21 bands, which is the same number as used in MP3. This is quite little compared to AAC which has 49 scalefactor bands at 44.1kHz sampling rate. This could be caused by the fact that AAC uses a slightly more efficient side information encoding system than we do, but the large difference is still somewhat surprising. Unlike AAC or MP3 we do not limit our bands to multiples of 4, since it seems unlikely this would cause a worthwhile performance advantage on the ARM architecture.

3.3 Psychoacoustic model

The psychoacoustic model in Bacon applies a Hann window on the time domain samples of the block being considered, and then performs an FFT to get frequency and phase information.

Per band, we determine the total spectral energy. The energy is spread between bands with the spreading formula similar to the one in [28] and [11].

Next, we calculate the Spectral Flatness Measure (SFM) per band. This consists of dividing the geometric mean by the arithmetic mean of the power spectrum. This value is translated to a dB scale in the range of [0, –60 dB],

ERB	ending f. (Hz)	ERB	ending f. (Hz)
1	26	22	2212
2	55	23	2489
3	87	24	2798
4	123	25	3142
5	163	26	3525
6	208	27	3951
7	257	28	4426
8	312	29	4955
9	374	30	5544
10	442	31	6200
11	519	32	6930
12	603	33	7743
13	698	34	8649
14	803	35	9657
15	921	36	10781
16	1051	37	12031
17	1196	38	13424
18	1358	39	14975
19	1539	40	16702
20	1739	41	18625
21	1963	42	20767

Table 3.1: ERB Scale

band	start. f. (Hz)	start. MDCT coeff.	size
1	0	0	2
2	65	2	2
3	129	4	2
4	215	6	2
5	323	8	3
6	452	11	4
7	603	15	4
8	797	19	6
9	1055	25	7
10	1357	32	9
11	1744	44	11
12	2218	52	14
13	2799	66	17
14	3531	83	21
15	4436	104	26
16	5556	130	32
17	6934	162	40
18	8656	202	49
19	10788	251	62
20	13437	313	76
21	16710	389	123

Table 3.2: Bacon spectral partitions (SFB's) at 44.1kHz

and then converted to a tonality index from 0 to 1, with 0 dB SFM meaning a completely noisy band (=0), and –60 dB SFM meaning a completely tonal band (=1).

The acceptable noise thresholds are then calculated by determining the necessary Signal to Masker Ratio as follows:

$$SMR = fudge(b) + 25dB \times t + (6.56dB - 0.118dB \times b) \times (1 - t)$$

with b being the band number, t being the tonality, and $fudge(b)$ being a function that slightly increases the SMR around 1kHz and decreases it for higher frequencies. This fudge means that higher frequencies will start being distorted earlier than the speech range, a trick which was found experimentally to give better results. The noise energy threshold is found from combining the required SMR and the band energy.

During the spreading, the bands at the end only received additional energy from one side, meaning that the middle bands have incorrectly receive a relative gain compared to the outer bands. We now renormalize the bands to correct for this.

Lastly, we apply the ATH by determining the energy of the loudest band, and assuming that everything 100dB or more below that is not audible, with increases for the very high and very low frequencies.

3.4 Quantization

3.4.1 Mide/Side Coding

Before feeding the spectral values into the quantizers, we will optionally translate the Left & Right channel spectral values into Mid & Side representation, and rerun the psymodel.

The decision which representation to use is made per band, by looking at the difference in calculated masking threshold. When the channels seem similar enough, the masking thresholds of the lower bands are artificially lowered to prevent BMLD effects from happening.

For the Mid & Side representation, the worst-case is taken between the masking threshold from the BMLD corrected Mid/Side thresholds, and the threshold of the Left or Right channel.

3.4.2 Quantizer

Quantization in Bacon is handled via a scalefactor system combined with a fixed quantizer. The design quite closely follows the MP3 and AAC system. The scalefactors have an 1.5dB accuracy, and the spectral samples are multiplied by them, after which they are quantized by a fixed power-law quantizer of $y = round(x^{3/4} + constant)$.

The inverse relationship cannot very easily be calculated on the fly during the decode phase, so it useful to implement the dequantization as a table lookup instead. We made the decision to limit the max absolute quantized value to 512, or 9 + 1 bits, corresponding to a maximum SNR of 30dB. This table easily fits in a limited amount of RAM or ROM, contrary to AAC and MP3 who support values up to 8192. We feel that this is a worthwhile trade-off given that high SNR ratios are only needed for high quality encoding, which is not really the application domain of an inexpensive low complexity decoder, and only occur with very tonal signals, which are not very common to start with.

The problem for the quantization stage is that there are 21 scalefactors, each of which has 32 possible values[6]. This means that there are a total of 32^{21} possible quantizations. For selecting the optimal one, we have as input the output from the psychoacoustic model which tells us the maximum allowed distortion per band, and a given number of bits that we can spend to reach it.

3.4.3 The 2-loop system

The 2-loop system is included in both the ISO MP3 and the AAC specification as an example quantization algorithm, and consequently used in many encoders for both formats. It consists of two parts:

1. An Outer Loop which modifies individual scalefactors and hence modifies the current distortion (quantization noise).

2. An Inner Loop which modifies all scalefactors at once to reach the target bitrate (not modifying the relative distortions).

The encoder will start the outer iteration loop by decreasing all scalefactors with a distortion that is more than the threshold allowed by the psymodel. At each run of the inner iteration loop, all scalefactors are increased or decreased as appropriate until the bitrate goal is met. At that point, distortions are calculated and the next iteration of the outer loop runs. We repeat this process until there are no scalefactors to decrease, we have to decrease all scalefactors[7], or an upper iteration limit is reached.

There are some modifications that can be done to the ISO specifications to improve the quality of the results. One obvious one is to remember the quantization with the lowest total error, instead of the latest quantization that was tried. This gave as serious improvement in Bacon. Others are to only decrease one or the most few distorted at a time. These were found to make little or no difference.

[6]The origin of this value is explained in the lossless coding section.

[7]Which, due to the inner loop, is equivalent to decreasing none.

The 2 loops system works, and can be used for both constant bitrate and constant quality encoding (by varying the target bitrate to match the demands from the psymodel). However, it also has some very serious issues. Firstly, because there are usually many quantizations to try, the entire spectrum has to be requantized several times, resulting in a large performance penalty. Secondly, if the demands from the psymodel are far too low or too high for the currently available bit demand, the output will be far from optimal. For example, if there are many more bits available than needed, the system will return instantly as all goals will have been met. However, the system will have given all bands roughly the same SNR, which means that bands which needed a high SNR will have a much lower security margins than bands that did not. Conversely, when there are not enough bits available, the system minimizes the total error, which might not be the best solution, perceptually speaking.

3.4.4 Scalefactor Estimation

Scalefactor Estimation is based on the observation that we know the required SNR per band directly from the psymodel, which means that we can give a good estimation for the scalefactors needed to put the distortion per band below the threshold. For each decrease of a scalefactor by 1, the amplitude of the values going into the quantizer increases by 1.5dB, and taking into account the pow-law function, the real increase in SMR is about 1.125dB. From this we can calculate the relative values of the scalefactors. The absolute values can be derived from a system similar to the inner loop.

The main advantage of this system is that it is much faster than the 2 loops system are there are a lot less quantizations to be tried. The disadvantage is that the quantization can be less than optimal, because we cannot detect situations when we are lucky or unlucky with quantizing, and have a bigger or smaller distortion in a band than what we estimated.

3.5 Lossless coding

For the lossless coding step, we rely on 2 methods: differential encoding and huffman encoding.

3.5.1 Differential encoding

The differential encoding is used when we transmit the side information consisting of the scalefactors, and the bit length of the quantized values. The latter needs to be transmitted because there are separate coding strategies for different quantized value ranges.

We transmit the initial value verbatim, and then transmit only the difference between the next value and the previous value for subsequent values.

