UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Study of the audio coding algorithm of the MPEG-4 AAC standard and comparison among implementations of modules of the algorithm

by

GUSTAVO ANDRÉ HOFFMANN

Dissertation submitted to examination as
partial prerequisite to the
Master Degree on Computer Science

Prof. Dr. Sergio Bampi
Advisor

Porto Alegre, March 2002.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Hoffmann, Gustavo André
Study of the audio coding algorithm of the MPEG-4 AAC standard and comparison among implementations of the algorithm / by Gustavo André Hoffmann. – Porto Alegre: PPGC da UFRGS, 2002.
101 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2001. Orientador: Bampi, Sergio.

1. Processamento digital de sinais. 2. Áudio. 3. Codificação de sinais. I. Bampi, Sergio. II. Título.

Acknowledgements

I thank my advisor Sergio Bampi for his wise advises. Besides, I am deeply grateful to Eric Ericson Fabris for enlightening conversation and his many tips. I also thank my colleagues of the Microelectronics Group at UFRGS for all the pleasant moments.

And I am very grateful to my family for their support and patience.

Table of Contents

List of Abbreviations

AAC	Advanced Audio Coding
ADPCM	Backward-Adaptive Differential Pulse Code Modulation
CRC	Cyclic redundancy check
dB	decibel
DC	Direct current
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DL	Difference limen
DLS	Downloadable Sound
DPCM	Differential Pulse Code Modulation
DSD	Direct Stream Digital
FFT	Fast Fourier Transform
GA	General Audio
Hz	Herz
ISO	International Standardization Organization
JND	Just Noticeable Difference
L/R	Left/ right
LPC	Linear Prediction Code
LSB	Least significant bit
LTP	Long Term Prediction
MDCT	Modified Discrete Cosine Transform
MIMD	Multiple Instruction Multiple Data
MLP	Meridian Lossless Packing
MPEG	Movie-Picture Experts Group
M/S	Mid/ side
N/A	Not available
PCM	Pulse Code Modulation
PNS	Perceptual Noise Substitution
RMS	Root Mean Square
SIMD	Single Instruction Multiple Data
SNR	Signal-to-noise ratio
SWAR	SIMD within a register
SPL	Sound pressure level
TDAC	Time-domain aliasing cancellation

List of Figures

List of Tables

Abstract

Audio coding is used to compress digital audio signals, thereby reducing the amount of bits needed to transmit or to store an audio signal. This is useful when network bandwidth or storage capacity is very limited. Audio compression algorithms are based on an encoding and decoding process. In the encoding step, the uncompressed audio signal is transformed into a coded representation, thereby compressing the audio signal. Thereafter, the coded audio signal eventually needs to be restored (e.g. for playing back) through decoding of the coded audio signal. The decoder receives the bitstream and reconverts it into an uncompressed signal.

ISO-MPEG is a standard for high-quality, low bit-rate video and audio coding. The audio part of the standard is composed by algorithms for high-quality low-bit-rate audio coding, i.e. algorithms that reduce the original bit-rate, while guaranteeing high quality of the audio signal. The audio coding algorithms consists of MPEG-1 (with three different layers), MPEG-2, MPEG-2 AAC, and MPEG-4.

This work presents a study of the MPEG-4 AAC audio coding algorithm. Besides, it presents the implementation of the AAC algorithm on different platforms, and comparisons among implementations. The implementations are in C language, in Assembly of Intel Pentium, in C-language using DSP processor, and in HDL. Since each implementation has its own application niche, each one is valid as a final solution. Moreover, another purpose of this work is the comparison among these implementations, considering estimated costs, execution time, and advantages and disadvantages of each one

Keywords: MPEG-4 AAC, Audio coding, Perceptual Coders, Psychoacoustics, PC, MMX, DSP, VHDL.

1 Introduction

In many situations, network bandwidth or storage capacity may be very limited. On a network, for instance, there may be many users trying to send large amounts of data. In this context, reducing the amount of information is imperative. The main application of audio coding is in reducing the amount of bits needed to transmit or to store an audio signal. Therefore, through an algorithm, the audio coder compresses the signal prior to sending or storing it.

Audio compression algorithms are based on an encoding and decoding process. The encoding is the step in which the uncompressed pulse code modulated (PCM) signal is transformed into a coded representation. The encoder's purpose is to compress the audio signal. Thus, the compressed audio signal takes fewer bits to represent audio information because of the set of parameters created by the encoder. This set of parameters is a bitstream that may be then transmitted on a network or stored on a media.

The coded audio signal eventually needs to be restored (e.g. for playing back). To accomplish this, the coded audio signal is decoded, where it will loose its compressed form and return to the original audio format. The decoder receives the bitstream and reconverts it into a PCM representation through audio synthesis, based on the parameters held by the bitstream. Figure 1.1 shows the encoding-transmission-decoding process:

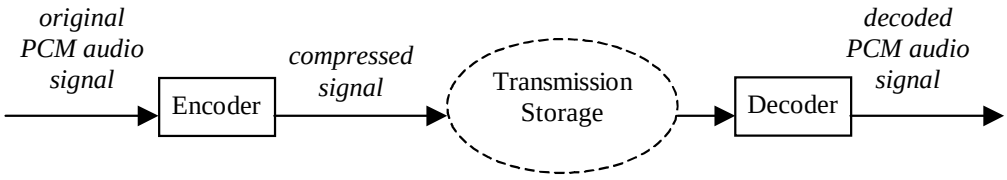

FIGURE 1.1 – Encoding and decoding of audio signal

The compression and decompression process performed by the coder may be lossless or lossy. Lossless compression provides bit-exact restoring of the original audio signal by the decoder, i.e. the original signal and the decoded signal hold the same byte-sequence. Therefore, no audio information is lost in the encoding process. Lossy compression does not guarantee that the decoded signal is a replica of the original signal. In this case, the decoded signal may be perceptually similar to the original signal.

Lossy compression algorithms generally employ Psychoacoustic models to codify audio signals [MOO 96]. Psychoacoustic models make use of characteristics and limitations of the human auditory system – like masking – to encode the signal. Thus, Psychoacoustic models provide inaudible (imperceptible) degradation of the quality of

the original signal. Through this degradation, audio information is removed from the original signal, thereby reducing the amount bits needed to represent it.

1.1 Implementation and platforms

When choosing a platform for implementation of an electronic system, there are many options in sight. One first option is the implementation in software that runs on a general-purpose processor. An interesting example is the software development for IBM-PC architecture, since it is widespread in the market. However, even when developing software for IBM-PC, there are still some choices to be done: for example, an application that only uses the standard PC (8086) instruction set could be developed, or special-purpose multimedia instruction sets could be applied, such as the Intel MMX instruction set.

In some situations, using personal computers as the target platform is unfeasible. For example, when designing a portable digital recording system, a general-purpose processor such as Intel Pentium cannot be used, since its unitary cost is too high for this application. In this case, depending on production scale, other options must be considered. For medium production scale, the use of a Digital Signal Processor (DSP) may be envisaged, since its unitary cost is lower than the cost of a general-purpose processor such as Intel Pentium. However, when large production scale is considered, a dedicated hardware application is the best choice. The so-called embedded systems are the functional integration of hardware and software for a specific application. In this case, the hardware description – which is usually done with the help of hardware description language (HDL) – may be synthesized on a Field-Programmable Gate-Array (FPGA), or detailed into a full-custom processor. Full-custom processors usually have very high costs. However, through large production scale, its cost is reduced to a price suitable for the consumer electronic market.

The purpose of this work is the implementation of blocks from the MPEG-4 AAC algorithm, considering the implementation options just mentioned. Since each implementation has its own application niche, each one is valid as a final solution. Moreover, another purpose of this work is the comparison among these implementations, considering estimated costs, execution time, and advantages and disadvantages of each one. These are the implementation accomplished in this work:

Implementation in C language. This is the first implementation and is strongly based on the C source-code provided by ISO. This source-code is a tutorial implementation of the MPEG-4 AAC algorithm and it is used for analysis and comparison with other implementations.

Implementation in Assembly of Intel Pentium. Compilers of high-level languages (e.g. C language) try to optimize the object-code through all compilation steps. However, it is not always possible for a compiler to optimize every aspect of the

code, since compilers are generic tools that have the purpose of generating guaranteed correct executable code. In many situations, some specific optimizations are overlooked by the compiler. Consequently, software development in Assembly can boost up the performance of mathematical-intensive algorithms (such as MPEG-4 AAC), making the executable code more efficient than its C-compiled correspondent. On the other hand, Assembly programming is very machine-dependent, i.e. Assembly code written for processor X (with its own instruction set) may not necessarily run on processor Y. Thus, the choice of the target processor is very important. The Intel Pentium processor was chosen, since it is widely used in the market and has a special-purpose multimedia instruction set. In conclusion, this implementation runs on an Intel Pentium processor with multimedia support.

Implementation in DSP processor. While the first two implementations were targeted on personal computer applications and general-purpose processors, the DSP implementation is targeted on dedicated devices, where using general-purpose processor is not a good financial approach. This implementation tries to take advantage of the DSP processor characteristics (e.g. optimized performance on mathematical-intensive computations) for increasing the performance of the MPEG-4 AAC algorithm.

Implementation in HDL. The last implementation is an AAC-dedicated hardware solution. In other words, it is the design of architecture uniquely dedicated to the implementation of a set of AAC module. This implementation should be well balanced, i.e. it should be optimized for computation time efficiency, but also realistic in terms of area and power.

1.2 Organization

This chapter has presented an introduction to audio coding. The second chapter presents basic concepts of Psychoacoustics. The third chapter describes the MPEG-4 AAC encoding algorithm. The fourth chapter presents the implementations of the algorithm. Finally, the last chapter presents the conclusions of this work.

2 Psychoacoustics and perceptual coding

This chapter presents a brief introduction to Psychoacoustics. This field deals with the understanding of the perception of sound. The study of human auditory system is essential for the design of perceptual coding algorithms, since perceptual coding algorithms take advantage of psychoacoustical features to compress audio signals without degrading their quality.

2.1 Decibel scale

A signal may be defined as a variable that changes with time. For example, a graph of temperature variation may define a signal. In the electrical domain, a signal x(t) may be defined by the voltage variation (in Volts). The signal power is given by

$P(t) = x(t)i(t)$, where i(t) is the instantaneous current in Amperes [HAR 98].

The average power is defined by

$\overline{P} = \frac{1}{T_D}\int_0^{T_D} P(t)dt$, where T_D is the length of time for which the averaging is taken.

The root-mean-square (RMS) value is based on the average power, and it is given by

$x_{RMS} = \sqrt{\overline{P}R} = \sqrt{\frac{1}{T_D}\int_0^{T_D} x^2(t)dt}$, where R is the resistance in Ohms (Ω) and it is usually defined as R = 1 Ω.

Both definitions of the average power and RMS value are useful to define the Decibel. Given two signals, with powers $\overline{P}_1$ and $\overline{P}_2$, the log relation between these two signals is given by

$$Decibel \stackrel{\Delta}{=} 10\log\left(\frac{\overline{P}_2}{\overline{P}_1}\right)$$

The Decibel may also be defined based on two signal amplitudes A_1 and A_2. In this case, the definition is

$$Decibel \stackrel{\Delta}{=} 20\log\left(\frac{\overline{A}_2}{\overline{A}_1}\right)$$

Note that the value in decibels (dB) is not a physical quantity as electrical voltage or acoustical pressure amplitude, but just a measure of relationship between two physical quantities.

When working in the acoustical domain, the instantaneous intensity I(t), measured in W/m^2, is based on the instantaneous pressure x(t), measured in N/m^2 or Pascals, and the fluid velocity u(t), measured in m/s. It is defined as follows:

$$I(t) = x(t)u(t)$$

The instantaneous intensity I(t) may also be defined by the acoustical impedance ρc, where ρ is the density in kg/m^3 and c is the speed of sound in m/s. The acoustical impedance ρc is defined in units of Rayls, and in the case of the air, it is 415 Rayls. In this case, the instantaneous intensity I(t) is defined by $I(t) = x^2(t)\rho c$.

The absolute acoustical Decibel scale is based on the threshold of hearing for a 1000 Hz sine tone. A reference pressure x_{REF} is provided by $x_{REF} = 20\mu Pa$. The reference intensity is therefore given by

$$I(t) = x^2(t)\rho c \cong 10^{-12}\, W/m^2$$

Hence, the sound pressure level (SPL) decibel scale is given by

$$\text{Level} \stackrel{\Delta}{=} 10\log\left(\frac{I}{I_0}\right)\text{[dB SPL]} \text{ or by}$$

$$\text{Level} \stackrel{\Delta}{=} 20\log\left(\frac{x_{RMS}}{x_0}\right)\text{[dB SPL]}$$

2.2 Threshold of hearing

In this section, we deal with loudness and the threshold of hearing. While acoustical intensity is a physical quantity, loudness is a perceptual quantity. Therefore, loudness cannot be measured with *physical* instruments, but only estimated in psychoacoustical experiments [HAR 98].

In the previous section, the absolute Decibel scale was defined based on the threshold of hearing. Therefore, a 1000 Hz sine tone at a level of 0 dB SPL is on the threshold of hearing. However, the human auditory system is not a linear device. Thus, for some intervals of frequencies, the threshold is higher than 0 dB SPL. For example, the threshold of hearing for an 80 Hz sine tone is 30 dB, and for a 40 Hz sine tone it is 50 dB.

The equal loudness contours are a set of curves that estimate the same loudness for audible frequencies. They are presented in figure 2.1:

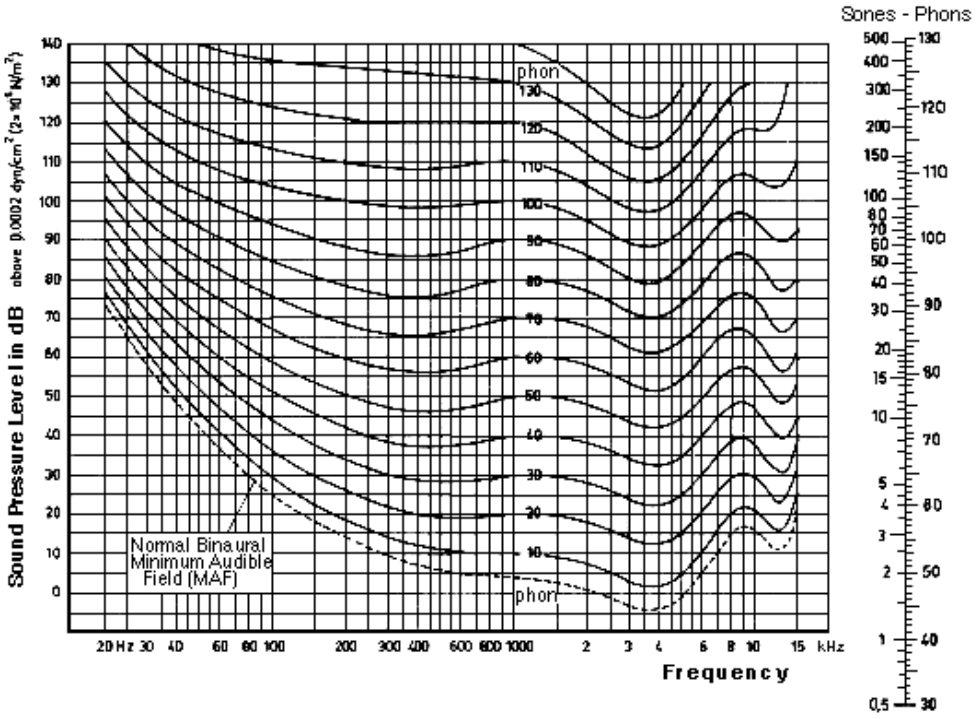

FIGURE 2.1 – Equal loudness contours (phon scale)

These contours are the base for the phon scale, which is actually a frequency-compensated decibel scale. As usual, the frequency of 1000 Hz is taken as reference. So a sine tone of 1000 Hz at a level of 50 phons (corresponding to 50 dB SPL) has the same loudness as a sine tone of 100 Hz at a level of 50 phons (corresponding to 60 dB SPL), although they do not have the same acoustical intensity.

2.2.1 Just-noticeable difference

In the study of loudness, an important measure is the smallest detectable change in loudness. This measure is known as a difference limen (DL), or just-noticeable difference (JND) [HAR 98].

The idealized measure of intensity DLs is the Weber's law. This law specifies that the DL in intensity ΔI is proportional to the intensity I, i.e. $\frac{\Delta I}{I} = k$, where *k* is a constant. In other words, the JND in level is a constant number of decibels, no matter what the starting level is.

While some psychoacoustic studies have shown that Weber's law is applicable to loudness, others have shown that it does not predict well the behavior of loudness. These results depend mainly on the kind of stimulus that is applied. For example, when white noise is used, Weber's law holds. However, for tonal signals, the intensity DL

decreases as the intensity increases. Therefore, studies of JND are not yet totally conclusive.

2.3 Perception of periodic complex tones

Periodic complex tones are based on the sum of pure (sinusoidal) tones [HAR 98]. That is, the frequencies of a periodic complex tone are integer multiples of a fundamental frequency. Mathematically, if ω_0 is the fundamental frequency, a periodic complex tone *x(t)* may be represented as a Fourier series [HAR 98] [WID 89]:

$$x(t) = \frac{A_0}{2} + \sum_{k=1}^{\infty} [A_k \cos(k\omega_0 t) + B_k \cos(k\omega_0 t)]$$

Integration is the ability of the auditory system to perceive a periodic complex tone as a single entity, instead of perceiving the individual harmonic components as separated (different) entities. In other words, the auditory system integrates the harmonic components into a single auditory entity.

Although it is usually perceived as a single entity, there are some special situations when listeners are able to perceive harmonic components individually. One example is the case when there is a spectrally prominent harmonic component (e.g. a fifth harmonic component that is 5 dB louder than the other harmonic components). This harmonic component is then heard individually as a separate entity.

Segregation is the ability of the auditory system to categorize sounds coming from uncorrelated sources. For example, the auditory system has the musical ability to separate drums and piano on a song. It also has the ability to concentrate on a given sound source and ignore other sound sources. Moreover, at a party, one is usually capable of concentrating himself on someone's voice and disregarding the other voices. This is known as the cocktail party effect. Segregation is basically a cognitive process of the auditory system, i.e. segregation is mostly performed in the brain. Gestalt principles of grouping can be successfully applied to explain auditory segregation phenomena [COO 99].

2.4 Masking

Masking occurs when one signal *x* hides another signal *y*. In this situation, signal *y* is not perceived by the auditory system (i.e. it is not heard). Thus, signal *x* is masking signal *y*. Signal *x* is called the masker and signal *y* is called the maskee or masked signal. The loudness of the masker and the masked signal are very important. The masking threshold is the loudness level when the masked signal is barely audible, and it is a used to make assertions about the masking characteristics. There are three main types of masking: tone-masking-tone (TMT), noise-masking-tone (NMT), and tone-masking-noise (TMN). Furthermore, masking has temporal characteristics. Therefore,

masking can also classified as simultaneous masking, backward masking, and forward masking [MOO 96].

Tone-masking-tone happens when a tonal signal *y* (i.e. a signal generated by a sine function) is masked by another tonal signal *x*. This may occur, for example, when signal *x* and signal *y* have near frequencies, and signal *x* is strong, while signal *y* is weak. This kind of masking is very useful for deriving masking patterns. Masking patterns (also called masked audiograms) show the masked threshold as a function of the masked signal frequency and intensity, while the frequency and intensity of the masker remains constant [MOO 96].

Noise-masking-tone takes place when a tonal signal *y* is masked by a noisy signal *x* (e.g. a signal generated by a random function). Generally, this situation comes up when *x* is weak and the noise is strong.

Tone-masking-noise shows up when a noisy signal *y* is masked by a tonal signal *x*. This is exactly the contrary situation of the previous kind of masking. This feature of the auditory system is important for audio coding, because the tonal signal (represented by a frequency line) should mask the quantization noise generated by the encoding process. Quantization noise is explained in details in the next chapter (chapter 3, section 3.4.1).

Simultaneous masking occurs when the masker and masked signals are played back simultaneously. Most psychoacoustical experiments and results are based on this kind of masking. Therefore, results from the study of simultaneous masking are the basis of the study of nonsimultaneous masking, i.e. backward and forward masking. Besides, nonsimultaneous masking is still poorly understood [MOO 96].

Backward masking (or pre-masking) happens when the masker signal can mask a signal that comes before the masker itself. This may occur, for example, when a soft signal is followed by a strong signal [COO 99]. Experiments show that backward masking depends strongly on the psychoacoustical expertise of the subjects [MOO 96].

Forward masking (or post-masking) happens when the masker signal masks a signal that comes after the masker. This may occur, for example, when a strong signal is followed by a soft signal [COO 99]. The amount of forward masking (in dB) is a linear function of *log(D)*, where D is the delay between the masker and masked signals. Besides, increments in masker level do not correspond to the same amount of increments in forward masking. For example, a masker level increase of 10 dB may represent an increase of only 3 dB in forward masking [MOO 96].

2.5 Auditory filters

2.5.1 Tuning curve and characteristic frequency

When sound waves come to us, they are converted in the cochlea into neural impulses. Roughly speaking, these impulses are used by the auditory nerve fibers (also called neurons) to determine the spectral content of the sound waves. Each nerve fiber has a characteristic frequency and a tuning curve, which are determined by psychoacoustic experiments using sine-tone stimulus [HAR 98].

The tuning curve determines what level of stimulus is needed to excite a nerve fiber by 10% over its spontaneous firing rate. Thus, the turning curve presents values of amplitudes (in dB SPL) as a function of frequency. The frequency that has the smallest amplitude value is called the characteristic frequency.

2.5.2 Excitation patterns

Nerve fibers are used by the auditory system to determine the spectrum of the incoming signal. Each nerve fiber has its own characteristic frequency, and the nerve fibers may be ordered by increasing characteristic frequency [HAR 98].

When a sine-tone stimulus is applied to the auditory system, it will excite the nerve fibers more or less, depending on the frequency of the stimulus and on the characteristic frequency of each nerve fiber. The level of excitation of each nerve fiber is considered to create an excitation pattern. Figure 2.2 presents the excitation pattern for a 750 Hz stimulus:

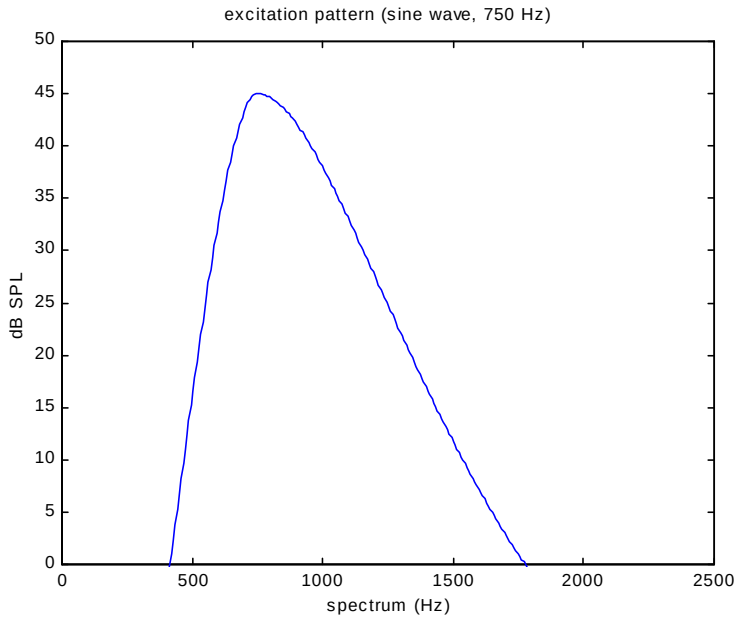

FIGURE 2.2 – Excitation pattern for a 750 Hz stimulus

As would be expected, each frequency has its own excitation pattern, which are used to determine critical bands.

2.5.3 Critical bands

Critical bands (also called auditory filters) are channels used by the auditory system to process the incoming spectral content. Each critical band represents a given portion of the spectrum, and its bandwidth is called critical bandwidth. They may be determined by noise-masking-tone experiments using notched noise [HAR 98] [MOO 96].

Critical bands may be described as rectangular filters. Although this is not a realistic model, it simplifies the mathematical treatment of critical bands. Therefore, the concept of rectangular critical bands is useful and commonly used. The bandwidth of the rectangular critical bands is called equivalent rectangular bandwidth.

The roex filter is a more elaborate filter used to describe critical bands. It gives a more realistic shape of the auditory filters, and, in mathematical terms, it is relatively simple, since it has not too many parameters. The roex*(p, r)* filter is given by [MOO 96]

$W(g) = (1-r)(1+pg)e^{-pg} + r$, where $g = \frac{|f - f_c|}{f_c}$ and f_c is the center frequency of the filter. The parameter *p* determines the bandwidth and the slope of the skirts of the auditory filter. Figure 2.3 shows the shape of the roex filter (using parameters *p*=3, *r*=0, and |*g*|=|-2..2|):

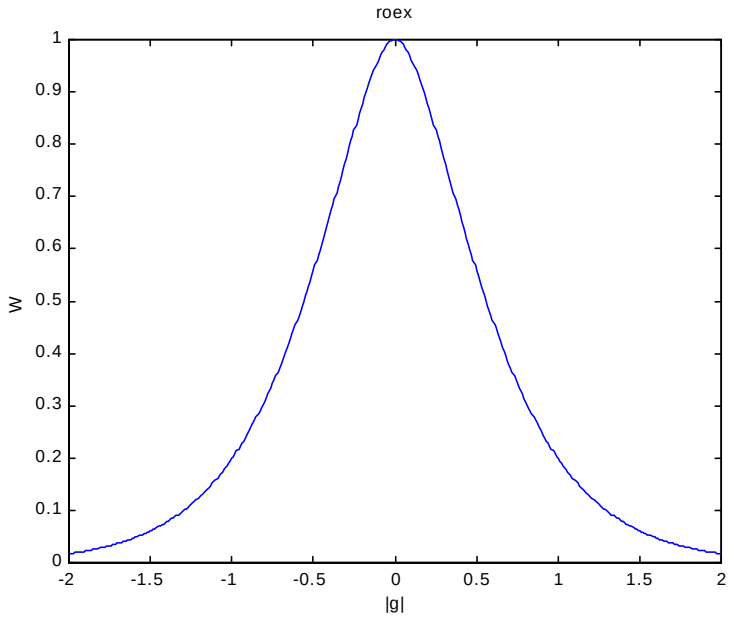

FIGURE 2.3 – Roex filter shape

2.5.4 Critical bandwidth

The critical bandwidth is the most important characteristic of a critical-band filter. Since many experiments were done and many models were conceived, there is not a unique way to determine the critical bandwidth. Currently, the most widely adopted models of critical bandwidth are the Cambridge, Munich, and the one-third-octave.

The Cambridge and Munich critical bandwidth models are both equivalent rectangular filters. They can be approximated by the following expressions, where F is the center frequency (in kHz) [HAR 98]:

$B_M = 25 + 75(1 + 1.4F^2)^{0.69}$, for the Munich critical bandwidth, and

$B_C = 24.7(1 + 4.37F)$, for the Cambridge critical bandwidth.

Both critical bandwidth models may be approximated by the one-third-octave, using the following expression:

$B_{1/3} = 232F$

Figure 2.4 presents a comparison between these critical bandwidth models:

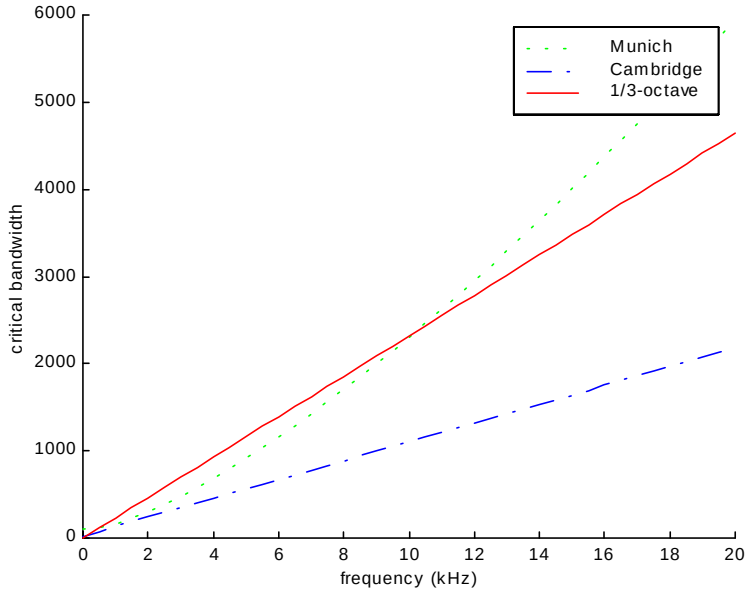

FIGURE 2.4 – Critical bandwidth models

As can be observed in the figure, the one-third-octave may be defined as a linearization of the Munich critical bandwidth. Besides, the Cambridge model always gives bandwidth values lower than the Munich model.

2.5.5 Critical-band number & Bark scale

A critical-band number is an alternative measure of frequency based on critical bands. The most widely adopted critical-band number is the Bark scale, and it is based upon the Munich critical bandwidth model. The integer Bark values correspond to lower band edges. Table 2.1 presents some values for the Bark scale [HAR 98]:

TABLE 2.1 – Bark scale

Bark	**Lower (Hz)**	**Center (Hz)**	**Upper (Hz)**
0	0	50	100
1	100	150	200
2	200	250	300
3	300	350	400
...			
23	12000	13500	15500
24	15500		

For example, the third band is from 200 Hz to 300 Hz (critical bandwidth of 100 Hz). And a value of 23.5 Bark corresponds to 13500 Hz.

3 MPEG-4 AAC

ISO-MPEG is a standard for high-quality, low bit-rate video and audio coding. It was created by the efforts of MPEG (Moving Pictures Experts Group), a group established by ISO/IEC. The standard consists of a collection of algorithms, chosen among several available coding algorithms for coding of video and audio. The audio part was produced in collaboration with AT&T, CCETT, FhG/University of Erlangen, Philips, IRT, and Thomson Consumer Electronics [BRA 96a]. The main task of the audio subgroup was to standardize an algorithm for low-bit-rate audio coding, i.e. an algorithm that should provide signals with bit-rates much lower than the CD bit-rate (1378.125 kbit/s). Besides, audio quality should not be lost in the codification, so the encoder must ensure the highest possible audio quality.

It is important to mention that the normative part of the standard only describes the decoder, whereas the encoder is left to an informative part. The standard does not explain how to build a high-quality audio encoder. It just presents the formulae and algorithms that should be used by the encoder. Therefore, new encoders may be developed, provided they are compliant with the standardized bitstream.

The Motion-Picture Expert Group (MPEG) has created many video and audio coding algorithms. The audio coding algorithms consists of MPEG-1 (with three different layers, corresponding to different coding algorithms) [BRA 96b] [BRA 99] [MPG 93], MPEG-2 [STO 96] [MPG 94], MPEG-2 AAC [BRA 98] [MPG 97], and MPEG-4 [MPG 99] [MPG 2000]. In this chapter, just the MPEG-4 AAC encoding process is described in details.

3.1 Overview

ISO-MPEG-4 Audio is composed by many audio coding tools, such as HILN [EDL 98] and Structured Audio [SCH 99a] [SCH 99b] [SCH 99c]. Among them, there is the MPEG-4 Advanced Audio Coding (AAC) tool [GRI 99], which is an improvement of the previous MPEG-1/2 Layer III and MPEG-2 AAC algorithms. Figure 3.1 shows the block diagram of the algorithm [BRA 96a] [BRA 96b], as defined by the standard [MPG 99]:

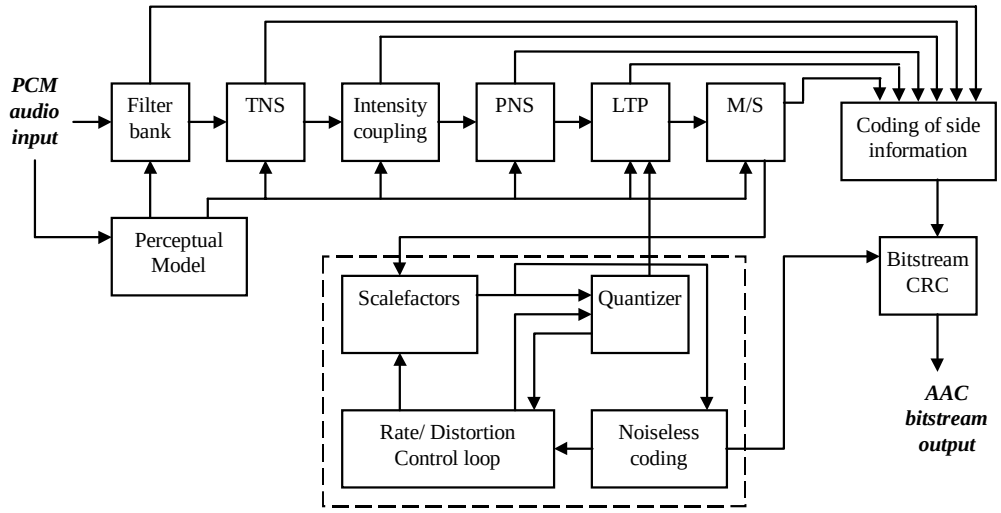

FIGURE 3.1 – MPEG-4 AAC block diagram

The AAC encoder receives PCM audio samples and transfers them to the filterbank and the perceptual model. The filterbank converts the samples from the time-domain to the spectrum domain through a modified cosine transform. The perceptual model, by its turn, provides information about bit-allocation (number of bits used in the encoding process) and maximum allowed noise. This information is used in the encoding process to ensure high quality of the codified signal at the lowest bit-rates.

After the filterbank, the spectral data is passed through some tools to enhance the coding quality or efficiency. The temporal noise shaping (TNS) tool is used to control the quantization noise in the time-domain by applying a filtering process in the spectral domain. The Intensity Coupling process is used in stereo channels to reduce the number of bits needed for encoding by coding two channels as a single channel. The perceptual noise substitution (PNS) is used to code noisy spectral data as white noise instead of spectral coefficients simply by indicating the level of the noise energy. While the PNS tool is used for noisy signals, the long-term prediction (LTP) tool is used to enhance the coding of tonal signals by applying a prediction process to time-domain samples and transmitting just the error (indicated by the difference between predicted and actual signal). The M/S decision is used for stereo signals and determines if the stereo channels should be encoded as the usual left and right channels, or as middle and side channels.

The spectral data is quantized by AAC quantization process, which is composed by four subblocks: quantization of scalefactors, quantization of spectral coefficients, noiseless coding, and rate/distortion control loop. These blocks use the information provided by the perceptual model to ensure high-quality and coding efficiency of the encoded spectral data.

In the last step, the quantized spectral data and the side information from the AAC tools are encoded in the bitstream. An additional CRC process is applied to the

bitstream to reduce its vulnerability to transmission errors. Finally, the bitstream is ready for transmission or storage.

Table 3.1 summarizes some characteristics of the AAC encoder. This information is explained in the following sections.