Because we don't expect large, sudden variances between subsequent bands, these are generally small and often zero, and can be efficiently encoded with Huffman coding.

3.5.2 Huffman encoding

Before starting the Huffman encoding, we look at the bit length of the largest quantized value. If this is 6 or more bits (not including the sign), we do not apply Huffman coding, but simply transmit the value followed by a sign bit (if the value was not zero). Because 6 bits correspond to a highly tonal band (SMR of 21dB), and requires a large amount of bits which is generally not available, we do not expect this case often and this simplified encoding saves time, memory and complexity.

For lengths of 3, 4 and 5 bits, individual values are Huffman coded and transmitted one by one. For a length of 0, nothing is stored.

For lengths of 1 and 2 bits, the spectrum is split in groups of 1, 2 and 3 values, and group Huffman coding is used. Each possible combination of values is assigned a single value and this value is Huffman coded.

The differential codes for the scalefactors and the quantized value lengths have their own, separate Huffman tables.

The Huffman tables were generated by running the encoder over a wide variety of inputs and bitrates, logging the quantized values, and then running a custom tool that constructs an optimal Huffman code for the signal statistics.

For fast decoding, we use a tree representation compactly stuffed in a 2-dimensional array. The tree is walked by putting the current tree index and the read value in the array, and reading the next index. If the index is greater than 64, we are at a leaf node and the decoded value equals the index minus 100. If it is not, we need to read the next bit and try again. The tree representation was obtained by manually processing the Huffman codes in a spreadsheet.

3.6 Additional tools

3.6.1 PNS

For bands which are classified by the psychoacoustic model as mostly noisy, we allow replacing the entire quantized band by a single 8-bit value, which represents the band energy quantized to 0.75dB. This is essentially the same system as used by the PNS tool in AAC.

A PNS band is marked as having a quantized value bit length of -1, and does not have an associated scalefactor.

3.6.2 LastVal

A simple trick borrowed from MP3, to gain some additional efficiency, is to store the last spectral line that is different from 0 after quantization, as one of the first values in the bitstream.

We can then simply not store any scalefactors, quantization lengths, and mid/side decision information for the bands that start after this point, as well as avoid coding a lot of groups of zero values in the spectral information.

Some simple testing showed a savings of 2–3kbps at 128kbps bitrate, with more expected for lower quality encodings.

3.7 Fixed point version

The initial version of our codec was developed entirely using 32-bit IEEE float arithmetic, for easy testing and speed of development. Upon statisfactory completion of the design, we set out to port the decoder to fixed point arithmetic suitable for use on the ARM.

We based our fixed point port on the hints given in [29], and made thankful use of the given C macro's, which allow one to simply replace all arithmetic operations in the program with the given equivalents.

By porting function by function it is quite easy to isolate errors in the fixed point versions of the routines. The fact that we can run the calculations in 32 bits also meant that we did not have to worry too much about loss of precision. The FFT was the only thing that needed some tweaking, since it loses 8 additional bits of precision if we use it with our N=256. The irregular structure of the Split–Radix butterflies makes it inconvenient to do the scaling in the FFT itself. We ended up doing a scaling of 1 bit in the last pass, which, as explained earlier, was separated anyway for performance reasons, and doing the rest of the scaling in the pre- and post-multiplications of the MDCT-via-FFT routine.

3.7.1 Number classes

To achieve a good trade-off between ROM usage and accuracy in the fixed point version, we define two different kinds of fixed point numbers, COEFF's and REAL's.

COEFF's are numbers with a large fractional part, and a small integer part, meant to be used to store coefficients in the range of [–1, 1]. REAL's have a small fractional part and a large integer part, and are mostly used to store samples.

In the case of Bacon, if we want to pack all our constants in 16 bit values, we can use up to 14 fractional bits for COEFF's and up to 4 fractional bits for REAL's. The limits for COEFF are determined by the need for 1 sign bit and 1 extra bit[8], leaving 14, whereas the limits for REAL is that it must be able to represent the maximum quantizer value, which is 4096, taking 12 bits, plus a sign bit, making 13 and leaving 3 fractional ones.

Experimental results seemed to indicate that packing the constants in 16 bits does not seem to result in an audible reduction of playback quality, at the gain of very significantly reduced ROM and chip bandwidth requirements.

The most extreme possibility is to pack REAL's with no fractional part, and COEFF's with only 9 bits. This allows the decoder to still produce reasonable output, while at the same time no longer requiring the possibility to do a 32 x 32 = 64 bit widening multiply. Given that the ARM has this instruction, this seems senseless, but there might nevertheless be situations in which having the possibility to work like this is handy. At the very least, it allows slightly faster decoding at a loss of audible quality.

3.7.2 Smart multiplying

One major issue for doing the fixed point version entirely in C is that we have no direct way to tell the compiler we need to generate a 32 bits x 32 bits = 64 bits multiplication. If we multiply two integers (32-bit), the result should be expected to overflow since it won't fit in a 32 bit result. If we multiply two 64-bit integers, there is no native corresponding ARM instruction and we end up with a very slow library call. We originally used an inline assembly language routine, but this turned out to be inconvenient when we wanted to compile selected parts of the Bacon decoder in Thumb mode. The eventual solution was to use the following routines:

```
static const __inline int MUL_C(const int A, const int B)
{
    return (((long long)A) * B) >> COEF_BITS;
}

static const __inline int MUL_R(const int A, const int B)
{
    return (((long long)A) * B) >> REAL_BITS;
}
```

which allow a smart enough compiler[9] to correctly infer that 32 x 32 = 64 multiplies need to be generated. Note that the shift ensures that the output value is in the same format as the input value, with the other one being a COEFF/REAL respectively.

[8]Caused by the range being inclusive of 1.0 itself. In retrospect it might have been better to "round" all 1.0 values to 0.9999... instead.

[9]In casu, GCC 3.4.4. We had less success with the Microsoft Visual C++ 2005 compiler, which generated a library call.

3.8 Performance evaluation

We will now present some performance results of the Bacon codec, consisting of encoder and decoder performance on an Intel PC platform, decoder performance on an embedded ARM9EJS platform, memory requirements for the decoder, as well as quality evaluations via PEAQ, compared to ISO MP3 and ISO AAC codecs and the free, open sourced Ogg Vorbis codec.

3.8.1 Processing requirements

PC encoding

For testing the encoding performance, we used an Intel Centrino 1.7Ghz laptop, encoding the first 30 seconds of Peccatum — No Title for a Cause to 128kbps CBR. This is a metal song which has a fair amount of variation between instrumental, vocal and guitar parts. We repeated the encoding 3 times and noted down the average times.

For the comparison, we used the following encoder versions, all of which are freely available with sources, with the given settings:

- Bacon: SV2 Ref4.2 (ERB) with "-b 128" option
- MP3: LAME 3.97 beta 2 with "-b 128" option
- MP3: Real Helix v5.1 with "-B64" option
- AAC: FAAC 1.24.1 with "-b 128" option
- Vorbis: Xiph.Org 1.1.2 with "-b 128 –managed" option

Encoder	Times realtime
Helix	75x
LAME	15.32x
FAAC	11.28x
Bacon	10.43x
Vorbis	2.43x

Table 3.3: PC encoder speeds for 128kbps CBR

It can be seen that Bacon's processing requirements are in the same ballpark as those of existing encoders. The speed of the Helix MP3 encoder is interesting, as well as the slowness of Vorbis in CBR mode. When allowed to use VBR mode, the Vorbis encoder sped up to 9.7x.

PC decoding

For the decoding speed test, we used the freeware foobar2000 0.9.1 audio player with the decoding speed test component, set to: repeat 10 times, highest priority, buffer entire file in memory. We decoded the file encoded in the previous benchmark.

Decoder	Times realtime
Bacon	195x
AAC	172x
Vorbis	153x
MP3	130x

Table 3.4: PC decoder speeds for 128kbps CBR

The low complexity design of Bacon allows it to pull ahead of the pack here. The speed of the MP3 decoding is surprisingly low and might point to a not particularly heavily optimized decoder being used by foobar2000. For AAC, we know that foobar2000 relies on the widely used open source FAAD2 decoder, and for Vorbis, it uses the Xiph.Org reference floating point decoder, which is also very commonly deployed.

ARM decoding

We benchmarked the fixed point version of the Bacon decoder against fixed point decoders for the other formats mentioned above. For MP3 we used “MAD” from Underbit Technologies, Inc., for Vorbis we used the “Tremor” decoder from Xiph.org, and for AAC we again used “FAAD2”, but now running in fixed-point mode. Sources for all three decoders are freely available.

Most of these decoders include pieces of ARM-specific hand-optimized assembler code. In the case the assembler code is disabled, there is, in the case of Vorbis and MP3, a choice whether to allow the decoder to run in reduced precision mode, at the cost of a lower audible quality. We tested all three configurations.

The reason for the existence of these reduced precision modes is again the 32 x 32 = 64 bit multiplication. Without assembler code to handle this case, some decoders incur a significant performance penalty, as can be seen in the table. The reduced precision modes avoid this penalty by not using the widening multiplies altogether, at a cost of audio quality. For Bacon the difference is much smaller due to the trick described in section 3.7.2.

We also tested the performance of the Bacon decoder running in Thumb mode. The results were quite surprising, in the sense that the Thumb decoder was only slightly slower than the ARM one. Reasons for this might be the existence of a small code cache on the OMAP5912, in which much more Thumb than ARM instructions can fit.

The performance numbers given are for the decoding of a 30 seconds long 128kbps CBR sample. Both the "number of times faster than realtime" and the calculated minimal OMAP CPU Mhz needed to run realtime are given. They include the time needed to read the input data from a buffer and to write the output data to another buffer, and are directly obtained from running the decoders on real hardware, in casu the OMAP5912 OSK. All tests were compiled with GCC 3.4.4 at "-O3 -mcpu=arm926ejs" settings.

Decoder and operating mode	×Realtime	min. CPU Mhz
Bacon, full precision	4.40x	43.64Mhz
Bacon, reduced precision	4.80x	40.02Mhz
Bacon, reduced precision (Thumb)	4.45x	43.13Mhz
MAD, ARM assembler mode	4.86x	39.52Mhz
MAD, full precision	0.65x	297.13Mhz
MAD, reduced precision	4.20x	45.68Mhz
Tremor, ARM assembler mode	4.51x	42.60Mhz
Tremor, full precision	1.86x	103.28Mhz
Tremor, reduced precision	4.85x	39.62Mhz
FAAD2, ARM assembler mode	3.30x	58.16Mhz
FAAD2, full precision	3.25x	59.08Mhz

Table 3.5: ARM decoder speeds for 128kbps CBR

As can be seen, Bacon provides performance similar to the fastest decoders and does not need assembler code to give acceptable performance and full precision at the same time.

3.8.2 Memory requirements

While the ROM and code size of the decoder can be easily determined from the compiler output, determining the RAM requirements is more tricky, and we did not find a suitable tool to determine this automatically for ARM. The figures given are based on an analysis of the largest known resident memory users, and the stack usage of the critical functions. This is necessarily an estimate because the actual usage can vary depending on how the compiler allocates it's registers, i.e. whether it stores some variables on the stack or keeps them around.

For comparison, we have the claimed memory usage of the Real Helix AAC and MP3 decoders, taken from their website. For MAD, the publisher does not give memory usage estimates, nor did he answer our inquiries, so the given number is a rough estimate. The numbers for "Tremor" are also surprisingly hard to find, despite (or because of?) the fact that this a key problem point for Vorbis implementers. The RAM data was gathered from posts on the Vorbis mailing lists, and the code size and ROM data were determined from the compiler output, as with Bacon. We used the "lowmem" version of the Tremor decoder, which is supposed to have reduced requirements as compared to the normal version. All numbers given pertain to stereo decoding and are in bytes.

Decoder	Code size	RAM	ROM
Bacon (Thumb)	5828	11516	13888
Bacon (ARM)	7960	11516	13888
Helix MP3 (ARM)	±21000	23816	13446
Helix LC-AAC (ARM)	20120	28864	27128
Vorbis Tremor (ARM)	35952	±81920	48428
MAD MP3 (ARM)	36184	±50000	50688

Table 3.6: ARM decoder memory usage

It can be clearly seen in table 3.6 that Bacon has a significantly smaller code size and memory footprint than any other decoder, and is hence much more suited for usage on small chips than any of the other existing psychoacoustic audio codecs.

3.8.3 Quality

For quality evaluations, we used the following set of songs given in table 3.7, with the first 20 to 30 seconds extracted. Because the PEAQ tools require a sampling rate of 48kHz, all source material was resampled with Adobe Audition 1.5 set to the highest quality level.

They have been chosen to cover a wide variety of musical styles, and because they have all been previously used for public listening tests. Most of them can be downloaded at the following location:

`http://www.rarewares.org/test_samples/`

We used the encoders with the previously given settings, and ran the PQEvalAudio PEAQ tool on them. The Helix and FAAC encoders introduce an additional delay into the decoded output, and a tweak to PQEvalAudio, namely skipping a given number of samples, dependent on the encoder, was needed to get correct results. The Vorbis encoder consistently produced files that were 2kbps too big, so we used the setting "–managed -b 126", which did give correct 128kbps output.

We also included the freely available NeroDigital AAC encoder, released May 1st 2006, to get a comparison to the current "state of the art" in ISO codecs, as well as a version of Bacon with the MDCT+TNS replaced by an MDWT with the same blocksize.

The results are given in table 3.8 and a box-and-whisker plot is given in figure 3.10.

Sample name	Title — Artist — Album
class.wav	Danse Hongroise 6 — Johannes Brahms — Danses Hongroises
disco.wav	Enola Gay — OMD — Organisation
floor.wav	Floor-essence — Man With No Name — CD Single
latin.wav	La voz de la experiencia — India (feat. Celia Cruz) Sobre el fuego
meta.wav	Gone — Die Apokalyptischen Reiter — All you need is Love
mozart.wav	Der Hölle Rache kocht in meinem Herzen — Mozart Die Zauberflöte (Act 2)
elec.wav	Coitus (remix) — Green Velvet — Green Velvet
saints.wav	Never Ever — All Saints — Hit Club 98
after.wav	No Title for a Cause — Peccatum — Amor Fati
voice.wav	It Could Be Sweet — Portishead — Dummy
wait.wav	Waiting — Green Day — Warning
jesus.wav	Put your trust in Jesus — The Harlem Gospel Singers Live at the Cologne Philharmonic Hall
oworld.wav	Ordinary World — Duran Duran – Greatest
scars.wav	Scars Left by Time — Yasunori Mitsuda Chrono Cross Original Soundtrack
vega.wav	Rosemary — Suzanne Vega The Best of Suzanne Vega (Tried and True)
death2.wav	Death To All Culture Snitches — Two Lone Swordsmen Tiny Reminders

Table 3.7: Quality test samples

Sample	Bacon	LAME	Helix	Vorbis	FAAC	Nero	MDWT
class	-0.623	-0.195	-0.467	-0.105	-0.160	-0.047	-1.447
disco	-0.888	-1.017	-1.095	-0.307	-0.718	-0.270	-1.019
floor	-0.963	-1.358	-0.946	-0.368	-0.635	-0.369	-1.087
latin	-0.370	-0.563	-0.594	-0.216	-0.453	-0.172	-0.572
meta	-0.877	-0.849	-0.893	-0.266	-0.553	-0.228	-1.034
mozart	-0.979	-0.867	-0.855	-0.318	-1.131	-0.175	-1.620
elec	-0.377	-0.982	-0.984	-0.301	-0.471	-0.304	-1.501
saints	-0.520	-0.748	-0.813	-0.256	-0.894	-0.176	-1.020
after	-0.518	-0.665	-0.683	-0.217	-0.539	-0.172	-0.732
voice	-0.157	-0.243	-0.111	-0.104	-0.819	+0.119	-0.244
wait	-0.566	-1.216	-1.262	-0.344	-0.732	-0.301	-1.662
jesus	-0.903	-1.013	-1.118	-0.395	-0.553	-0.326	-1.032
oworld	-0.880	-1.092	-1.141	-0.315	-0.703	-0.288	-1.148
scars	-0.911	-1.046	-1.006	-0.334	-0.669	-0.337	-1.303
vega	-0.607	-0.902	-0.860	-0.395	-0.838	-0.309	-0.818
death2	-0.588	-0.797	-0.215	-0.191	-0.793	-0.050	-1.476
Average	-0.670	-0.847	-0.815	-0.277	-0.666	-0.213	-1.107