TABLE 3.1 – MPEG-4 AAC characteristics

Characteristic	Specification
Bit-rates	32-160 kbit/s (kbps) for mono signals 64-320 kbit/s (kbps) for stereo signals
Sampling-rates	8, 11.025, 12, 16, 22.05, 24, 32, 44.1, 48, 64, 88.2, 96 kHz
Stereo encoding modes	Mono, Stereo, Joint-stereo using intensity stereo, and Joint-stereo using M/S (middle-side). Support for multichannels (up to 48 channels).
Block length (input PCM samples)	2048 and 256
Filterbank	MDCT
Filterbank coefficients	2048 and 256
Overlap	50%
FFT (points)	1024
Encoding	Uses Huffman coding and bit-reservoir

3.2 Filterbanks

A filterbank is a basic block of any audio encoder. It is used as a time-frequency mapping from an audio signal (time-domain signal) to spectral components called frequency lines [BRA 96b]. When coding signals, redundancies and irrelevancies may be removed from the original signals, thereby reducing the amount of bits needed to code the signals. Filterbanks allow for signal decorrelation, and therefore reduce the redundancies of the signal. Moreover, when filterbanks are combined with psychoacoustical models, they reduce the irrelevancies of the original signal [BOS 99].

Redundancies appear, for example, in a pure sinusoidal signal. In this kind of signal, the values repeat themselves after a fixed period. Therefore, this signal is redundant. Moreover, it requires many samples to be represented in the time-domain. However, only three parameters (frequency, phase, and amplitude) are needed to represent it in the frequency-domain. Thus, by representing the signal in the frequency-domain in place of the time-domain, redundancies are removed from the original signal [BOS 99].

Common audio signals do not have just one spectral component. Instead, they have many components that appear at low and high frequencies. When all spectral components have the same energy, it is called a flat spectrum. In this case, no coding gains are achieved, since all spectral components demand the same number of bits.

However, in many cases, audio signals usually do not have a flat spectrum; the low frequency components have more energy than the high frequency ones (this kind of spectrum is called a *colored spectrum*). In this case, by redistributing the *bit pool* (i.e. the number of bits used for coding each spectral component) and assigning fewer bits to high frequencies, coding gains can be achieved. Therefore, the spectral *flatness* measure determines the coding gains of the bit pool redistribution [BOS 99].

Irrelevancies may be removed when psychoacoustical models are used. In this case, some portions of the spectrum that are irrelevant to the human auditory system are isolated and coded using fewer bits. Irrelevancy of each spectral component is calculated through masking thresholds, as explained in section 3.5.

MPEG-4 AAC uses a time-to-frequency transform as its filterbanks. It is a modified version of the discrete cosine transform (DCT), accordingly called modified discrete cosine transform (MDCT). The modification of the DCT consists of an improvement for time-domain aliasing cancellation (TDAC) [BRA 96a] [BRA 96b]. Besides, the MDCT provides critical sampling (i.e. the number of frequency components is equal to the number of time samples), and it is used in association with windows of different lengths. The transition between different windows is called adaptive window switching [NOL 97]. These windowing methods are described in the next section.

The DCT is strongly associated with the Discrete Fourier Transform (DFT). In fact, the DCT can be derived from the DFT. Like the DFT, the DCT is an orthogonal transform. However, while the DFT is a transform from real or complex sequences to complex ones, the DCT is a transform from real sequences to real ones. Thus, the spectral coefficients resulting from a forward DCT are real coefficients. While the DFT is based on sine and cosine basis sequences, the DCT is based solely on cosine basis sequences [OPP 98].

The forward DFT and backward DFT transforms are defined, respectively, by [OPP 98]:

$$A[k] = \sum_{n=0}^{N-1} x[n]\phi_k^*[n], \text{ and } x[n] = \frac{1}{N}\sum_{k=0}^{N-1} A[k]\phi_k[n]$$

There are many different definitions of the DCT. Each definition constitutes a different type, based on specific assumptions on symmetry and periodicity. The DCT-2 (DCT type two), for example, is defined by [OPP 98]:

$$X[k] = 2\sum_{n=0}^{N-1} x[n]\cos\left(\frac{\pi k(2n+1)}{2N}\right), \text{ and}$$

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1}\beta[k]X[k]\cos\left(\frac{\pi k(2n+1)}{2N}\right), \text{ where } \beta[k] = \begin{cases}\frac{1}{2}, & k = 0\\ 1, & 1 \le k \le N-1\end{cases}$$

The main advantage of the DCT is its energy compaction property. Specifically, the DCT-2 is used in many data compression applications (e.g. video and speech compression), since the coefficients resulting from the forward transform are more highly concentrated at low indices than the coefficients from the DFT [OPP 98].

The input block for the AAC filterbank has a length of 2048 or 256 time-domain samples, depending on the type of window used by the algorithm. Windowing is dealt with more specifically in the next section. The MDCT formula is defined by [LIU 99]:

$$X_k = \sum_{i=0}^{N-1} x_i \cos[\frac{\pi}{2N}(2i+1+\frac{N}{2})(2k+1)], \text{ for k = 0, 1, ..., N/2 –1, where:}$$

x_i are the input time-domain samples

X_k are the output spectral coefficients

i is the index of the time-domain samples

k is the index of the spectral coefficients

N is the length of the input block of samples

The MDCT can be calculated from a DCT-4 (DCT type four) by using two equations of input permutation, as described in [LIU 99].

An example of the use of filterbanks is shown below. Figure 3.2 presents an input audio signal:

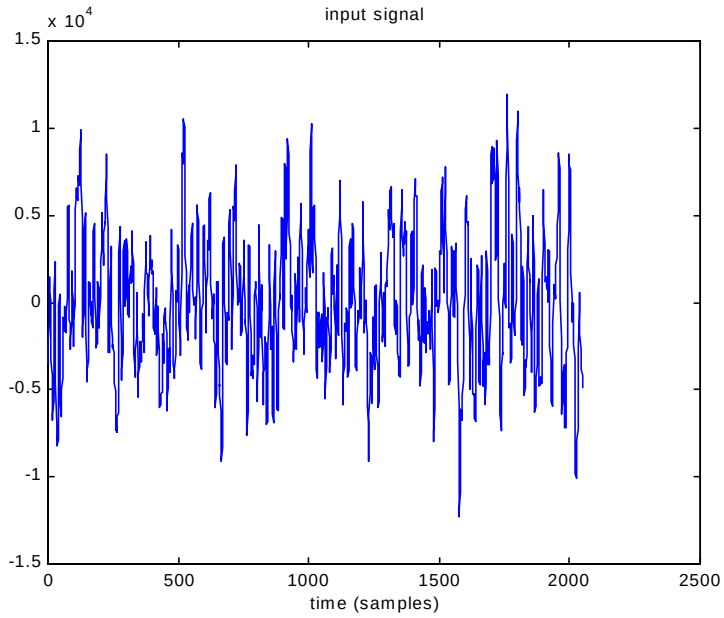

FIGURE 3.2 – Input audio signal (example)

Figure 3.3 shows the MDCT and the FFT of this input audio signal:

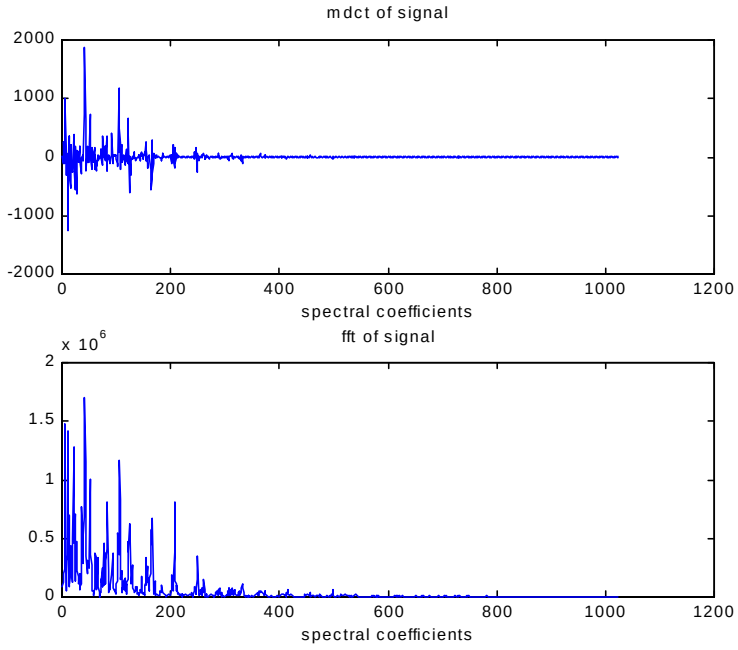

FIGURE 3.3 – MDCT and FFT of the audio signal

These figures are referenced in section 3.4, where AAC's quantization is discussed.

3.3 Adaptive window switching

When an MPEG-4 AAC encoder receives the input time-domain samples, it groups them into blocks of samples. These blocks are also known as windows. There are two main types of windows, which are classified according to their lengths: the long window (which holds 2048 samples), and the short window (which holds 256 samples). There is an overlap of 50 % between consecutive windows. For long windows, for example, two adjacent windows will have 1024 samples in common.

Because of its limited time-resolution, a window may not represent well the rapid-changing (transient) signals. When a long window is used to encode signals with many time-domain events, such as triangles or castanets, preechoes are heard in the decoded signal, i.e., noise is spread out over some time before the music event causing the noise. At a sampling frequency of 44.1 kHz, this spreading may occur over a signal block of 46.44 ms (time resolution of 23.22 ms). By using a short window with one eighth of the length of a long window, a time resolution of 2.9025 ms is achieved [BRA 96b] [BRA 99].

Adaptive window switching provides dynamic changing of the window length and shape. The adaptation is based on the assumed condition for preechoes, which may be evaluated by the amount of bits needed to encode the signal: when there is a demand for bits far exceeding the average, a preecho condition is presupposed [BRA 96b].

As the encoder receives the input samples, a sequence of windows is created. In this sequence, when changing from one type of window to another one (e.g. when changing from a long window to a short one), a different type of window must be used to smooth this transition. Therefore, when changing from long to short windows, a start window is used, and when changing from short to long windows, a stop window is used. Besides, for the sake of simplicity, the number of consecutive short windows if fixed. Only eight windows may be used in a sequence, since eight short windows have the same length as one long window [MPG 99].

When using windows in sequence, the terminology is modified. All the “sequenced windows” have the same length of 2048 samples, so this may be the justification for the change of name. For example, a sequence of eight short windows becomes an eight-short sequence.

Table 3.2 summarizes these windows:

TABLE 3.2 – Adaptive windows

Window	Sequence	Description
Long window	Only long sequence	This is the normal window used for blocks holding stationary signal.
Start window	Long start sequence	This hybrid window is used to switch between a long and short window.
Short window	Eight-short sequence	It has the same shape as the long window, but compressed to one-eighth of the window length.
Stop window	Long-stop sequence	This window is used to switch from a short window back to a long window.

Figure 3.4 illustrates the four types of windows. It shows the long, short, start, and stop windows [MPG 99].

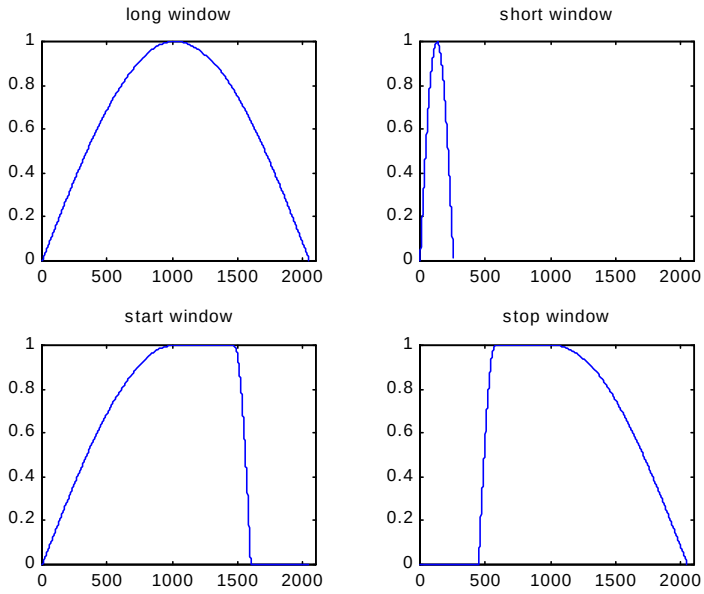

FIGURE 3.4 – Shape of adaptive windows

Figure 3.5 exhibits a typical sequence of window types if adaptive window switching is used. It contains two long windows (only-long sequence), followed by one start window (long-start sequence), an eight-short sequence, one stop window (long-stop sequence), and one long window.

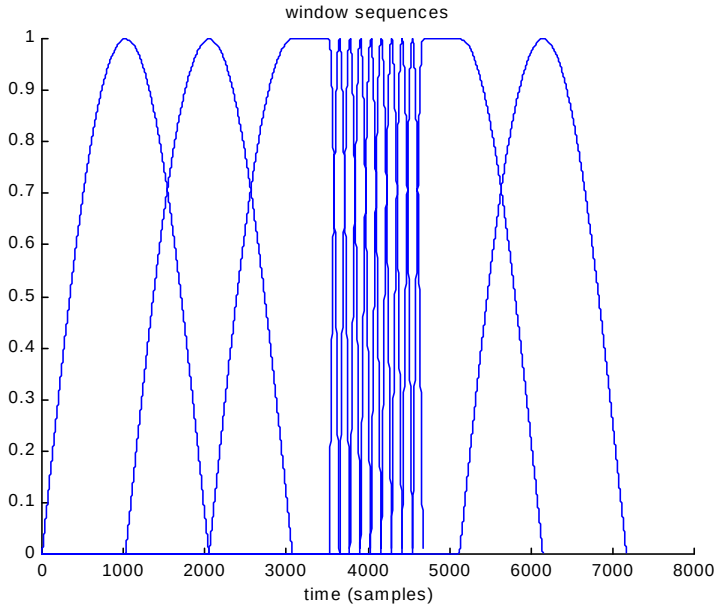

FIGURE 3.5 – Adaptive window switching sequence

3.4 Quantization and coding

Audio digitalization is the process that transforms an analog audio signal (e.g. an electrical signal coming from a microphone) into a digital representation (also called digital word). It consists of sampling and quantization. The following subsection provides an overview of quantization.

3.4.1 Quantization and quantization noise

Quantization [ORF 96] [OPP 98] [BLE 78] [GRA 98] is assigning a digital word to an incoming continuous analog voltage. Mathematically, if *x(t)* is the analog input signal, and $x_Q(t)$ is the digital output signal, the quantization is represented by:

$$x_Q = x(t) + e(t)$$

Since digital information can only take a finite number of possible values, the quantized value will always correspond to the original value plus an error *e(t)*. The signal *e(t)* is called quantization noise, and is considered uncorrelated to the original signal *x(t)*.

The number of bits of quantization plays an important role in the degree of quantization noise. Increasing the number of bits decreases the quantization noise. Therefore, the number of bits (B) is used as a measure of quality of an audio system. For linear conversion, the signal-to-noise ration (SNR) of an audio system is determined by:

$$SNR = 20\log_{10} 2^B = 6.0206B \approx 6B \text{ [dB]}$$

Therefore, CD-quality systems (using 16 bits) have an SNR of 96 dB. And newer 24-bit audio systems have an SNR of 144 dB.

Linear pulse-code modulation (LPCM) [BLE 78] [ORF 96] [OPP 98] is the most classical and widespread digitalization format. As the name implies, the quantization is linear, i.e. there is a fixed quantization interval between one quantization level (corresponding to a given digital word) and the next quantization level (corresponding to the next digital word). Although LPCM is the most common format, other digital representation systems exist, such as floating-point converters, differential PCM (DPCM) [BLE 78], backward-adaptive differential PCM (ADPCM) [SAB 96] [WYL 96], Meridian Lossless Packing (MLP) [GER 99], and Direct Stream Digital (DSD) [VER 98] [BRU 98], among others.

MPEG-4 AAC quantization is actually a requantization process, since any digital signal is already a quantized signal. The original digital signal representation is usually in the pulse code modulation (PCM) format with fixed number of bits. One example is the compact disc (CD) that uses the 16-bit PCM format. AAC's requantization changes the original digital signal representation into a new representation, using fewer bits without affecting the perceived audio quality.

3.4.2 Scalefactors

While the CD is based on the PCM format, which stores time-domain samples, AAC uses time-frequency samples, given by spectral information based on short-term windows of time-domain samples. As mentioned in section 3.2, the output of AAC’s filterbanks is spectral MDCT coefficients. In the quantization process, these coefficients are converted into a kind of floating-point representation, given by a mantissa and an exponent. In AAC terms, the exponent is called scalefactor, since it is a factor that determines the resolution of the quantization scale.

The MDCT coefficients are grouped into different consecutive bands. Each band is assigned a scalefactor. Therefore, these bands are called scalefactor bands. The scalefactor bands try to imitate the auditory critical bands (as discussed in chapter 2). The number of coefficients for a given scalefactor band depends on sampling rate and window length. As would be expected, there are more scalefactor bands for long windows (e.g. 49 for 44.1 kHz sampling rate) as for short windows (e.g. 14 for 44.1 kHz sampling rate), since long windows have more MDCT coefficients than short windows [MPG 99].

To obtain the value of all scalefactors, a common scalefactor (*csf*) is calculated based on the greatest MDCT coefficient value (*max_x*), using the following formula [MPG 99]:

$csf = start\_cfs + \text{int}(\frac{16}{3}\log_2(\text{abs}(\frac{max\_x^{\frac{3}{4}}}{MAX\_QUANT}))$, where $start\_cfs = 40$ and $MAX\_QUANT = 8191$. If *max_x* is zero, then $csf = 20$.