Table 3.8: PEAQ Basic Model quality test results

Although we have to be cautious from inferring too much, because the PEAQ tools are not 100% reliable, Bacon is clearly very competitive even with the best MP3 encoders, and ties the open source AAC encoder FAAC. However, it is no match for both Vorbis and state of the art AAC encoders. The Wavelet-based version is clearly worse than the MDCT-based one, although it still manages to deliver a quite usable quality level.

3.9 Conclusions & future research

From these results it can be seen that Bacon achieved it's design goals. It provides a quality level at least competitive with MP3, at a significant reduction in RAM and ROM requirements, not to mention decoder complexity: a decoder will fit in less than 2000 ARM instructions.

Nevertheless, we want to note some areas where improvement is possible:

- We wrote the code entirely in C emphasizing on maximal portability, readability and ease of experimentation. Some optimizations could particularly speed up the codec on ARM, at the cost of readability, for example rewriting some of the core FFT in ARM assembler, and rewriting critical routines to use pointer arithmetic instead of array indexing, so the availability of pre/postincrements on ARM is utilized.
- The codec is heavily based on ISO MP3 and ISO AAC, mostly due

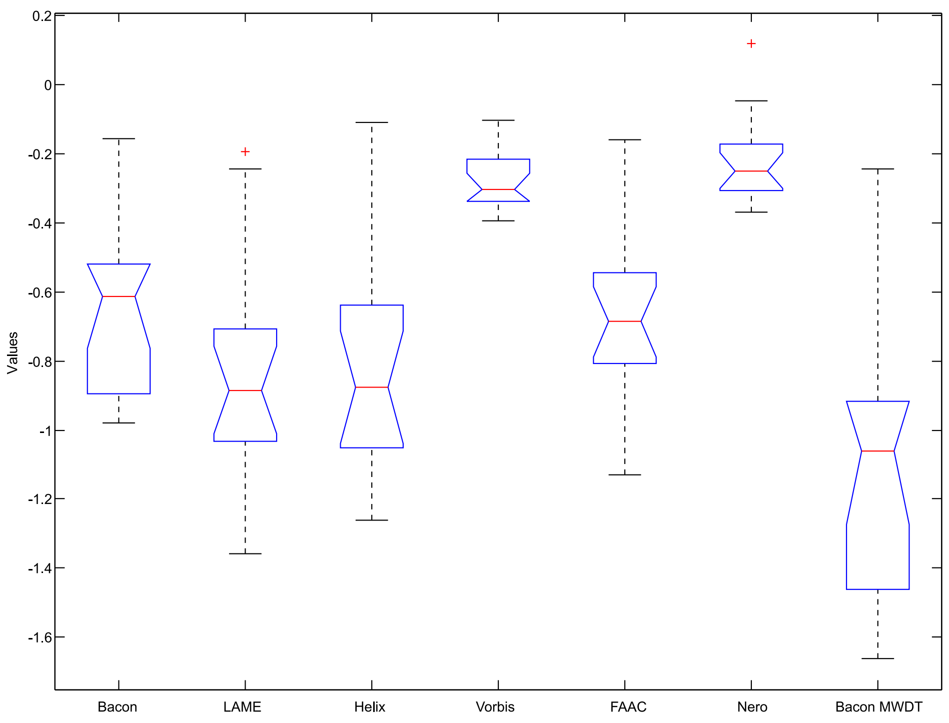

Figure 3.10: PEAQ Basic Model quality test results

to the availability of good explanations and specifications for those formats. In retrospect, adopting some features of Vorbis might allow increased performance at a minimal cost in complexity, notably the piecewise linear floor and the way this is used to allow intensity stereo coding.

- After implementing it and testing with it, we started to severely distrust the 2-loop system. It looks like a very inefficient method, and does not work well when the goals of the psymodel cannot be reached or are too easily reached. An improved algorithm based on Scalefactor Estimation would almost certainly be better.

- It might be worthwhile to try grouped Huffman encoding on the side information such as Mid/Side decisions, quantizer bit lengths and scalefactor information.

- The psychoacoustic model can be further improved. It currently does not account for postmasking. One could experiment with non-linear psymodels as well.

Another conclusion to be made is that designing and writing a complete encoder and decoder for sure is an excellent way to increase one's understanding of the field by at least several orders of magnitude.

Chapter 4

An ARM-based Bacon audio player

For demonstrating and testing the Bacon codec, we used two different platforms. One was a ready-made development board built by Texas Instruments and running Linux. The second one was an own design built around an inexpensive Flash based embedded ARM microcontroller.

4.1 The high end - TI OMAP5912 OSK

4.1.1 About the OSK

The OMAP range of chips consists of ARM cores coupled with Texas Instruments DSP cores, and optionally on-chip accelerators for audio and video decoding, controlling LCD screens, and so on. They are extremely widely used in all kinds of modern GSM phones, PDA's and GPS devices.

Texas Instruments offers two ranges of OMAP chips. The "Wireless Handset & PDA Solutions" range is only available to high-volume customers such as OEM's and ODM's, and hence not available to us. The "Portable Data Terminal Solutions" range is normally available through Texas Instruments' online shop or through electronic device distributors such as DigiKey. The OMAP5912 chip belongs to the second range and retail for 26.15$ per thousand, or about 32$ to 50$ for single unit purchases.

Texas Instruments together with Spectrum Digital offer a development board based around the OMAP5912, called the OMAP Starter Kit (OSK), which retails for 295$. The exact specifications of the OSK are:

- ARM926TEJ core running at 192 Mhz.
- TMS320C55xx DSP core running at 192 Mhz.
- 32 Mbyte DDR RAM

- 32 Mbyte NOR Flash ROM
- TLV320AIC23 24-bit 96kHz DAC/ADC
- RS-232 serial port
- 10 Mbps Ethernet port
- USB port
- CompactFlash socket, suitable for type I or II cards

The combination of a high quality DAC coupled with the CompactFlash socket, fit for mass storage of encoded Bacon files, make it very suitable to our purposes. There is only one bad thing to say about the package: the cheapest available JTAG connector available from Texas Instruments which can be used for on-chip debugging the OMAP costs as much as 1500$! Luckily, we are not going to be working on the operating system so software-only debugging is sufficient to get Bacon running.

4.1.2 The ARM Linux kernel

The OSK is delivered with MontaVista Linux installed, a Linux version specifically adopted to embedded platforms. However, it is based on the older Linux 2.4 kernel. On their "TI Linux Community" website, Texas Instruments offers instructions, a version of GCC suitable for cross-compiling to the ARM, and prebuilt newer kernel images. Alternatively, one can fetch the very latest Linux/ARM kernel directly from the source control repositories, or download just the ARM patches for a given stable Linux kernel version. We chose the latter option. We got a GCC crosscompiler from the TI website, fetched the latest public Linux kernel and applied the corresponding patchset.

The Linux ARM kernel is pretty well developed and contains some very interesting features such as the Journalled Flash Filesystem 2 (JFFS2), which allows one to use the NOR Flash ROM as if it were a normal, writable filesystem. The kernel then takes care the data is written to the Flash in such a manner that maximizes the lifetime of the Flash ROM.

4.1.3 Host software requirements

It was not noted above, but all of the crosscompiling or debugging tools assume that the host platform for development is an x86 platform that is itself also running Linux. For transferring new kernels of filesystems to the OSK, we need to run a TFTP and NFS server on the host, and it is handy to run a DHCP server as well. This kind of setup makes it possible to boot the OSK with as the root Linux filesystem a directory on the remote machine.

This way, we can simply copy new software to this directly, and it will be available to the OSK board instantly.

We found VMWare Workstation to be a very handy tool for this kind of development need. It allows us to read websites, manuals and datasheets, as well as operate our favorite editor or IDE in Windows, while having a separate virtual Linux machine running in a window. With the help of a standalone router and the "bridged networking" support in VMWare, the virtual Linux machine in VMWare looks like a real machine for all intents or purposes, to any other machine on the network, in our case the OSK.

4.1.4 Targeting the OSK

Getting an application built for Linux ARM is generally as simple as recompiling it with the CC environment variable pointed to the GCC ARM crosscompiler. Any failures would generally be caused by the usage of non-portable or not properly ANSI C compliant code, which was not a problem for Bacon.

Strictly speaking, there is not even a need to port the application to fixed point — the Linux kernel includes a floating point emulator. Of course, performance will be really bad, but if the floating point routines are not time-critical to the application this will not be an issue.

4.1.5 Buildroot

Although our OSK was now equipped with the most recent Linux kernel, we wanted to add a few more utilities to the filesystem, and if possible obtain a crosscompiler based on a more recent GCC version. The most interesting system for doing this appears to be "buildroot". The webpage describes it as:

> Buildroot is a set of Makefiles and patches that makes it easy generate a cross-compilation toolchain and root filesystem for your target Linux system using the uClibc C library.

The configuration system allows one to choose the ARM target, select the wanted GCC crosscompiler and GNU Debugger version, choose a large amount of the standard Unix tools, and directly generate a JFFS2 image or .tar file with everything readily set up as a root filesystem.

Unfortunately, reality was that buildroot seems to contain a significant number of bugs, and certain of the tools can't be built with certain choices of compiler, options, and/or the wanted target platform (ARM). We eventually did succeed in getting buildroot to complete, but only having to reduce the number of selected software packages a bit, and with some manual fixing of others where it looked doable to do so.

The main reason to go through these pains was to get some more flexibility for the eventual use of the system. For example, our OSK now runs a fairly complete Linux 2.6 system, including Secure Shell (SSH) and DHCP, which allows us to plug in the board in any given network and access it as if it were a regular Linux system. The GNU Debugger is now also fully functional in both normal and client-server operation. Lastly, we can use Bash shell-scripts which feed our Bacon player with all files from a given directory, for a simple playlist-like functionality.

4.1.6 Audio output with OSS

For audio output on a Linux 2.6 kernel, two methods are available: ALSA and OSS. We chose to use OSS because it looked like the most simple method, and because a good manual was available [30].

Using OSS consists of opening a Linux device file, setting several parameters with the use of ioctl's, and then writing to the device file. We used two freely available console based mixer programs, called 'lmixer' and 'smixer', which we crosscompiled for ARM, to change the volume settings.

4.1.7 Some remarks on difficulty

The main thing to consider when using the OSK as a development board is that for all intents and purposes it is first and foremost based on a Linux system. If one is familiar enough with handling such systems in-depth, then the OSK works as any other system would, is as “easy” to use, massively extendable, and presents a straightforward learning curve. The main problems occur when supporting packages are buggy, or when documentation is out of date. We had our fair share of problems with both.

If one is not intimately familiar with the usage and administration of Linux systems, and terms such as TFTP or rebuilding the kernel are alien, one will be much more limited in the use of the OSK, though the board as shipped does contain everything needed to get basic applications tested, if only against a slightly out of data system kernel. The only thing that then one then really needs to learn is how to install and use a GCC based crosscompiler.

4.2 The low end - Atmel AT91SAM7S

A fairly new appearance on the market are low-cost microcontrollers built around ARM cores combined with Flash memory. They are offered by multiple suppliers in multiple models, such as the Philips LPC21xx, the Atmel AT91 and the Analog Devices ADuC7xxx. Typically offered with a large variety of peripheral controllers and ports, and running at clock speeds in excess of 50Mhz while still consuming a minimal amount of power, they are making inroads against smaller, 8 or 16-bit based microcontrollers in applications where the extra computing power can be put to good use.

One such application is portable audio playback. An ARM7TDMI processor at 30Mhz is fast enough to decode CD quality stereo audio from a perceptual audio codec, removing the need to add a separate decoding chip. The main problem when trying to use such a microcontroller for this purpose are the high RAM requirements for decoding, which limits one's options for choosing a chip to the very top end, and consequently most expensive models. But by using Bacon as our audio codec, we can comfortably use a smaller chip, saving per-device costs.

4.2.1 Choosing a CPU

Memory space

We investigated the offerings from Philips, Atmel and Analog Devices. The Analog Devices chips had to be discarded because they were not available with RAM sizes of more than 8k, which is too small even for Bacon. This left the Philips LPC series and the Atmel AT91 as contenders.

Flash performance

We tended to favor the Philips LPC chips since by most reports they give superior performance when running from Flash. When running from Flash, most of the Flash interfaces cannot retrieve code or instructions quickly enough when the chip is running at high speeds. To resolve this issue somewhat, the Flash based ARM chips use a prefetching mechanism to try to increase the performance. Of course, this fails in loops or with unpredictable branches, and the quality of the prefetches can determine the chips' real performance to some extent. For the same reasons, it is beneficial to use Thumb mode when running some of these chips at the highest speeds, since it will reduce the needed memory bandwidth for instruction fetching by half.

I^2S and DACs

A search of available suitable stereo CD quality DAC quickly revealed that almost all of these rely on an I^2S bus interface, which is a high speed serial protocol. We did not find any capability on the Philips LPC controller to control such a bus, whereas the Atmel do contain a Serial Synchronous Controller (SSC) suitable for this, and Atmel provides an application note regard the usage of the SSC to control an I^2S bus. This steered us toward our final choice of the Atmel AT91SAM7S to design our board around. The various AT91SAM7 chips and their prices are listed in table 4.1.

AT91SAM7S options

The AT91SAM7S64 is the smallest chip that can house a Bacon decoder. For the purposes of convenience, we used the most powerful chip, the AT91SAM7S256, when we built our board. The two chips do not differ aside from the memory sizes and are trivially interchangeable.

Model	RAM	ROM	Price (> 100 units)
AT91SAM7S32	8K	32K	3.74€
AT91SAM7S64	16K	64K	4.91€
AT91SAM7S128	32K	128K	7.26€
AT91SAM7S256	64K	256K	9.09€

Table 4.1: The AT91SAM7 range

4.2.2 Getting the input

Normally one would equip a portable audio player with some kind of Secure Digital/ Multi Media Card (SD/MMC) or CompactFlash based storage, built around a FAT filesystem. As this was only a development model, we decided to sidestep the complexity of interfacing with these cards, and the associated filesystem, for now, and instead rely on simple streaming over the serial port. With standard hardware, transfer rates of 115200 bits per second are possible over the built-in UART, and this is enough to get high quality audio.

In the meantime, Atmel and other parties have published a lot of code related to interfacing with such Flash based storage cards, so a new design should probably consider this option more seriously.

4.2.3 Playing the output

As explained earlier, nearly all DACs suitable for CD quality stereo output rely on the I^{2}S protocol. One option was to use the cheap and highly capable TLV320AIC23 chip, the same as used on the OSK board. We eventually settled on the Cirrus Logic CS4334 because it also combines high quality with a minimal complexity in board layout and required application logic. The chip does have requirements on the clock with which it is steered. One suitable clock is 18.4320Mhz, which is the same crystal that is needed for the AT91SAM7 to support USB operations. So, the choice of the crystal to use on the board was hereby fixed as well.