All scalefactors use the common scalefactor as a starting point, thus reducing the amount of bits necessary to encode scalefactors. Actually, the effective scalefactor (*real_sf*) for band *sb* will be $real\_csf = \mathbf{sf}(sb) - cfs$, where ***sf*** is an array of scalefactors.

The value of all scalefactors is defined within the quantization loops. Thereafter, the scalefactors are encoded. The first scalefactor is usually coded as the *global gain*, and the other scalefactors are differentially coded according to the previous scalefactor. For example, if scalefactor **sf(**2**)** is 24 and **sf(**3**)** is 23, **sf(**3**)** will be encoded as -1 (*minus one*).

3.4.3 Quantization of MDCT coefficients

The requantization of the MDCT coefficients reduces the amount of bits necessary to encode these coefficients. The requantization is based on the scalefactors. For each MDCT coefficient *x*, its quantized value x_{quant} is given by [MPG 99]:

$$x_{quant} = \text{sign}(x) \cdot \text{int}\{[\text{abs}(x)^{\frac{3}{4}} \cdot 2^{[\frac{3}{16}(\mathbf{sf}(sb) - csf)]}] + MAGIC\_NB\}$$

where *MAGIC_NB* is 0.4054.

The inverse quantized value x_{inv_quant} is also used in the requantization. This value is compared to original coefficient *x* in order to calculate the quantization noise added to the signal. Quantization noise results from the fact that *x* and x_{inv_quant} may be different. The inverse quantized value $x_{inv\_quant}(i)$ is given by [MPG 99]:

$$x_{inv\_quant} = x_{quant}^{\frac{4}{3}} \cdot 2^{[-\frac{1}{4}(\mathbf{sf}(sb) - csf)]}$$

The distortion caused by the requantization (on a scalefactor band basis) is calculated by [MPG 99]:

$distortion(sb) = \sum_{i=lower(sb)}^{upper(sb)} [\text{abs}(x(i)) - x_{inv\_quant}(i)]^2$, where *lower(sb)* and *upper(sb)* indicate the lower and upper limits of the scalefactor band (i.e. they are indices to MDCT coefficients).

3.4.4 Noiseless coding

Noiseless coding [QUA 97] is performed over the quantized information (spectral coefficients and scalefactors) to further reduce the amount of bits needed to encoding. As the name implies, no distortion is added to the signal through this kind of

coding. The process is based on spectral clipping, Huffman coding, and sectioning (for long windows) or grouping and interleaving (for short windows).

Spectral clipping is a dynamic range limiting method. It is applied to the quantized spectrum when it results in savings of bits. The *clipped* coefficients are then coded separately, and sent as side information in the bitstream. This kind of coding is also called pulse escape method [MPG 99].

Up to four clipped coefficients are allowed. An index *i* specifies the scalefactor band of the lowest-frequency clipped coefficient. The clipped coefficients are coded as magnitude and an offset from the base index *i* [MPG 99]. Figure 3.6 illustrates the spectral clipping method applied to 32 quantized spectral coefficients:

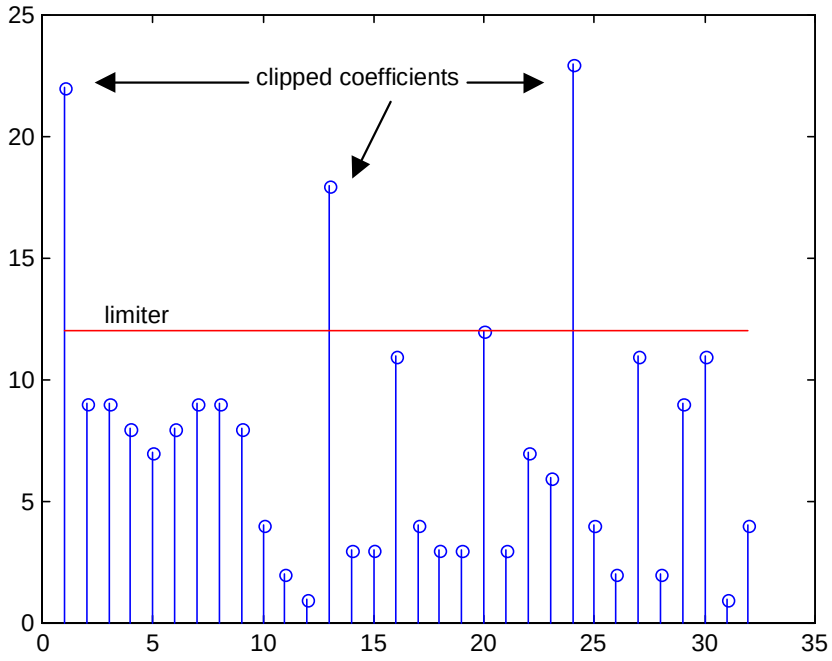

FIGURE 3.6 – Spectrum clipping example

Huffman coding is then applied to the quantized information. Scalefactors and quantized spectral coefficients are coded separately. There are eleven codebooks for spectral coefficients and one codebook for scalefactors. The codebooks are distinguished by dimension (two or four), sign (unsigned or signed), and largest absolute value (ranging from 0 to 12, or greater values using the escape codebook) [MPG 99].

For each scalefactor band, one of these eleven codebooks is chosen to code the spectral coefficients (i.e. each scalefactor band may be coded using a different codebook). Within each scalefactor band, the coefficients are grouped into 2- or 4-tuples, depending on the codebook. For example, let the quantized spectral coefficients be [0 -1 0 0 1 1 ...]. In this case, codebook 1 may be used, for which the spectral

coefficients are signed, their largest absolute value is 1, and they are grouped in 4-tuples sequences. After choosing the codebook, the codeword (i.e. the bit sequence used to encode the coefficients) must be determined. For each codebook, a specific algorithm is used to determine the codeword. For codebook 1, the algorithm goes as follows.

The first 4-tuple is [0 -1 0 0], and the index to the codebook is given by the dot-product with [27 9 3 1] plus 40. In this specific case, the result is 0*27 + (-1)*9 + 0*3 + 0*1 + 40 = 31. In codebook 1, the codeword for index 31 is 10111 (i.e. the hexadecimal number 17), which only takes 5 bits for representation. Therefore, coefficients [0 -1 0 0] are encoded as 10111.

Because consecutive scalefactor bands may employ the same codebook for coding its coefficients, they can be grouped into sections [MPG 99]. Therefore, each section corresponds to a group of consecutive scalefactor bands that employ the same codebook. As a first step, each scalefactor band corresponds to a section. A sectioning method merges scalefactor bands that employ the same codebook. This method also tries to merge scalefactor bands that do not use the same codebook, choosing the codebook with higher index. Naturally, a section merge is only carried out when this results in savings of bits. Please note that sectioning is only used for long windows. Figure 3.7 illustrates sectioning:

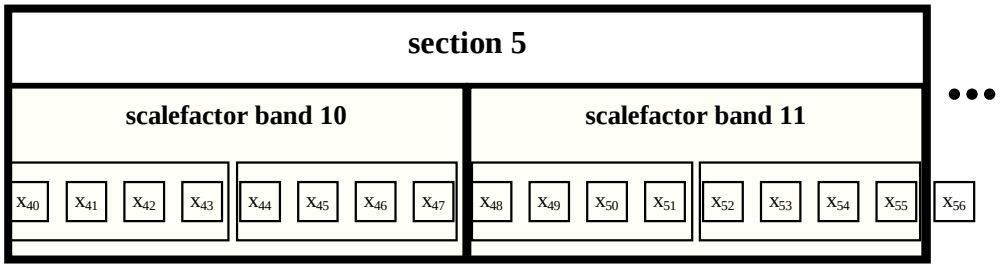

FIGURE 3.7 – Sectioning of scalefactor bands

For short windows, a grouping and interleaving method is used instead [MPG 99]. Since there is always a sequence of eight consecutive short windows being applied to the audio signal, these windows may be grouped. When two or more windows are grouped, the scalefactor bands share the same scalefactor (e.g. if windows 2 and 3 are grouped, then scalefactor band 1 would have the same scalefactor for these windows). The interleaving method interchanges the order of scalefactor bands and windows. Figure 3.8 shows how the bitstream would be arranged if windows 2 and 3 are grouped:

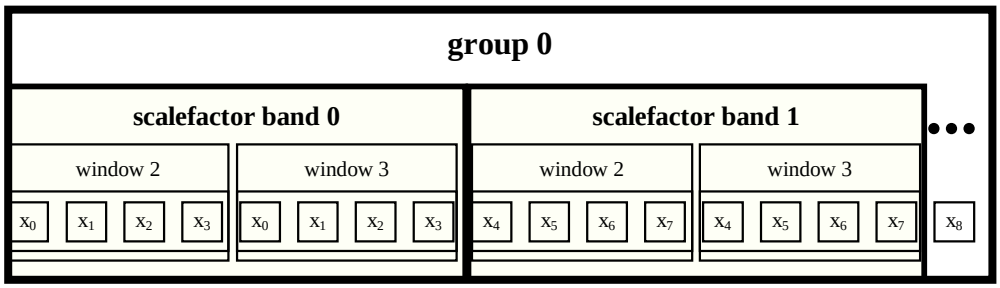

FIGURE 3.8 – Grouping and interleaving

3.4.5 Rate and distortion control loop

AAC requantization is composed by two loops: an outer and an inner loop [MPG 99]. The inner loop is also called rate loop and it controls the bit-rate of the quantization. The outer loop is also called distortion loop and it controls the distortion produced by the requantization.

The inner loop (rate control loop) makes the quantization of coefficients, according to the formulae indicated in section 3.4.3, and calls the noiseless coding process. Thereafter, the loop counts the amount of bits used to encode the coefficient and scalefactors. If the amount of bits is higher than a maximal value (derived from the pre-established transmission bit-rate), the quantization is redone, until the value is not higher than the maximal value.

The outer loop (distortion control loop) begins by executing the inner loop. Then, it calculates the distortion produced by the requantization. If the distortion is higher than the allowed distortion (indicated by the x_{min} provided by the psychoacoustic model, as explained in section 3.5), the loop adjusts the scalefactors and calls the inner loop again.

In order to avoid infinite loops, exit points must be defined in the outer loop. Imagine the situation when the inner loop reduces the amount of bits for quantization, because it exceeds the maximal value, and the outer loop increases the amount of bits, because the distortion is too high. Of course, this can easily lead to an infinite loop. Hence, scalefactors cannot be adjusted forever. Therefore, limiting the allowed adjustment of scalefactors creates an exit point. As expected, this exit point allows some distortion to be added to the encoded signal.

The result of the quantization depends largely on the input signal and the implementation of the control loops. In section 3.2, an input audio signal was presented. An example of quantization of the MDCT coefficients and scalefactors is presented in figure 3.9. Please note that the sign of the scalefactors was inverted in this figure.

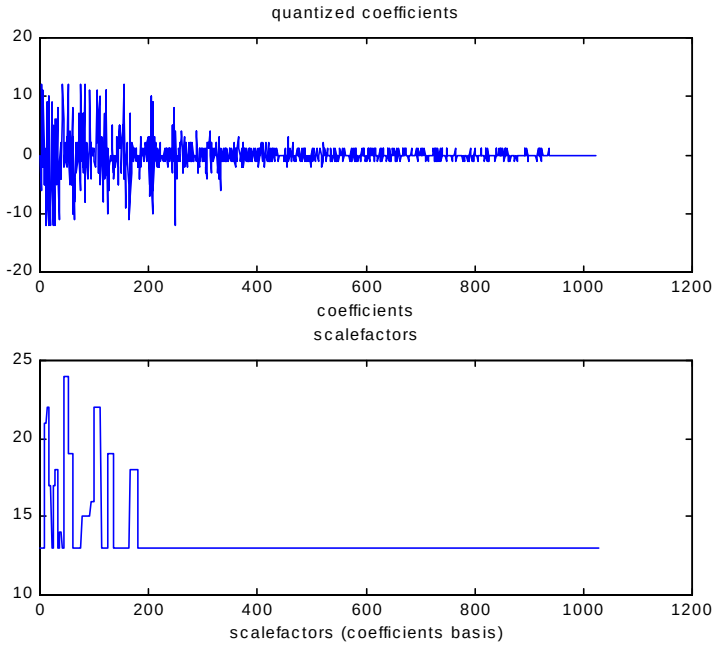

FIGURE 3.9 – Quantization of MDCT coefficients and scalefactors

Comparing this figure with the original unquantized MDCT coefficients (figure 3.3), we observe that the scalefactors try to mimic the behavior of the coefficients, assigning a higher scalefactor to scalefactor bands holding higher coefficient values, as clarified by figure 3.10:

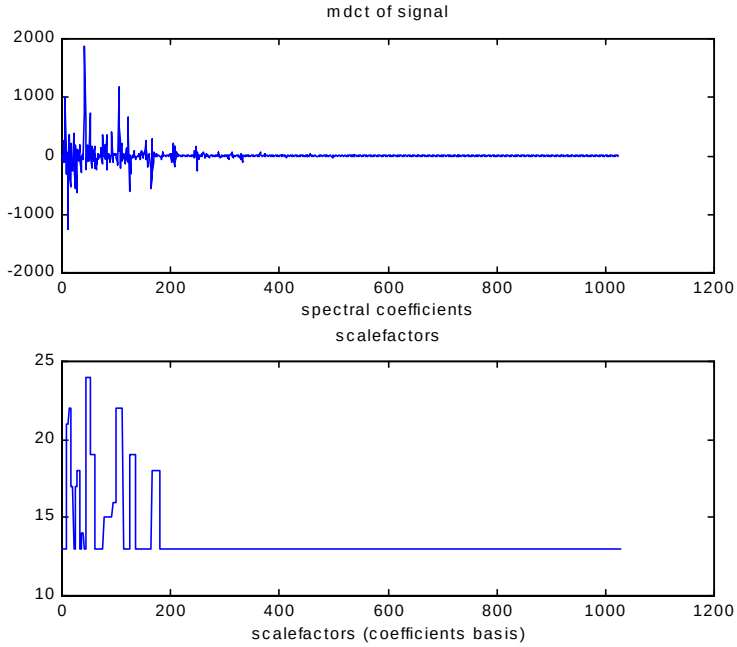

FIGURE 3.10 – Original MDCT coefficients and quantized scalefactors

As shown in the figure, the scalefactors graph shows peaks in the same spectral range as the original MDCT coefficients.

3.5 Psychoacoustical model

This sections presents the AAC psychoacoustical model. Many features of Psychoacoustics (explained in chapter 2) are employed here. Firstly, noise shaping is explained, followed by an explanation of masking of quantization noise. Then, the psychoacoustical model is presented.

3.5.1 Noise shaping

Noise shaping was primarily employed in the design of analog-digital (A/D) signal converters. As explained in section 3.4.1, an error *e(t)* appears when quantization is applied to the input signal, as given by $x_Q = x(t) + e(t)$. The error *e(t)* is the quantization noise, and it is assumed to be white noise. Therefore, the spectrum of the quantization noise is flat. However, in many applications, it is desirable that the spectrum of quantization noise be reshaped. For instance, delta-sigma converters, which are based on oversampling, employ noise shaping to shift the spectrum of the quantization noise to higher frequencies [ORF 96] [JES 2000], thereby reducing the amount of quantization noise in audible frequencies.

In perceptual coders, noise shaping is used to lessen the amount of quantization noise generated by the requantization of the spectral coefficients. For example, AAC non-linear quantization formula (as shown in section 3.4.3) already provides some sort of noise shaping. However, to make sure that quantization noise will not be heard in the decoded signal, a masking mechanism must be used, as explained in the next section.

3.5.2 Masking of quantization noise

Masking of quantization noise is accomplished through the use of psychoacoustical concepts explained in the previous chapter, such as masking (section 2.4) and auditory filters (section 2.5).

For each critical band, a masking threshold is estimated. Within each critical band, the spectral component with highest level is the masker, whose level is used to estimate the masking threshold. All spectral components within the critical band that have levels below the masking threshold are masked. This masking occurs for quantization noise as well as for low-level tonal components. Therefore, if the level of quantization noise is correctly set to be below the masking theshold, the quantization noise remains inaudible [NOL 97].

The signal-to-mask ratio (SMR) is the distance between the minimum masking threshold and the level of the masker [NOL 97]. Note that, at this point, all estimations are based on the sound pressure level of the frequency components.

When quantization is applied to the spectral coefficients, a given signal-to-noise ratio (SNR) depending on the number of bits used results, as explained in section 3.4.

Actually, the SNR is not directly related to sound pressure level, but it is used to estimate and control the quantization noise.

At this point, two ratios are available: the SMR and the SNR. To ensure that quantization noise remains inaudible, the SNR must be greater than the SMR. Therefore, the number of bits used to encode a given spectral component may be reduced by requantization, as long as the SNR resulting from the requantization remains greater than the SMR [NOL 97].

3.5.3 AAC perceptual model

The main purpose of the perceptual (or Psychoacoustical) model is to calculate the signal-to-mask ratio (SMR), the codec masking threshold (x_{min}), the bit allocation information (e.g. the minimal number of bits used for encoding in addition to the average bits), and to indicate the window type (long, start, stop, or short type) to be applied to the current block [MPG 99]. When coding stereo channels, the perceptual model is not only applied to the left and right channels, but also to the middle and side channels (see section 3.7.1 for more details about M/S-coding).

The perceptual model receives the input time-domain samples of the audio signal. Hann-windowing is applied to the input audio signal x. This is given by

$x_H(t) = x(t) \cdot (0.5 - 0.5\cos(\frac{\pi(t+0.5)}{N})$, where N is the number of samples (window length). Figure 3.11 presents this window:

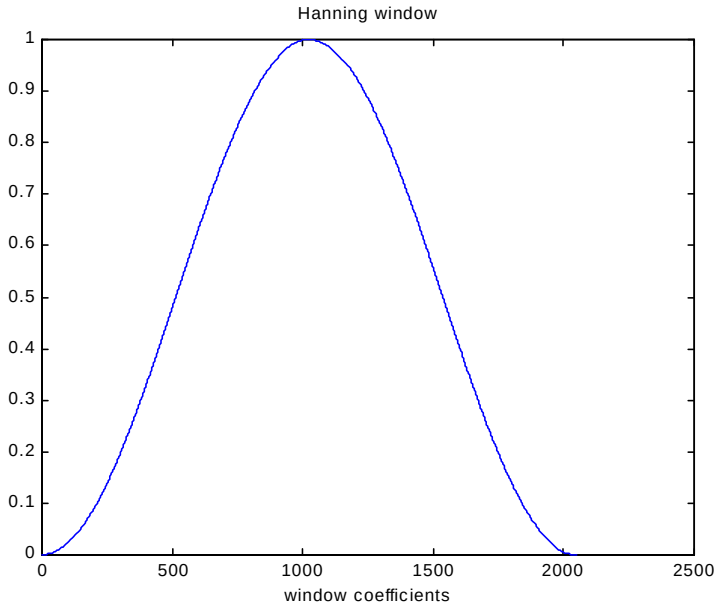

FIGURE 3.11 – Hanning window

The Hann-windowed signal x_H is then used to compute the spectrum using the Fourier Transform. Computationally, this is accomplished through the Fast Fourier

Transform (FFT). At first, the spectrum is in the rectangular form, which uses complex numbers. The spectrum is then converted to the polar form, which is composed by the magnitude and phase spectra.

The spectrum is divided into spectral partitions. These partitions are used to calculate the *threshold*, which is the maximum distortion energy masked by the signal energy. Hence, the partitions are also called threshold partitions. This partition is almost equal to the one-third-octave division of the spectrum [MPG 99].

The partitions are based on the same concept as the scalefactor bands applied to the MDCT coefficients. However, one threshold partition does not correspond to one scalefactor band. For example, using 44.1 kHz sampling rate and considering long window, the first partition takes the first two spectral coefficients, while the first scalefactor band takes the first four MDCT coefficients. Besides, for this very configuration (44.1 kHz and long window), there are 70 spectral partitions and 49 scalefactor bands.

The spectra of the two previous input blocks are used to calculate the predicted magnitude and phase. Considering that consecutive spectra represent an evolution of the frequency components in time and assuming that musical signals are quasi-periodic, there is always a degree of predictability that can be observed in succeeding spectra. That is, we can predict the spectral coefficients of subsequent spectra by observing the previous spectra. However, predictions may be not totally correct, i.e. the predicted and the actual spectra may diverge to some degree. There are some spectral components that cannot be predicted; i.e. they are unpredictable. Therefore, the unpredictability measure is calculated, which measures how much of the spectrum cannot be predicted. Once calculated for each spectral component, the unpredictability measure is calculated for each spectral partition [MPG 99].

Based on the spectrum, the energy of each spectral partition is calculated. This is given by $energy(p) = \sum_{i=lower(p)}^{upper(p)} spectral\_magnitude(i)^2$, where p is the partition and *lower(p)* and *upper(p)* are the limits of each partition in terms of spectral index. The energy is used to calculate the actual energy threshold in a subsequent step.

After the energy and the unpredictability measure for each partition are calculated, they are convolved with the cochlea spreading function. The convolution of the energy and the spreading function generates the excitation pattern. The spreading function estimates the effect of masking across critical bands and it is based on the Bark scale. Therefore, a table that indicates the Bark value of each frequency is provided. Figure 3.12 presents the spreading function:

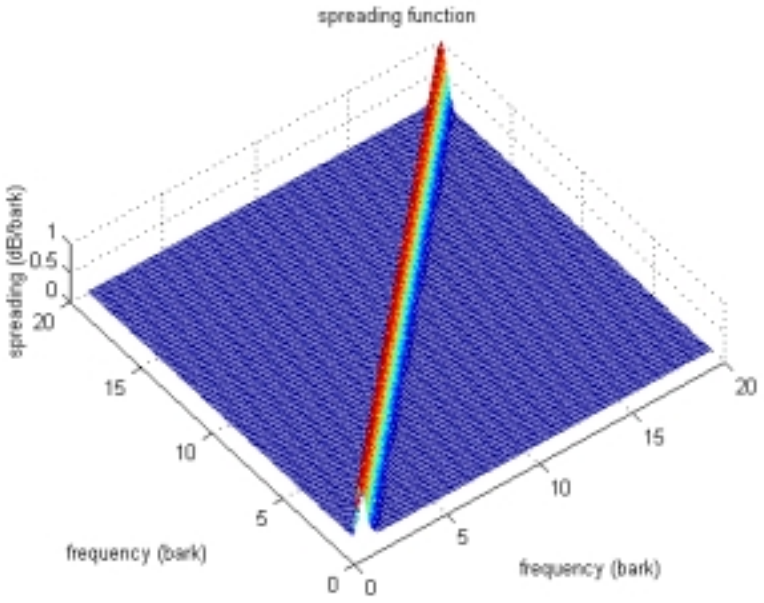

FIGURE 3.12 – Spreading function

Figure 3.13 presents a two-dimensional graph of the spreading function for a signal at the center frequency of 5 barks (510 Hz).

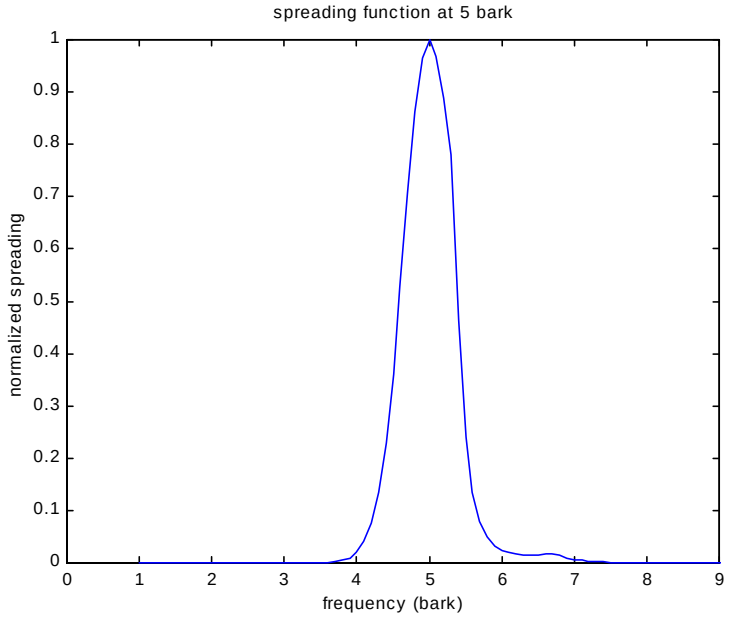

FIGURE 3.13 – Spreading function for signal at 5 bark

The convolved unpredictability measure is used to calculate the tonality index of each partition. Considering that any signal may have tonal (i.e. periodic) and noisy (i.e. random) components, the tonality index measures the level of tonal components of the spectral partitions [MPG 99].

The tonality index is used to calculate the signal-to-noise ratio (SNR) in each partition. The SNR is then adapted to the *power ratio*, given by $pow\_ratio(p) = 10^{[\frac{-SNR(p)}{10}]}$.

Through the *power ratio* and the convolved energy, the actual energy threshold is calculated. Then, the masking threshold is recalculated, taking the threshold in quiet and the threshold for the previous block into consideration. The recalculated threshold and the energy are used to calculate the perceptual entropy. The perceptual entropy is mainly used in preecho control [MPG 99].

Preecho control is actually accomplished by the use of window decision. As explained in section 3.3, preecho is avoided by switching from long window to the short window type. Therefore, depending on the value of the perceptual entropy, one kind of window or another is chosen.

Thereafter, the calculation of the SMR and the codec threshold x_{min} is performed [MPG 99]. Both calculations are adapted from the original spectral partitions to the actual scalefactor bands. This is necessary because the AAC quantization uses this information to quantize the MDCT coefficients, which are based on scalefactor bands. The x_{min} gives the maximum allowed distortion (i.e. quantization noise) for each scalefactor band. This value is used by the distortion control loop to adjust the scalefactors, as explained in section 3.4.5.

The last step is the calculation of the bit allocation information. This information indicates to quantizer how many bits are available for encoding. This information is used by the rate control loop, as explained in section 3.4.5.

3.6 Temporal noise shaping

When low time-resolution is available (e.g. at low bit-rates and lower sampling frequencies), the time-structure of some signals may be too *detailed* to be correctly represented by the encoder. In this case, the quantization noise is spread over the entire temporal block, and it may become audible. The situation is critical when there are transient signals, which are present, for example, at speech signals. Therefore, temporal noise shaping (TNS) is employed to enhance the quality of the audio signal, especially speech signals at low bit-rates [GRI 99] [BRA 99].

As explained in section 3.5.1, noise shaping is used to reshape the spectrum of quantization noise. In the case of TNS, noise shaping is used to reshape the time-domain shape of quantization noise within the transform block.

Temporal noise shaping is based on the fact that the spectrum of a sinusoidal signal corresponds to a single peak. In other words, the time-domain tonal signals have spectra that contain transients. For example, the signal $x(t) = \cos(2\pi \cdot 440t)$ has a

spectrum that holds a single peak at the frequency of 440 Hz. This is actually the basis for the Fourier Transform theory.

Clearly, when viewing the spectrum as another signal, the peak at this frequency corresponds to a transient signal. Therefore, it becomes apparent that transient signals at the time domain correspond to tonal signals at the frequency domain [LIN 2001]. Either at time or frequency domain, tonal signals are easily predicted through a technique called linear predictive coding (LPC), as explained in the next subsection. Therefore, filtering of the spectrum with LPC coefficients attenuates the increase of quantization noise generated by transient signals.

3.6.1 Linear predictive coding

Linear predictive coding (LPC) [SPA 2000] [PRE 92] is used for the analysis and the synthesis of signals. It is a process where the current sample is predicted by a linear combination of previous samples. The p-order LPC is represented by $x_{pred}(k) = \sum_{i=1}^{p} a_i x(k-1)$, where x is the input signal, x_{pred} is the predicted signal, and a_i are the prediction parameters. The prediction parameters are obtained by minimizing the mean square of the prediction error (ε), as indicated by $\frac{\partial \varepsilon}{\partial a_i} = 0,\ i = 1,2,..., p$, where $\varepsilon = E[(x(k) - x_{pred}(k))^2]$. In the last formula, *E[.]* denotes the statistical expectation operator.

The most famous algorithm for efficient computation of LPC is the Levinson-Durbin algorithm [SPA 2000]. This algorithm calculates reflection coefficients, which can then be transformed into LPC parameters.

3.6.2 AAC Temporal noise shaping

The first step of AAC temporal noise shaping (TNS) [MPG 99] is the choice of the region of the spectrum where TNS will be applied. For example, the range from 1.5 kHz up to the higher scalefactor band may be a choice. Note that this frequency range is indicated in terms of scalefactor bands.

LPC is calculated for the MDCT coefficients (for the pre-determined frequency range). The LPC calculation is based on a fixed maximum noise shaping filter order, which is 5 for long/start/stop windows, and 3 for short window. The LPC calculation (through the Levinson-Durbin algorithm) provides the expected prediction gain g_p and the reflection coefficients r. Depending on the value of g_p, TNS is used or not. This is determined by a threshold value (established in the encoder) that should be exceeded to activate the use of TNS.

The LPC reflection coefficients r are quantized. Then, the quantized reflection coefficients r_q are converted into index values *index* using inverse sine transformation

[MPG 99] [SPA 2000]. These *index* values are sent as side information in the bitstream. From *index* the inverse quantized reflection coefficients r_{inv_q} can be derived.

The actual noise shaping filter order is determined by searching for the last reflection coefficient with an absolute value greater than a pre-determined threshold. All reflection coefficients with absolute value smaller than the threshold are ignored, and the remaining reflection coefficients indicate the filter order, as long as the filter order does not exceed the maximum noise shaping filter order.

The remaining reflection coefficients are used to get the LPC coefficients $coeff_{lpc}$. These coefficients are used as FIR filter coefficients, and a FIR filtering process is applied to the frequency range chosen in the first step of the TNS encoding, i.e. the MDCT coefficients are processed through a FIR filter using $coeff_{lpc}$.

3.7 Stereo coding

Stereo signals are usually coded as two independent channels, i.e. the left (L) and the right (R) channels are coded separately, as though they were completely uncorrelated. Two alternative methods exist for coding of stereo channels: middle-side (M/S) coding, and intensity coupling. These methods are mutually exclusive, i.e. M/S-coding cannot be used in companion with intensity coupling, and vice-versa. The following subsections describe these stereo coding methods.

3.7.1 Middle-side coding

Middle-side coding [MPG 99] groups the stereo channels and explores the differences between two channels in a simple way: the middle channel is the sum of the left and the right channels ($M = (L + R)/2$) and the side channel is the difference between the channels ($S = (L - R)/2$). Middle-side coding is an efficient coding method for stereo channels when there is a great degree of correlation between two channels. For example, the same signal may be being transmitted through the left and right channels. Or the right signal may be a phase-delayed version of the left signal. In these cases, middle-side coding may be more efficient than the usual left-right coding.

As already mentioned in section 3.5.3, the perceptual model is applied to the M/S-channels, in addition to the usual L/R-channels. An imaging-control process is used to adapt the results from the perceptual model to the fact that there may be correlations between the L/R- or between the M/S-channels. This adapted information from the perceptual method is then used within the rate and control loops for quantization of scalefactors and coefficients. The number of bits required for coding the M/S- and the L/R-channels is calculated and, based on this, the method that uses fewer bits is chosen.

3.7.2 Intensity coupling

While M/S coding does not necessarily incur in lost of information, intensity coupling uses psychoacoustical features to reduce the number of bits needed for encoding. Intensity stereo exploit irrelevancies at high frequencies. Coding using intensity coupling is done as follows [MPG 99].

A region of the spectrum (in terms of scalefactor bands) is chosen. For example, the region of frequencies above 6 kHz may be chosen. For each scalefactor band, the energy (E) of the left (L), right (R), and sum (S=L+R) channels is calculated. Using the information of the energy, the intensity position is calculated by

$$is\_position(sfb) = \text{round}\left[2\log(\frac{E_l(sfb)}{E_r(sfb)}) \right]$$

The intensity positions are coded as side information in the bitstream.

Next, the spectral coefficients for intensity stereo are calculated, based on the spectral coefficients of the left and right channels and the energy information as follows:

$$spec_{is}(i) = [spec_l(i) + spec_r(i)] \sqrt{\frac{E_l(sfb)}{E_s(sfb)}}$$

For the scalefactor bands where intensity stereo is being applied, the spectral coefficients of the left channel are replaced with the coefficients for intensity stereo ($spec_{is}$), and the all coefficients of the right channel are set to zero. Then, the standard quantization process may be applied to the coefficients and scalefactor bands.

3.8 Perceptual noise substitution

Perceptual noise substitution (PNS) [MPG 99] is a method for efficient coding of noisy signals. The algorithm scans a given spectral range (e.g. frequency above 4 kHz) and performs a noise detection process. A spectral band is classified as noise-like if it is neither tonal (what is indicated by the tonality index) nor has strong energy changes over time. The tonality index is calculated in the psychoacoustical model, as explained in section 3.5.3.

The noise detection process is performed in a scalefactor band basis. When a noisy scalefactor band is detected, the whole scalefactor band marked as noisy (through a *noisy-flag*). All coefficients of the band are set to zero before quantization, so no bit is necessary for encoding these coefficients. The energy of the noisy scalefactor band is calculated and coded into the scalefactor, which provides a resolution of 1.5 dB. This scalefactor is then differentially encoded like all other scalefactors. Therefore, the only

difference between the encoding of *normal* scalefactor bands and PNS-coded scalefactor bands is a *noisy-flag* (which is assigned in the noiseless coding, using one pseudo-codebook called NOISE_HCB).

3.9 Long-term prediction

Long-term prediction (LTP) is used to reduce the redundancy between successive signal blocks. LTP is especially efficient for tonal signals [MPG 99] and it is only used in the encoding process when it leads to saving of bits.

The first step of LTP encoding is the estimation of the optimal LTP. This is accomplished by minimizing the error through variation of parameters α and b_k in the following formula:

$$P(z) = \sum_{k=-m_1}^{m_2} b_k z^{-(\alpha+k)} .$$

Parameter α corresponds to the delay (lag) for prediction. Parameter b_k represents the prediction coefficient. Both parameters are quantized and sent as side information.

The predicted signal x_{pred} for the (m+1)th frame is calculated from the inverse quantized time-domain signal from previous frames, as indicated by

$$x_{pred}(i) = \sum_{k=-m_1}^{m_2} b_k x_{inv\_quant}(i - 2N + 1 - k - \alpha), \text{ where}$$

$i = mN + 1, mN + 2, ..., (m+1)N$.

The predicted signal and the actual signal are used to calculate the error signal. The error signal is quantized and transmitted as side information.

4 Implementations of critical AAC modules

This chapter presents the details of each implementation of AAC modules. As already mentioned in chapter 1 (section 1.1), the implementations are in C language, in Assembly, in DSP, and in VHDL.

4.1 C implementation

The C implementation consists of source-code provided by ISO-MPEG (version 990224). The source-code is completely written in C, and it is made as portable as possible. It was used as a basis for average computation time analysis and implementation and optimizations in other architectures. No modifications or optimizations were done in the source-code; just adaptations to make it compile under the PC platform.