During the development we monitored the AT91 mailing lists and forums, and became gradually aware that operating the SSC to control an I^{2}S DAC was an often reoccurring problem for many other developers. For this reason, we added an additional DAC to the board design, a Texas Instruments

TLV5616. While not built for audio applications, and limited to 12-bit, it should be good enough to provide a usable audio signal for testing purposes. It is based around a fairly simple serial interface, which can support, among others, the SPI protocol.

4.2.4 Programming the ARM

Contacting the chip

The AT91SAM7S has an internal bootloader and can be programmed in 3 different ways: via a JTAG, via a serial debug port, and via the USB port.

The JTAG is by far the preferable option, since it also allows direct hardware debugging. They are also quite expensive. We requested a price quote on the Keil ULINK JTAG and it turned out to cost over 300€. What's worse is that some of the JTAG's are very closely linked to the manufacturers own debugging and flashing toolkit, which tends to be all but cheap as well.

A solution turns out to be the existence of the Macraigor Wiggler. Offered by Macraigor as a cheap ARM JTAG solution, it is available for 150€. However, the hardware used itself is extremely simple and has apparently been reverse-engineered. One can easily find "Macraigor Wiggler compatible" self-made JTAG circuits on the internet, as well as compatible debugging software based on GDB. We originally planned to build ourselves such a JTAG, but this would mean we would have to debug an untested circuit with an untested debugger. In the case of a single failure, we are a total guess where exactly the problem lies. We found a "Macraigor Wiggler compatible' JTAG on offer for only $19.95 at Olixmex Ltd. in Bulgaria, and ordered one. It arrived quickly and works fine. The schematics for a Wiggler compatible JTAG are given in figure 4.1.

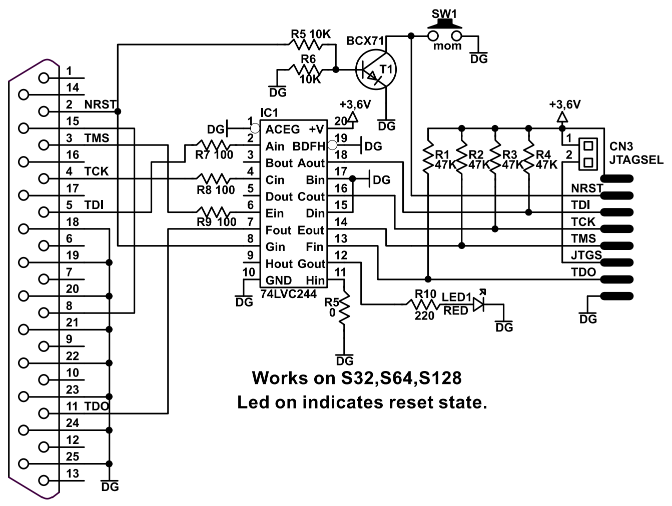

Figure 4.1: Wiggler compatible JTAG schematic

The AT91SAM7S can be programmed over the serial (RS232-like) port, or over the USB port, with the help of the SAMBA utility available from Atmel. The availability of a working JTAG made this mode of operation rather redundant, though, particularly as it is less flexible and does not allow easy debugging in the same way the JTAG does. For this reason, we removed the USB interface circuitry from our board design. We kept the serial port since it only requires 2 lines to work.

Compilers & IDEs

Many compilers and development environments are available for ARM microcontrollers. The official development environment from Keil ARM costs 1700€, but then you still have to use the free GCC compiler. If you want ARM's own compiler to go with that package, the price raises to 3300€. You'd have to have a compatible JTAG as well, so add another 300€. Clearly, this was not a fit option for us and neither is it an option for any hobby developer.

One alternative is the WinARM distribution by Martin Thomas, available at:

`http://gandalf.arubi.uni-kl.de/avr_projects/arm_projects/`

which bundles a very recent GCC crosscompiler, a small C standard library suitable for embedded chips, together with example projects for most common ARM microcontrollers and several debuggers.

We settled on another option, which was to use the Rowley Crossworks for ARM toolkit. It combines an IDE with the GCC compiler and built-in support for most ARM-based microcontrollers, a fair amount of example projects and native support for several JTAG debuggers, including the Wiggler and clones. Free trial versions are available, as well as low-cost educational licenses. We found the IDE to be very convenient to use, particularly as the JTAG support works well and the debugging support is well integrated.

4.2.5 Circuit design

The circuit design of our board is based directly on the Atmel datasheet, the publicly available reference design for Atmel's own development board, as well as the also publicly available design of Olimex's ARM development boards.

It is fairly straightforward and should contain few surprises. The power circuitry consists of a fixed-voltage low drop regulator (LDV1117V33), which feeds the internal regulator built into the AT91SAM7 as well as the VCC of the peripheral components. The internal regulator of the AT91 then provides 1.85V output for the core. A large amount of decoupling capacitors are present to provide stable power even at high clockspeeds with very variable loads.

One notable part is the external PLL circuitry. The values for these components must be calculated via an Excel sheet that is available on Atmel's website. Our values are suitable for a PLL clocked at 192Mhz.

The design was done in Eagle. All used chips and the reasoning for their presence have been mentioned before, with the exception of the ST3232CD serial driver chip, needed to convert the AT91's I/O line signals to a level suitable for RS232 serial communication. The complete schematic is presented in figure 4.2.

4.2.6 PCB design

The design of the PCB for the Bacon player proved to be one of the most difficult and time consuming aspects of this entire thesis. Aside from the author's inexperience in this area, the main problem cause was the AT91SAM7 chip itself, which comes in an LQFP package. With 64 pins, each 10 mil wide and 10 mil apart, and with many of those connected and some carrying high frequency signals, it presents a severe routing challenge. We used SMD components exclusively, which did not help either.

There was a choice whether to design the board for 4 layers and let the manufacturing be done at an external plant, or to try to design the board for 2 layers, and attempt to let the Hogeschool Gent's lab manufacture the PCB. Neither option was particularly appealing. Letting an outside firm manufacture a single 4 layer PCB is quite costly, and manufacturing the board requires a process capable of delivering 10 mil traces with 8 mil spacing, which is not even available easily. Should we let the PCB be prepared at the school lab, we faced the additional issue of having to drill & fill the numerous vias by hand.

After insistence from the responsible person in the lab that the 8/10 mil specs were in fact attainable, we eventually designed and routed it for 2 layers. Some noteworthy things we (re)learned:

- Keep the traces parallel and get sufficient distance from the chip, before forking them off one by one. Keep enough distance between the forks too, so a via can be placed if needed.

- The Eagle autorouter is usable with some pitfalls. Route the power grid first, HF traces next. Then save your work — you cannot 'undo' the autorouter. If it fails to do a reasonable enough job, look for traces it can't route and either give it more space or route them manually. You might have to route most of the board manually but it might still be handier than routing all 200+ traces by hand. Use keepout zones and Design Rule Checks to steer the autorouter where you want it.