4.1.1 Average computation time analysis for all modules and functions

Based on the ISO implementation of the AAC encoder, without modifications, the average computation time for each frame was determined. The original C source-code was compiled on an IBM-PC using Borland C++ 5.0 compiler. Two processors were used for average computation time analysis: Intel Pentium II 350 MHz processor, and Intel Pentium III 750 MHz processor. Twenty CD-quality (16-bit 44-kHz stereo) PCM input files were used, with file size (duration) varying from 3.291 s to 10.029 s. The sum of all file sizes gives 113.0150 s of sound used in the analysis of the average computation time. The selection of the files took the variety of music styles into account (e.g. classical, jazz, rock, and pop music). Appendix 1 presents details of the input files. For the first analysis of the average computation time, the coding bit-rate was 320 kbps.

Table 4.1 presents the results for the average computation time per frame using Pentium II (PII) and Pentium III (PIII) processor. The results are based on input sound files holding stereo channels, and the performance for mono channel is estimated by simply dividing the result for stereo channels by two.

TABLE 4.1 – Average computation time analysis (C version, 320 kbps)

Analysis	***PII 350 MHz***	***PIII 750 MHz***	***Speed-up***
Average coding time (stereo channels)	0.836625 s	0.365897 s	2.2865
Average coding time (mono channel)	0.418313 s	0.182949 s	2.2865

As indicated in the table, running the same executable file on Pentium III 750 MHz instead of Pentium II 350 MHz means a speed-up of 2.2865 times in the average computation time.

Figure 4.1 shows the average coding time for each AAC module or internal function (for Pentium II 350 MHz), considering the codification of one frame and stereo channels. Appendix 1 explains the purpose of each module or internal function.

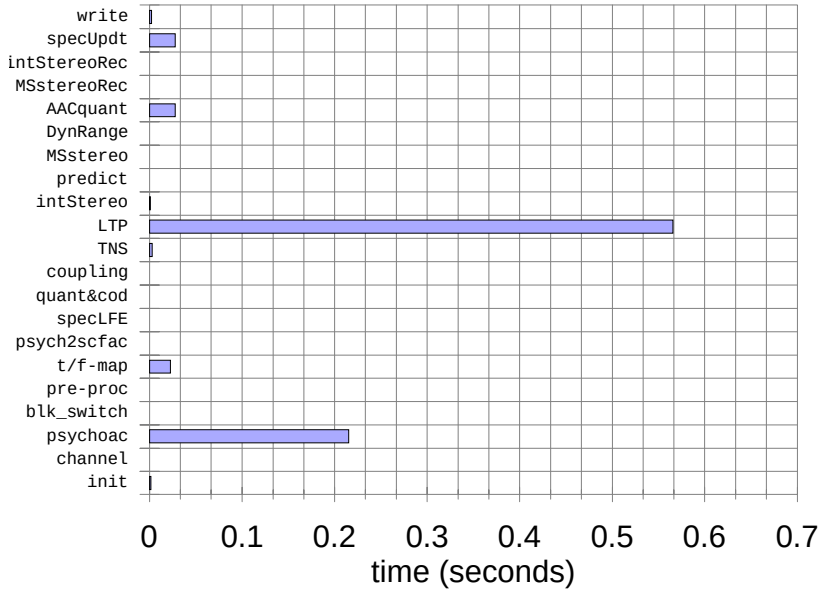

FIGURE 4.1 – Computation time per frame for each module (stereo, 320 kbps, PII)

Figure 4.2 shows the average computation time distribution for each AAC module or function. We notice that the LTP and Psychoacoustic modules are the most time-consuming modules, taking respectively 65.27% and 24.86% of the average computation time. Both correspond to 90.13% of the average computation time. Therefore, considering this analysis, optimizations of the execution of these modules are necessary and provide sensible reduction in the overall average computation time.

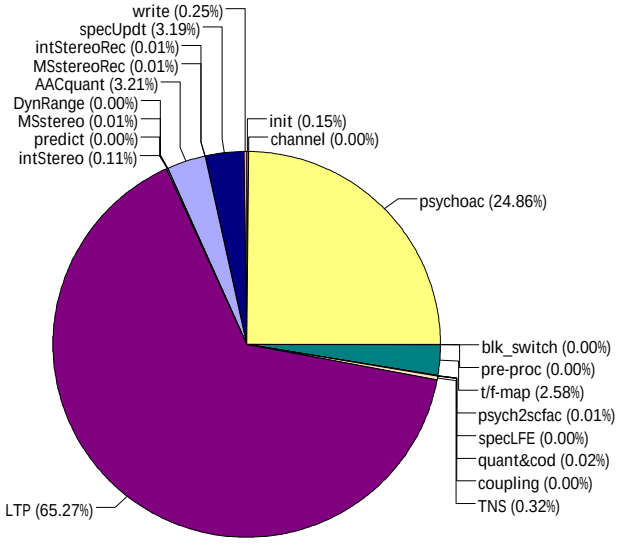

FIGURE 4.2 – Average computation time distribution (320 kbps, PII)

Of course, the average computation time per frame (stereo) for each module using Intel Pentium III 750 MHz processor is lower, but the average computation time distribution remains practically the same. For this processor, the most time-consuming modules are the LTP (62.48%) and the Psychoacoustic module (26.33%), therefore not too far from the results for the Pentium II 350 MHz processor.

Changing the coding bit-rate also changes the results. Using the coding bit-rate of 128 kbps in a new analysis, the average computation time of most modules remains practically the same (compared to the results for 320 kbps). However, the average computation time of the AAC quantization module increases, contributing to an increase of 17.86% for the average computation time of the encoder. In addition, some modules presented a very small reduction in average computation time (less than -0.5 % contribution). As would be expected, these results modify the distribution of computation previously presented in figure 4.2. Figure 4.3 presents the average computation time for each module:

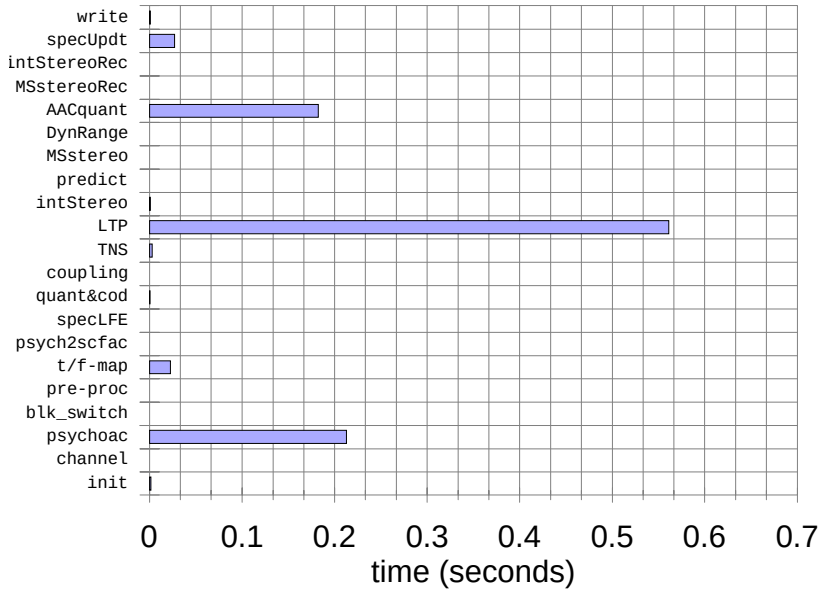

FIGURE 4.3 – Computation time per frame for each module (stereo, 128 kbps, PII)

Using the coding bit-rate of 32 kbps, the average computation time of the AAC quantization module increases again. In comparison with the average computation time of encoding at 320 kbps, the average computation time of the encoder presents an increase of 73.87%. The AAC quantization module contributes to an increase of 72.18% of the average computation time of the encoder, while the LTP module contributes to an increase of 2.09% of the average computation time, and the average computation time of the other modules remains practically the same. The following figure presents the average computation time of each module, considering the 32 kbps coding bit-rate:

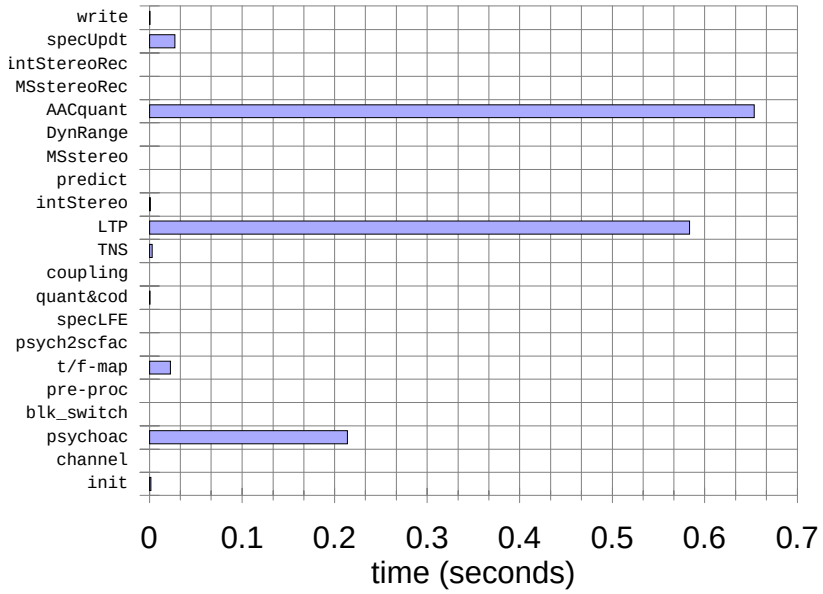

FIGURE 4.4 – Computation time per frame for each module (stereo, 32 kbps, PII)

This demonstrates that the average computation time of the AAC quantization module depends deeply on the choice of coding bit-rate, while the other modules present a fairly constant average computation time. This occurs because the AAC quantization module (as coded on the original C code from ISO) is based on the rate/distortion loops without optimizations. When coding for low bit-rates, these loops take too many cycles to minimize the distortion and at the same time achieve the specified bit-rate. This job is easier for higher bit-rates (on this C code). In the following sections, only the coding bit-rate of 320 kbps is considered, since it was the chosen coding bit-rate for analysis and comparisons.

4.1.2 Average computation time of the LTP module

The profiling of the LTP module provides the following average computation time distribution for its internal routines, as shown by figure 4.5:

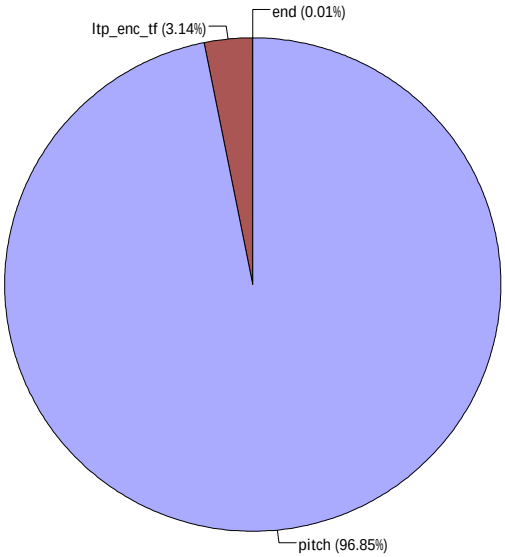

FIGURE 4.5 – Computation time distribution of LTP's routines (Pentium II)

We can clearly see in this figure that the *pitch* function takes most of the average computation time, i.e. 96.85%. Therefore, a detailed analysis of this function was done, as shown by figure 4.6:

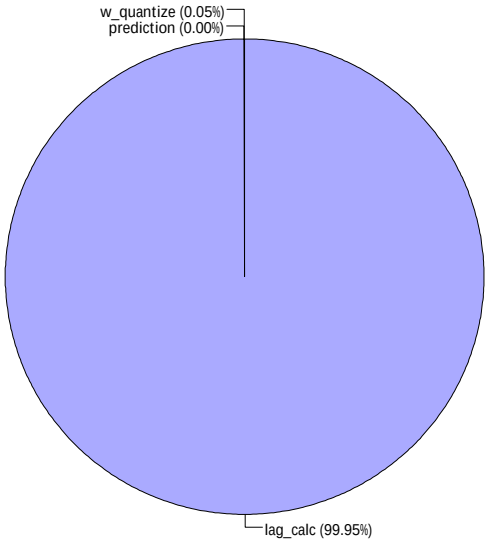

FIGURE 4.6 – Computation time distribution of LTP's *pitch* function (Pentium II)

Within the *pitch* function, we observe that most of the average computation time is spent on the *lag_calc* routine, which searches for the best correlation lag (the one that gives the biggest value of correlation). This routine takes 99.95% of average computation time of the *pitch* function. Therefore, optimizing this single routine should reduce the average computation time of the LTP module, and consequently the average computation time of the AAC encoder.

4.1.3 Average computation time of the Psychoacoustic module

The profiling of the Psychoacoustic module resulted in the graph shown in figure 4.7:

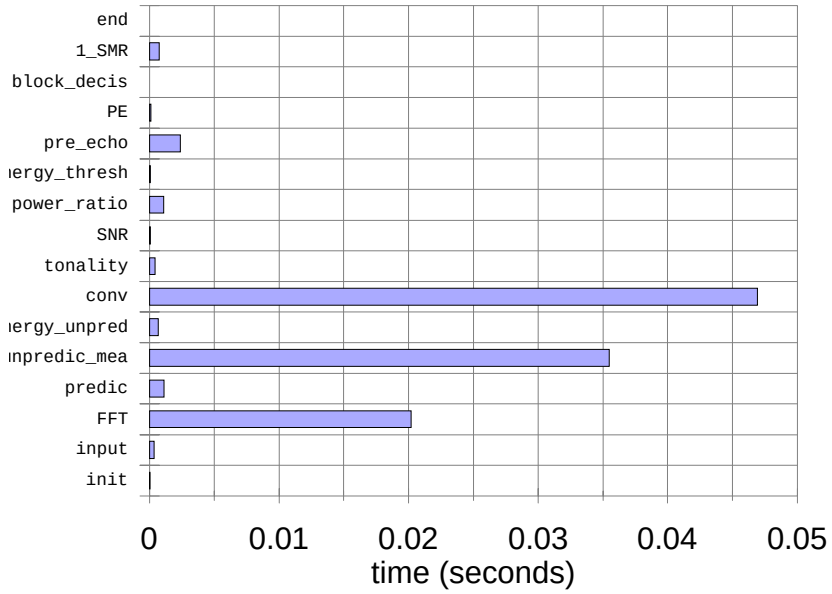

FIGURE 4.7 – Computation time of Psychoacoustic module (Pentium II)

As may be observed, the convolution function (*conv*) takes most of average computation time, followed by the unpredictability measure (*unpredic_mea*) and the FFT function. Figure 4.8 shows the average computation time distribution of the Psychoacoustic module:

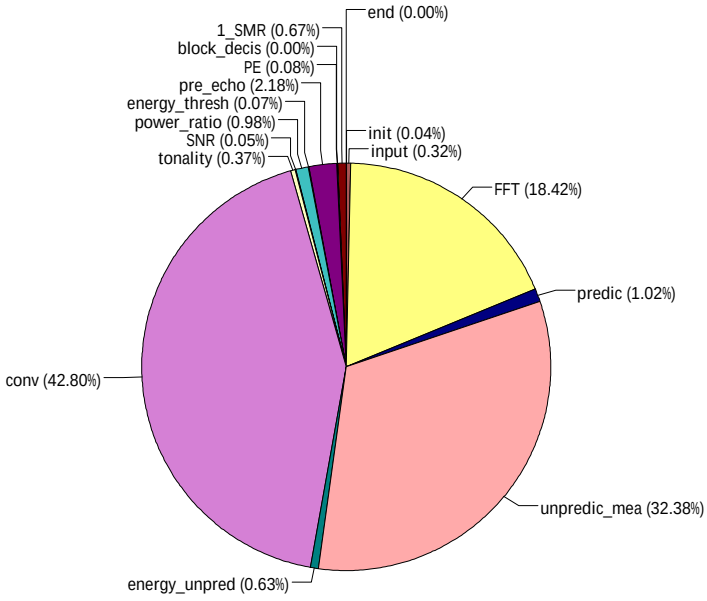

FIGURE 4.8 – Computation time distribution of Psychoacoustic module (PII)

We see that the convolution functions takes 42.80% of average computation time, the unpredictability measure takes 32.38%, and the FFT takes 18.42%. Therefore, these three routines take 93.60% of average computation time and its optimization is needed.

4.2 Assembly implementation

This section describes the Assembly implementation. First, a discussion of SIMD architectures and the MMX architecture is accomplished. Then, results of implementation of some modules are presented and an analysis of the results follows.

4.2.1 Single-data multiple-instructions architectures

The original single-instruction single-data (SIMD) concept was based on multiprocessor architectures [PAT 96]. The idea is that the same instruction is executed by multiple processors using different data streams. The processors are not strictly independent. There is a single instruction memory and a control processor, responsible for fetching and dispatching instructions. The processors receive instructions from the control processors and have their own data memory. Considering that each processor's memory has *different* data, each processor executes the same instructions on multiple data.

SIMD was used in the early multiprocessors, and received more attention in the 1980s. However, multiprocessor architectures are now all based on the multiple-instruction multiple-data (MIMD) concept. This concept is more flexible, since multiple processors can follow the SIMD concept when needed (thereby executing the same instructions) or they can execute different tasks in parallel. Besides, there is a cost-performance advantage, since they can be built by a combination of multiple microprocessors, i.e. there is no need to create a custom microprocessor for MIMD architectures. These advantages of MIMD architectures were the very cause of some lost of interest on SIMD architectures [PAT 96].