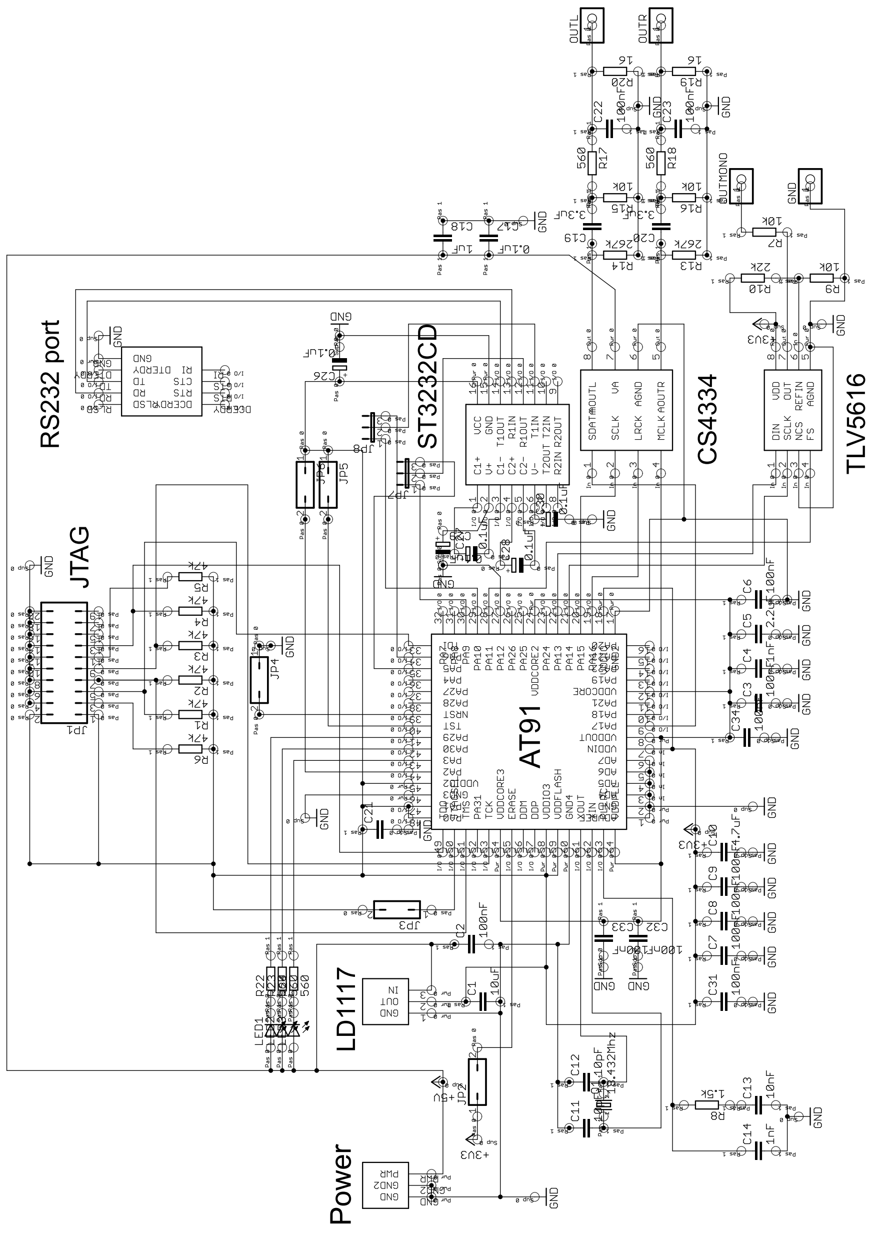

Figure 4.2: AT91SAM7S Bacon player schematic

- Be very careful with where a trace is supposed to connect to a connector. It cannot be soldered together if it is sitting below the physical package. In this case we have to go to the other side with a via first.
- Check the packages that are in the Eagle library against reality. We already learned from previous experience that components are not to be trusted, but some packages are clearly wrong as well.

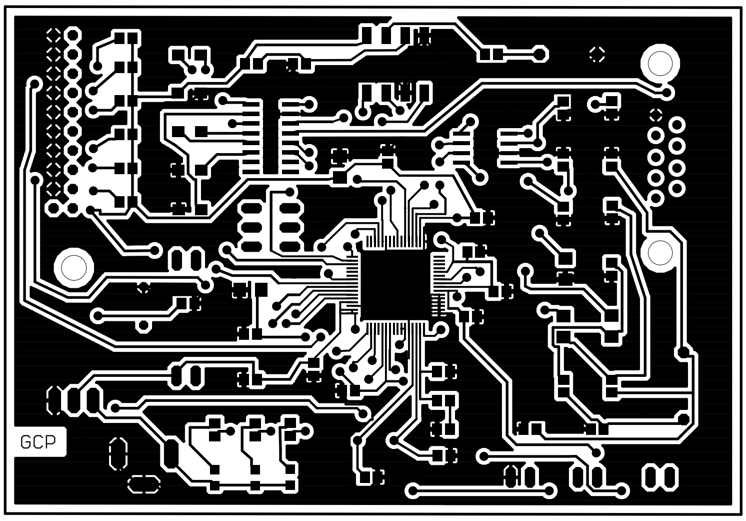

Figure 4.3: PCB Front

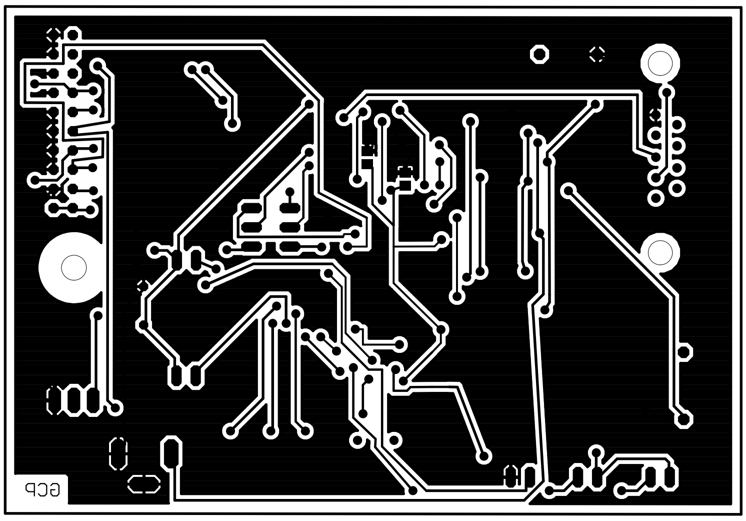

Figure 4.4: PCB Back

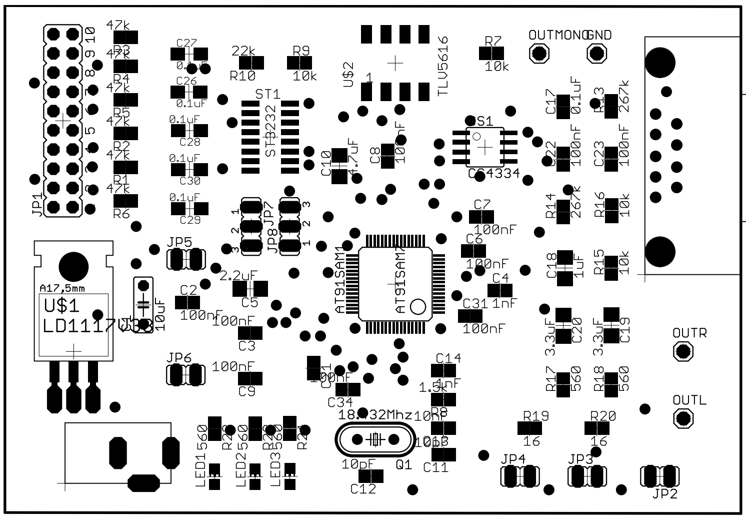

Figure 4.5: PCB Soldermask

4.2.7 Finishing & testing the PCB

Soldering

Soldering on the AT91SAM7 with it's LQFP dimensions is a daunting task. We made use of special soldering paste, soldering lint and several microscopes to verify the connections visually. It's a good idea to have a schematic next to you while doing this. We were confused multiple times by non-visible connections between pads, which we incorrectly ascribed to superfluous solder contacts.

We got some further problems with pins and pads which were not exactly matching between the PCB design and the actual components. In the case of the audio output pins, this was very problematic since the holes could not be enlarged safely without short-circuiting the signal line to a nearby ground plane. In some other cases, like the power connector or the TLV5616, a pair of cutting plier to 'adjust' the component pins a bit was sufficient.

Testing

We started the test by putting a multimeter probe on the ground plane and touching all known signal and power pins to verify there were no short-circuits. This pointed out a few badly soldered connections, which we fixed.

For the next test, we set the power supply to a restricted current output and applied power to the board. We then probed the known supply pins with a fixed voltage. This did not reveal any further issues.

We subsequently connected the JTAG, fired up the board again and tried to connect with the debugger. This worked, indicating that the chip had successfully booted and that the power circuitry and JTAG circuitry were functioning correctly. As a next step, we loaded a testing program into RAM that programmed the PLL, switched the chip to a PLL based clock (at 192Mhz / 4 = 48Mhz) and toggled the LED output pins. This did not work. Further investigation finally revealed that two of the LEDs were badly soldered and that one of them was blown. After fixing, we were successful in blinking the LEDs.

The next step was to test the serial RS232 interface, by running a program on the ARM to output debugging messages over it. This failed. Investigating showed that the ARM chip appeared to give the correct signaling on its output pins, but the voltages on the ST3232CD serial driver chip did not appear to be correct. When delivered from the supplier, this same chip was also missing all orientation signs, despite them being clearly displayed in the accompanying datasheets. So our current suspicion is that the chip is either broken or that our best guess at the correct orientation was incorrect. Because so far our attempts to extract the chip from the PCB again without breaking any of the connected traces have failed, testing and development are stalled at this point.

4.3 Conclusions & evaluation

We were successful in porting and deploying the Bacon codec on a development board based on ARM Linux. We gathered useful insights in the issues related to porting audio applications to Linux, on the operation of embedded Linux systems, on crosscompiling applications for ARM, and on the peculiarities of the ARM ports of the Linux kernel and GCC compiler.

We also succeeded in designing, building and bringing up a development board for a Flash based ARM7TDMI microcontroller. Unfortunately, deploying that failed either due to a broken component, incorrect soldering or perhaps another, not identified cause.

Nevertheless, the latter project gave us insight into available embedded ARM Flash-based microcontrollers, compilers, IDEs, debugging tools, protocols required for audio applications, and on the design, routing and soldering of PCB's based around very small package SMD ARM chips.

One positive evolution we have noticed is that when we started, the ARM Flash chips were relatively new and little documentation or good free tools where available. However, this situation has improved from week to week due to the efforts of both the chip makers and the hobbyists alike. This bodes well for any projects involving ARM microcontrollers in the future.

Bibliography

[1] WIKIPEDIA, "Acorn Computers — Wikipedia, The Free Encyclopedia", March 2006. URL: `http://en.wikipedia.org/w/index.php?title=Acorn_Computers&oldid=3744922%9`.

[2] C. ATACK AND A. VAN SOMEREN, *The ARM RISC Chip: A Programmers' Guide*. Addison-Wesley, 1993. URL: `http://www.ot1.com/arm/armchap1.html`.

[3] R. PHELAN, ed., *The ARM Thumb-2 Core Technology Architecture Extensions*, ARM Ltd, Cambridge UK, 2003. URL: `www.jp.arm.com/kk/arm_forum2003/ppt/architecture%20_extensions.pp`.

[4] S. WILSON, "ARM architecture design vs. Thumb-2 design", March 2006. Personal communication.

[5] D. SEAL, ed., *ARM Architecture Reference Manual*. Addison-Wesley, second ed., 2001.

[6] A. N. SLOSS, D. SYMES, AND C. WRIGHT, *ARM System Developer's Guide*. Elsevier Inc., 2004.

[7] J. D. JOHNSTON, "The Science of Audio in 2003", 2003. URL: `http://www.ece.rochester.edu/~gsharma/SPS_Rochester/presentations/audio%2003.pdf`.

[8] WIKIPEDIA, "Ear canal — Wikipedia, The Free Encyclopedia", March 2006. URL: `http://en.wikipedia.org/w/index.php?title=Ear_canal&oldid=41748372`.

[9] H. TRAUNMÜLLER, "Analytical expressions for the tonotopic sensory scale", *Journal of the Acoustical Society of America*, vol. 88, pp. 97–100, 1990.

[10] J. D. JOHNSTON, "Perceptual Coding of Audio Signals - A Tutorial", 2003. URL: `http://www.ece.rochester.edu/~gsharma/SPS_Rochester/presentations/Johns%tonPerceptualAudioCoding.pdf`.

[11] J. D. Johnston, "Transform Coding of Audio Signals Using Perceptual Noise Criteria", *IEEE Journal on Selected Areas in Communications*, vol. 4, pp. 314–323, February 1988.

[12] J. D. Johnston, "US Patent 5,481,614", *US Patent and Trademark Office*, 1993.

[13] J. D. Johnston, "A filter family designed for use in Quadrature mirror filterbanks.", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 291–294, April 1980.

[14] J. P. Princen and A. B. Bradley, "Analysis/synthesis filter bank design based on time domain aliasing cancellation", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 34, no. 5, pp. 1153–1161, 1986.

[15] Wikipedia, "Modified Discrete Cosine Transform — Wikipedia, The Free Encyclopedia", April 2006. URL: `http://en.wikipedia.org/w/index.php?title=Modified_discrete_cosine_tran%sform&oldid=43540251`.

[16] C. Montgomery, "Ogg Vorbis stereo-specific channel coupling discussion". URL: `http://www.xiph.org/vorbis/doc/stereo.html`.

[17] A. H. Tewfik and A. Murtaza, "Enhanced Wavelet Based Audio Coder", 1993. URL: `http://citeseer.ist.psu.edu/tewfik93enhanced.html`.

[18] D. Meares, K. Watanabe, and E. Scheirer, "Report on the MPEG-2 AAC Stereo Verification Tests", *ISO/IEC JTC1/SC29/WG11*, February 1998.

[19] D. Sinha and J. D. Johnston, "Audio compression at low bitrates using a signal adaptive switched filterbank", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 1053–1056, 1996.

[20] G. Strang, "The Discrete Consine Transform", *SIAM Review*, vol. 41, pp. 135–147, 1999.

[21] T. Yu, "QMF Filter Bank Design Using Nonlinear Optimization", May 1997. URL: `http://citeseer.ifi.unizh.ch/yu97qmf.html`.

[22] Wikipedia, "Linear prediction — Wikipedia, The Free Encyclopedia", May 2006. URL: `http://en.wikipedia.org/w/index.php?title=Linear_prediction&oldid=48144%424`.

[23] WIKIPEDIA, "Linear predictive coding — Wikipedia, The Free Encyclopedia", April 2006. URL: `http://en.wikipedia.org/w/index.php?title=Linear_predictive_coding&oldi%d=51510425`.

[24] J. HERRE AND J. D. JOHNSTON, "Enhancing the Performance of Perceptual Audio Coders by Using Temporal Noise Shaping (TNS)", *101st AES Convention*, November 1996.

[25] P. DUHAMEL, Y. MAHIEUX, AND J. PETIT, "A Fast Algorithm For The Implementation Of Filterbanks Based On "Time Domain Aliasing Cancellation"", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 2209–2212, May 1991.

[26] H. V. SORENSEN, M. T. HEIDEMAN, AND C. S. BURRUS, "On Computing the Split-Radix FFT", *IEEE Transactions on on Acoustics, Speech, and Signal Processing*, vol. 34, February 1986.

[27] D. M. EVANS, "An Improved Digit-Reversal Permutation Algorithm for the Fast Fourier and Hartley Transforms", *IEEE Transactions on on Acoustics, Speech, and Signal Processing*, vol. 35, pp. 1120–1125, August 1987.

[28] T. PAINTER AND A. SPANIAS, "Perceptual Coding Of Digital Audio", *Proceedings of the IEEE*, vol. 88, pp. 451–513, April 2000.

[29] ADVANCED RISC MACHINES LTD. (ARM), "Application Note 33: Fixed Point Arithmetic on the ARM", September 1996. URL: `http://www.arm.com/pdfs/DAI0033A_fixedpoint.pdf`.

[30] J. TRANTER AND H. SAVOLAINEN, "Open Sound System: Programmers Guide", November 2000. URL: `http://www.opensound.com/pguide/oss.pdf`.