The new SIMD concept is not based on multiprocessor architectures, but on a kind of multiprocessing within a single processor. Sometimes, this concept is called SIMD within a register (SWAR) [FIS 99]. In this case, a general-purpose register is divided in multiple fields. Each field corresponds to a new register with shorter word length. A single instruction is applied to the register and is executed in parallel for all register fields. All register fields have their own, independent result. Alternatively, consecutive fields may be concatenated to give the result of the instruction execution. However, the concatenated fields are still independent of each other. This kind of processing is very easy to be adapted to current general-purpose microprocessors. Therefore, there is a growing interest in the SWAR concept.

4.2.2 MMX architecture and instructions

Intel Multimedia Extensions (MMX) architecture [PEL 97] provides a instruction set designed for exploiting the parallelism inherent in many multimedia and communications algorithms. The MMX architecture was introduced in the Pentium processor and its succeeding processor families. It is fully compatible with the Pentium architecture, i.e. no new mode or state was created and all existing software run without modification on a Pentium with MMX technology, even if they are not *MMX-aware*.

The MMX architecture is composed of eight 64-bit MMX registers. To allow backward-compatibility, these registers are mapped into the existing 80-bit floating-point registers. Therefore, no new states are created. However, the drawback is that no mixing of MMX and floating-point instructions in source-codes is allowed. These two kinds of instructions must be well separated in the source-code so that the software runs smoothly.

MMX registers provide SWAR for integer data processing. Each register holds packed data that correspond to independent variables within the same register. For example, register MMX1 may hold 8 bytes (8-bit x 8 = 64 bits) or 4 words. Table 4.2 specifies the supported packed data types:

TABLE 4.2 – MMX packed data types

Data type	***Word-length***	***Capacity***
Packed byte	8-bit	8
Packed word	16-bit	4
Packed doubleword	32-bit	2
Packed quadword	64-bit	1

The MMX architecture features two different types of arithmetic: wraparound and saturation arithmetic. The difference relies in the arithmetical treatment on overflows and underflows. The wraparound arithmetic, which is the most common integer arithmetic found in computer architectures, usually truncates the most significant bit when overflow or underflow occurs. For example, when two large numbers are added, wraparound causes the result to be smaller than the input operands. On the other hand, saturation arithmetic “clips” the result when overflow or underflow occurs, i.e. the result will be the largest or the smallest possible representable number in the data type [PEL 97].

The MMX instruction set is composed by many mathematical instructions: add/subtract instructions, arithmetical and logical shifts instructions, logical instructions, multiply instructions, multiply-add instruction, compare instructions, packing and unpacking instructions, data transfer instructions, and an “empty floating-point registers” instruction. Table 4.3 presents a summary of these instructions:

TABLE 4.3 – MMX instructions

Instruction type	***Opcode***
Add/subtract	padd[b/w/d] psub[b/w/d]
Shift	psra[w/d] psll[w/d] psrl[w/d]
Logical	pand pandn por pxor
Multiply	pmullw pmulhw
Multiply-add	pmaddwd
Compare	pcmpeq[b/w/d] pcmpgt[b/w/d]
Packing/unpacking	packss[wb/dw] punpckl[bw/wd/dq] punpckh[bw/wd/dq]
Data transfer	mov[d/q]
Empty register	emms

As an example of use of the MMX instruction set, an algorithm that computes the dot-product of vectors A and B ($A \cdot B = A_0B_0 + A_1B_1 + A_2B_2 + ... + A_7B_7$) is presented. The following source-code implements the algorithm using MMX instructions:

```
        pxor            mm7, mm7
Loop:   movq            mm0, dword ptr [a_vector]
        movq            mm1, dword ptr [b_vector]
        pmaddwd         mm0, mm1
        padd            mm7, mm0
        add             [a_vector], 8
        add             [b_vector], 8
        sub             count, 4
        jnz             loop
        movq            mm0, mm7
        psrlq           mm7, 32
        padd            mm7, mm0
        movd            dword ptr [mem_vdp], mm7
```

The algorithm works as follows. First, register *mm7* is reset. Then, the main loop begins, where four 16-bit array positions are loaded into registers *mm0* and *mm1*. Multiply-add instruction is performed, and the result is added to register *mm7*. Pointers are updated, and the loop restarts. After the loop, *mm7* holds two partial sums of products: one at the high double word, and the other at low double word. To get the whole result, *mm7* is copied into *mm0*, a logical shift is performed on *mm7*, and *mm0* is added again into *mm7*. Now, *mm7* holds the complete dot-product result. The last instruction transfers the content of the low double word of *mm7* into the memory.

4.2.3 Analysis of precision of the LTP module

An analysis of fixed-point precision was done. LTP's correlation lag routine was converted from its original floating-point version into a fixed-point version, i.e. most floating-point variables (C's *double*) were converted into integer variables (C's *long int*). Through an analysis of precision, it was verified that some routine variables (like *corr1*, *corr2*, *energy*, and so on) should have at least 64-bit precision in order to provide the correct final answer. It was verified that most values of *energy* were in the magnitude order of 10^9.

Since the Intel's MMX architecture does not deal with 64-bit variables in an efficient manner, an alternative scaling scheme was used. In this case, some precision was lost in the least significant bits (LSBs) of the word. Fortunately, this precision lost did not change the final answer, i.e. the MMX version of the routine outputs the same information as the original C version.

4.2.4 Implementation of the LTP module

In this implementation, LTP's correlation lag routine was completely rewritten in Assembly for Intel-PC, using MMX instructions. Microsoft MASM 6.15.8803 was used to assemble the routine. Table 4.4 presents the performance results for this implementation, always considering the computation for one frame. The first row presents the encoder average coding time (using stereo channels) for the optimized version. The second row presents the speed-up in average computation time for the encoder, in comparison to the result of the original implementation. The third row presents the speed-up in average computation time, when comparing the original LTP module to the optimized one. The fourth row presents the speed-up in average computation time, when comparing the original correlation lag routine with the optimized one.

TABLE 4.4 – Average computation time analysis (LTP-optimized version)

Analysis	***PII 350 MHz***	***PIII 750 MHz***
Encoder average coding time (stereo)	0.376642 s	0.198002 s
Encoder computation speed-up	2.2213	1.8479
LTP computation speed-up	6.9448	5.1857
Correlation lag routine speed-up	8.7299	5.0401

Figure 4.9 indicates this reduction in average computation time:

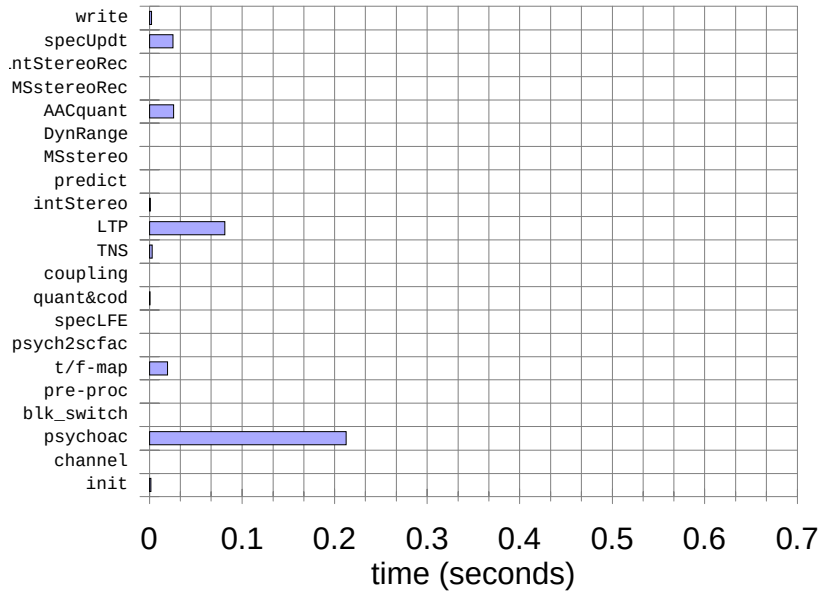

FIGURE 4.9 – Average computation time using optimized LTP module (Pentium II)

Note that because of the optimization of the LTP module, the Psychoacoustic module becomes the most time-consuming module.

4.2.5 Implementation of the Psychoacoustic module

The optimizations in the Psychoacoustic module consisted of hand-coding in Assembly of computationally intensive routines. However, the floating-point variables of the original code were not converted to integer variables. Therefore, MMX instructions were not used. For example, by hand coding the unpredictability measure routine, a speed-up of 5.4753 times (for Pentium II 350 MHz) in the average computation time of this routine was accomplished.

Besides, a lookup table scheme was used for the spreading function. That is, instead of reusing the function each time a new value was needed, a simple table look-up was enough to get the new value. Since only a limited set of points is calculated and used for this function, these values may be pre-calculated and stored in a table. Moreover, there is no lost of precision and no interpolation is required. Two lookup tables were used: one for the long window, and one for the short window. The lookup tables for the spreading function take 40.5 kbytes (for the long window) and 18 kbytes (for the short window). The use of these tables leads to a speed-up of 8.7286 times (for Pentium II 350 MHz) in the average computation time of the convolution routine.

Table 4.5 presents the results of the average computation time analysis for this implementation (seconds per frame), considering all optimization (of both modules):

TABLE 4.5 – Average computation time analysis (Assembly-optimized version)

Analysis	***PII 350 MHz***	***PIII 750 MHz***
Encoder average coding time (stereo)	0.217543 s	0.143212 s
Encoder computation speed-up	3.8458	2.5549
Psychoacoustic module speed-up	2.4749	2.3427

The first row presents the encoder average coding time (using stereo channels) for the optimized version (considering both optimizations on the LTP and Psychoacoustic modules). The second row presents the speed-up in average computation time for the encoder, comparing to the result of the original implementation. The third row presents the speed-up in average computation time, when comparing the original Psychoacoustic module to the optimized one.

Figure 4.10 shows the average computation time within the optimized Psychoacoustic module:

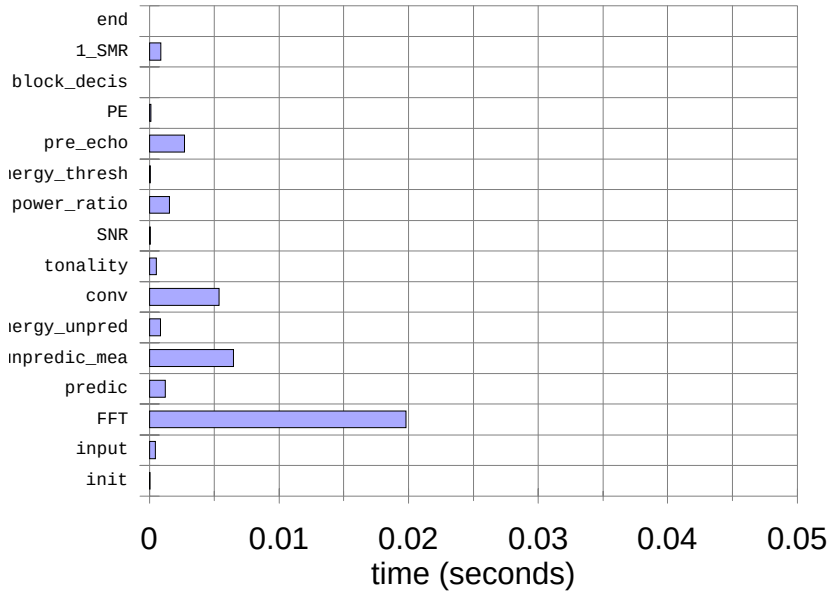

FIGURE 4.10 – Computation time within the optimized Psychoacoustic module (PII)

Figure 4.11 shows the average computation time for this optimized version of the AAC encoder:

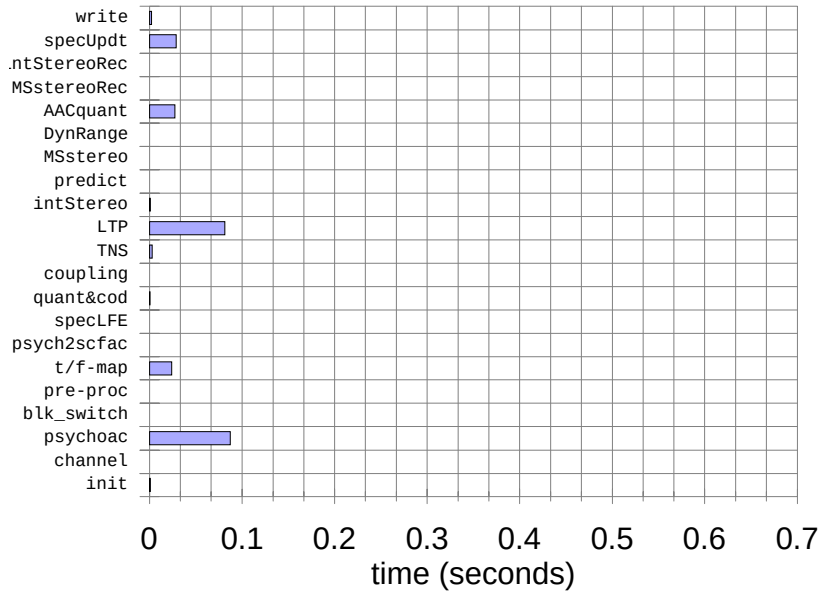

FIGURE 4.11 – Average computation time for each module (optimized version, PII)

This figure indicates that, although optimizations were done on the most time-consuming modules, they remain the most time-consuming modules.

4.3 DSP implementation of the LTP module

This section presents the DSP implementation of the LTP module. It begins with an overview of DSP architectures characteristics, a brief overview of the Texas C31 and the Motorola DSP56309 architectures, an overview of other DSP implementations of the AAC encoder, the results of this work's implementation, and comments to these results.

4.3.1 DSP architectures

Digital signal processors, as the name implies, are used to boost up the performance of digital signal processing algorithms and as a basis for embedded signal processing applications. These architectures are optimized for the repetitive nature of sample-by-sample processing algorithms. They provide fast operators (multiplication, accumulation), and efficient performance for memory moves [ORF 96].

The most visible feature of DSP architectures is the multiply-acummulate (MAC) operators, which perform $reg = reg + i1 \cdot i2$ usually within one clock cycle. Since this kind of operation is very common in DSP algorithms, the use of MAC operators enhances the performance of implementations of DSP algorithms. In addition, many DSP processors provide parallelization of operations, so that two or more operations are performed in the same clock cycle.

4.3.2 Texas C31 architecture

The Texas TMS320C3x (such as the ‘C30, ‘C31, and others) contains a register-based CPU architecture, which is composed by a floating-point/integer multiplier, an arithmetic-logic unit (ALU), a 32-bit barrel shifter, internal data, address and DMA buses, auxiliary register arithmetic units, and a register file [TEX 97].

The register file is composed by 28 registers. Among them, there are 8 extended precision register that can store 32-bit integer or 40-bit floating point numbers, 8 auxiliary registers that store 32-bit numbers and can be used to generate 24-bit addresses.

The C31 memory is composed by two RAM blocks that hold 1k x 32 bits and support two accesses in a single cycle. Four groups of addressing modes are available: general instruction addressing modes (register, short immediate, direct, and indirect), 3-operand instruction addressing modes (register and indirect), parallel instruction addressing modes (register and indirect), and branch instruction addressing modes (register, and PC-relative).

4.3.3 Motorola DSP56300 architecture

The Motorola DSP56300 is a 24-bit digital signal processor that features a highly parallel instruction set, a fully pipelined parallel MAC, a data arithmetic logic unit (data ALU), a 56-bit parallel barrel shifter, an address generation unit (AGU), direct memory access (DMA) controller, program RAM, instruction cache, and two data RAM [MOT 2000].

The architecture owns 6 data ALU registers that may be concatenated into two 56-bit general-purpose accumulators. In fact, the accumulators are 48-bit registers with extension of 8 bits. This extension helps on the minimization of errors due to overflow.

The memory space is divided into program memory, and X and Y data memory. The size of each space is programmable. Besides, the memory space may be expanded by using off-chip memory.

4.3.4 Overview of other AAC implementations

Hilpert et al [HIL 98] worked on two real-time implementations of the MPEG-2 AAC algorithm using DSP processors (the Analog Devices ADSP-21060 and the Motorola DSP563xx-80). The Low Complexity (LC) profile of the MPEG-2 AAC algorithm was employed. An interesting and efficient fixed-point implementation of the Psychoacoustic module is described in this paper. However, in terms of average computation time reduction, no explicit comparisons between the original C code from ISO and the optimized one is made.

Chen and Tai [CHE 99] presented an implementation of the MPEG-2 AAC coder on Texas Instruments TMS320C26x fixed-point DSP processor. They implemented all profiles of the algorithm (Main, LC, and Scalable Sampling Rate), and they have chosen the last two profiles for real-time implementation. Nevertheless, no specific optimization technique is described in this paper.

Huang et al [HUA 99] showed an implementation of the MPEG-4 AAC encoder on Analog Devices ADSP-21060 SHARC. The authors cite optimizations made in the original source-code provided by ISO for the DSP implementation. However, no specific optimization in the Psychoacoustic module is mentioned. Moreover, since this is an implementation of the Low Complexity profile, the LTP module is not used.

Hilpert et al [HIL 2000] also worked on the implementation of the MPEG-4 AAC algorithm on the Motorola DSP56300. This work is a progression of the implementation presented at the AES 105th Convention [HIL 98]. Again, several optimization techniques are described. Nevertheless, no comparison between the original and the optimized implementation is made. The authors also mention that the LTP module was not yet implemented.

4.3.5 Implementation results on the Texas TMS320C31

Texas Instruments TMS320C31 floating-point processor was chosen for the first DSP implementation. The choice was based on availability and convenience. Texas Instruments Code Composer was used for developing C code. This application provides C compilation and linking of C and Assembly routines. Besides, it provides simulation of code execution. This simulation is optimistic, i.e. the results may have a better (lower) average computation time than the one resulting from real execution of the code on the C31 processor.

To get the simulated average computation time, a 25 MHz processor was considered (clock cycle of 40 ns). Other choices of processor frequency were available, e.g. 16.6 MHz, 20 MHz, 25 MHz, 30 MHz e 40 MHz (corresponding to clock periods of 60 ns, 50 ns, 40 ns, 33 ns and 25 ns).

Two implementations of the LTP’s correlation lag routine were taken into consideration. Both were coded using C language. The first one is the C version that was already used in the Pentium II and III implementations. This implementation uses floating-point variables, and will be referred as “floating-point” version in the following discussion. The second implementation is an integer version (also coded in C) of this routine. This implementation employs integer variables in place of floating-point variables, when possible.

Table 4.6 presents the simulation results for the floating-point version of the LTP's correlation lag routine:

TABLE 4.6 – Performance of floating-point LTP's correlation lag routine

	Cycles	***Time (at 25 MHz)***
Average computation (total)	50,835,140.0	2.0334 s
Average (per loop)	24,821.8	0.9929 ms
Maximum (per loop)	49,379.0	1.9752 ms
Minimum (per loop)	249.0	9.96 µs

As shown in the table, the average computation time is 2.033 s. This is considerably greater than the average computation time of the same routine on Pentium II 350 MHz. Table 4.7 presents a comparison among implementations, considering the average computation time of the LTP's correlation lag routine (*corr_lag*):

TABLE 4.7 – Comput. time comparison (Pentium vs. C31, floating-point version)

Analysis	***PII 350 MHz***	***PIII 750 MHz***	***C31***
Corr_lag average coding time (C version)	0.308725 s	0.136255 s	2.0334 s
Corr_lag average coding time (Assembly version)	0.035364 s	0.027034 s	–
Speed-up of Pentium implem. (C version)	6.5851	14.9206	–
Speed-up of Pentium implementation (Assembly version)	57.4878	75.2016	–

The first row presents the average computation time for the Pentium II, Pentium III, and C31 implementations, considering the C version. The second row presents the average computation time for the Pentium II and Pentium III, considering the Assembly version. The last two rows present the speed-up of the PC implementations over the C31 implementation.

In the integer version of the C code, the average computation time of the LTP's correlation lag routine is better (i.e. lower). Table 4.8 presents the simulation results of this version:

TABLE 4.8 – Performance of integer LTP's correlation lag routine

	Cycles	***Time (at 25 MHz)***
Average computation (total)	26,196,824.0	1.0479 s
Average (per loop)	12,791.4	0.5117 ms
Maximum (per loop)	26,859.0	1.0744 ms
Minimum (per loop)	247.0	9.88 µs

Because of the modifications, the average computation time of the routine had a speed-up of 1.9399 times. Despite this fact, this result is still greater than the average computation time on a Pentium II 350 MHz. The following table presents a new comparison among implementations.

TABLE 4.9 – Computation time comparison (Pentium vs. C31, integer version)

Analysis	***PII 350 MHz***	***PIII 750 MHz***	***C31***
Corr_lag average coding time (C version)	0.308725 s	0.136255 s	1.0479 s
Corr_lag average coding time (Assembly version)	0.035364 s	0.027034 s	–
Speed-up of Pentium implem. (C version)	3.3946	7.6915	–

Analysis	*PII 350 MHz*	*PIII 750 MHz*	*C31*
Speed-up of Pentium implementation (Assembly version)	29.6347	38.7660	–

The first row shows the average computation time for the Pentium II, Pentium III, and C31 integer implementations, considering the C version. The second row presents the average computation time for the Pentium II and Pentium III, considering the Assembly version. The last two rows present the speed-up of the PC implementations over the C31 implementation.

We may compare the results of both implementations by *normalizing* the C31 implementation results. This is accomplished by matching the clock frequency of the implementations. In this case, this corresponds to extrapolate the clock frequency of the C31 processor to equal the clock frequency of the Pentium II and III (350 MHz and 750 MHz, respectively).

The performance of the DSP implementation is enhanced when considering normalized clock frequencies. Table 4.10 presents the normalized comparison for the C31 floating-point version. We may observe that the *normalized performance* of the C31 floating-point version is better than the performance of the C (floating-point) version of the Pentium II and III implementations. However, the performance of the Assembly (MMX) version is better than the performance of the C31 floating-point version.

TABLE 4.10 – Normalized comparison (Pentium vs. C31, floating-point version)

Analysis	*PII 350 MHz*	*PIII 750 MHz*
Corr_lag average coding time (C version)	0.308725 s	0.136255 s
Corr_lag average coding time (Assembly version)	0.035364 s	0.027034 s
Normalized C31 computation time	0.145243 s	0.067780 s
Speed-up of Pentium implem. (C version)	0.4705	0.4975
Speed-up of Pentium implem. (Assembly version)	4.1071	2.5072

Table 4.11 presents the normalized comparison for the C31 integer version. In this case, we observe again that the performance of the Assembly version of the Pentium implementation is better than the *normalized performance* of the C31 integer version.

TABLE 4.11 – Normalized comparison (Pentium vs. C31, integer version)

Analysis	*PII 350 MHz*	*PIII 750 MHz*
Corr_lag average coding time (C version)	0.308725 s	0.136255 s
Corr_lag average coding time (Assembly version)	0.035364 s	0.027034 s
Normalized C31 computation time	0.074848 s	0.034929 s
Speed-up of Pentium implem. (C version)	0.2424	0.2564
Speed-up of Pentium implem. (Assembly version)	2.1165	1.2920

As observed in the tables, the performance of the Assembly version of the Pentium implementation is better than the *normalized performance* of both versions of

the C31 implementation. This may be a sign that the MMX architecture is better for integer processing than the C31 architecture.

4.3.6 Implementation results on the Motorola DSP56309

The floating-point and the integer implementations used in the C31 processor have run without changes in the Motorola DSP56309 processor. Table 4.12 presents the results of the implementation on the DSP56309 processor:

TABLE 4.12 – Performance of implementations on the Motorola DSP56309

	Cycles	***Time (66 MHz)***
Computation time (floating-point version)	447,643,529	6.782478 s
Computation time (integer version)	47,597,705	0.721177 s
Speed-up (integer version)	–	9.4047

It is interesting to compare the performance of the two DSP processors (Texas C31 versus Motorola DSP56309). Table 4.13 presents this comparison. We may observe that the performance of the C31 is better than that of the DSP56309 for the floating-point version, and the performance of the DSP56309 is better than that of the C31 for the integer version. This could be justified by noting that the C31 is a floating-point processor, and the DSP56309 is an integer-point processor. However, this statement is false when normalized comparison is considered, as will be discussed below.

TABLE 4.13 – Comparison between DSPs (C31 vs. DSP56309)

Analysis	***C31***	***DSP56309***	***Speed-up (C31)***
Computation time (floating-point version)	2.0334 s	6.782478 s	3.3355
Computation time (integer version)	1.0479 s	0.721177 s	0.6882

We may normalize the results of the C31 processor, by considering the clock frequency of the DSP56309 processor (66 MHz). Table 4.14 presents this normalized comparison. In this case, the performance of the C31 is better than the performance of the DSP56309 in both floating-point and integer versions. The enhanced performance of the C31 may be justified by a better architecture of this processor, or a more efficient code generation of the C compiler.

TABLE 4.14 – Normalized comparison between DSPs (C31 vs. DSP56309)

Analysis	***C31***	***DSP56309***	***Speed-up (C31)***
Computation time (floating-point version)	0.7702 s	6.782478 s	8.8058
Computation time (integer version)	0.3969 s	0.721177 s	1.8169

Table 4.15 presents a computation time comparison among the Pentium II and III implementations and the DSP56309 implementation, considering the floating-point version:

TABLE 4.15 – Comput. time comparison (Pentium vs. 56309, floating-point version)

Analysis	***PII 350 MHz***	***PIII 750 MHz***	***56309***
Corr_lag average coding time (C version)	0.308725 s	0.136255 s	6.78248 s
Corr_lag average coding time (Assembly version)	0.035364 s	0.027034 s	–
Speed-up of Pentium implem. (C version)	21.9693	49.7778	–
Speed-up of Pentium implementation (Assembly version)	191.7905	250.8869	–

Table 4.16 presents a computation time comparison among the Pentium II and III implementations and the DSP56309 implementation, considering the integer version:

TABLE 4.16 – Comput. time comparison (Pentium vs. 56309, integer version)

Analysis	***PII 350 MHz***	***PIII 750 MHz***	***56309***
Corr_lag average coding time (C version)	0.308725 s	0.136255 s	0.72118 s
Corr_lag average coding time (Assembly version)	0.035364 s	0.027034 s	–
Speed-up of Pentium implem. (C version)	2.3360	5.2929	–
Speed-up of Pentium implementation (Assembly version)	20.3930	26.6767	–

The comparisons presented in table 4.15 and table 4.16 are both unfavorable for the DSP56309 processor. The performance of Pentium implementations remains better than that of the DSP56309 when normalizing the results of the DSP56309 floating-point version. Table 4.17 presents the normalized comparison:

TABLE 4.17 – Normalized compar. (Pentium vs. DSP56309, floating-point version)

Analysis	***PII 350 MHz***	***PIII 750 MHz***
Corr_lag average coding time (C version)	0.308725 s	0.136255 s
Corr_lag average coding time (Assembly version)	0.035364 s	0.027034 s
Normalized DSP56309 computation time	1.278981 s	0.596858 s
Speed-up of Pentium implem. (C version)	4.1428	4.3804
Speed-up of Pentium implem. (Assembly version)	36.1662	22.0781

The performance of the DSP56309 is only better than the performance of the C version of the Pentium implementations. When the Pentium Assembly implementation is considered, the performance of the Pentium processor is again better than the performance of the DSP56309. This is presented in table 4.18.

TABLE 4.18 – Normalized compar. (Pentium vs. DSP56309, integer version)

Analysis	***PII 350 MHz***	***PIII 750 MHz***
Corr_lag average coding time (C version)	0.308725 s	0.136255 s
Corr_lag average coding time (Assembly version)	0.035364 s	0.027034 s
Normalized DSP56309 computation time	0.135993 s	0.063464 s
Speed-up of Pentium implem. (C version)	0.4405	0.4658
Speed-up of Pentium implem. (Assembly version)	3.8455	2.3475

In summary, the comparisons presented in the previous tables indicate that the performance of the DSP56309 is generally worse than the performance of the Pentium

II and III. The performance of the DSP56309 is only better than that of the Pentium II and III when the specific case of the *normalized comparison* between the Pentium C version and the DSP56309 integer version is considered.

4.3.7 Comments to the results of the DSP implementations

One usually expects that DSP processors run signal processing algorithms more efficiently than a general-purpose processor. That was not the case for the LTP module implementation on DSP processor. Although the *normalized performance* of the C31 floating-point version is better than the performance of the Pentium C version, the comparison of the results are in general more favorable for the Pentium processors.

However, this may be not so surprising. General-purpose processors with DSP enhancements are becoming a serious threat to DSP processor vendors. A comparison among current DSP processors and general-purpose processors shows, for example, that the performance of the Intel Pentium III 1.3 GHz beats the performance of processors such as Analog Devices ADSP-218x, ADSP-2106x, ADSP-2116x, Lucent DSP16410, Motorola DSP563xx, and Texas Instruments TMS320C54xx (for a benchmark with 256-point FFT and an FIR filter) [EYR 2001]. This demonstrates that, for some kind of applications that do not require specific embedded application constraints (e.g. power, compactness, portability, cost, etc), the use of a general-purpose processor may be worthwhile.

However, when embedded systems needs (such as low unitary cost and low-power consumption) are taken into consideration, the situation changes. General-purpose processors are not usually suitable for embedded applications, since many of them have high unitary cost and high power consumption. Therefore, when designing an embedded system, DSP are usually preferred in place of general-purpose processors.

4.4 HDL implementation of the LTP module

VHDL [CHA 99] is a language used for hardware description and simulation. It is very suitable for the description of algorithmic circuits (such as microprocessors or finite automata). This kind of circuits may be modeled using algorithmic-state machine charts [GAJ 97].

The following subsections present an overview of Altera's FPGA, an overview of embedded applications, and some details of the VHDL implementation.

4.4.1 Altera FPGA and synthesis process

For the simulation of the VHDL description, Altera's Flex10k devices were used [ALT 2001]. Each FLEX 10K device contains an embedded array and a logic array. By the combination of embedded and logic arrays, systems may be implemented into a single device.

The embedded array is used to implement several types of memory functions or complex logic functions (such as digital signal processing, microcontroller, etc). It consists of a series of embedded array blocks, each providing 2048 bits.

The logic array performs the same function as the sea-of-gates in the gate array, i.e. it is used to implement general logic (such as counters, adders, state machines, and multiplexers). It consists of logic array blocks, each consisting by its turn of eight logic elements and local interconnect. A logic element consists of a 4-input lookup table, a programmable flip-flop, and dedicated paths for carry and cascade functions.

4.4.2 Embedded applications

Embedded systems are based on the integration of hardware and software for a specific application. There are many ways to design embedded systems. One way is building an application-specific integrated circuit (ASIC), which requires large-scale production to lower the unitary cost of the chip design and fabrication. Alternatively, the circuit may be described in a hardware description language and synthesized on an FPGA. The FPGA version is largely preferred in low-cost and low performance (or just-needed performance) applications. In both cases, there is a custom designed circuit.

However, instead of designing custom circuits, largely available embedded microprocessors may be employed. These microprocessors are usually especially designed for embedded applications, taking into consideration factors such as dimension, power consumption, and unitary cost. Examples of current embedded microprocessors include ARM [ARM 2001], MIPS [MIP 2001], and PowerPC [IBM 2001]. The complete embedded system is built by the combination of embedded microprocessor and a software layer.

Some complex embedded systems may require an operating system. For embedded applications, real-time operating systems are usually employed. This kind of operating systems is optimized for real-time applications, such as interactive multimedia applications. Examples of current real-time operating systems are VxWorks [WIN 2001], and Microsoft CE [MIC 2001].

4.4.3 Overview of the VHDL implementations

The LTP correlation lag routine was described as an algorithmic machine in VHDL. The description was synthesized for the Altera's Flex10k device family. Altera's MaxplusII software was used for synthesis and simulation of the description.

Three different descriptions were developed in this work, all written in Altera VHDL. The difference among them are indicated by:

- The use of a separate *ltp_lag_proc* or integration (merging) of the *ltp_lag_proc* description into the main processor. In the separate version, the *ltp_lag_proc* is responsible for performing the routine mathematical

operations, while the main processor reads data from memory and specifies the state of the *ltp_lag_proc* processor. In the merged version, there is only one main processor (*ltp_lag*) that reads data and performs calculations.

- The use of separately designed multiply-and-accumulate (MAC) units or the use of the Altera's standard operators (provided in standard libraries or so-called mega-functions). The specially designed MAC units are based on bit-serial adders and multipliers.

Table 4.19 summarizes the differences among the implementations:

TABLE 4.19 – LTP VHDL descriptions

Implementation	***lpt_lag_proc***	***Operators***
1	Separate processor	MAC units
2	Integrated into *ltp_lag*	MAC units
3	Integrated into *ltp_lag*	Altera's operators library

This section has presented an overview of the implementations. The next section provides some details of them.

4.4.4 Description of the VHDL implementations

In this section, the units mentioned in the previous section (section 4.4.3) are presented in details. This section presents the MAC unit, the *ltp_lag_proc* processor, and the *ltp_lag* processor.

The *n*-bit MAC (*multiply-accumulate*) unit is responsible for performing multiply-accumulate operations using bit-serial arithmetic. This kind of arithmetic allows for lower area usage (indicated by the number of cells, in the case of FPGAs) than the parallel arithmetic. Figure 4.12 presents a simplified diagram of this unit. Note that the MAC unit is used in implementations *one* and *two*. Implementation *three* employs adders and multipliers from the Altera's operators library.

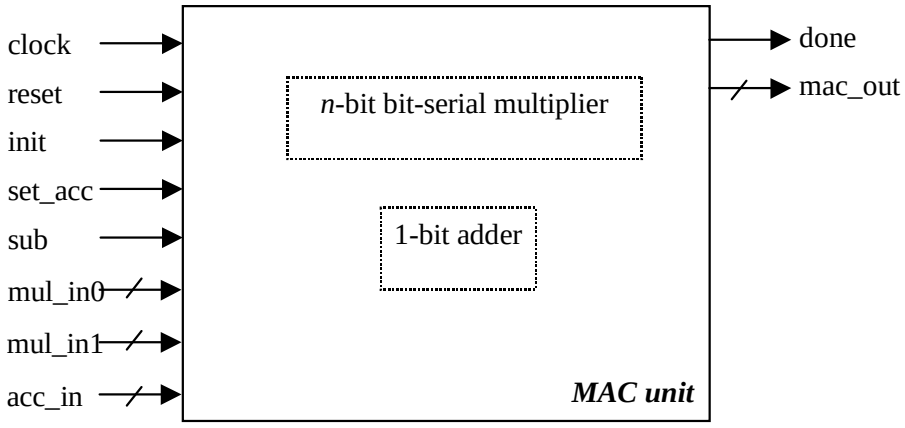

FIGURE 4.12 – Simplified diagram of the *n*-bit MAC unit

As indicated by the figure, the MAC unit uses one 1-bit adder and an *n*-bit bit-serial multiplier. The *n*-bit bit-serial multiplier is composed by *n* bit-serial multipliers. The *clock* input receives the clock. The *reset* input allows for resetting the contents of the MAC unit. The *init* input starts the multiply-accumulate operation. The *set_acc* input allows for the MAC unit to be set by the value indicated at input *acc_in*. When the *sub* input is on, the sign of the *mul_in1* is reversed. The inputs *mul_in0* and *mul_in1* are the input values to the MAC unit.

The *ltp_lag_proc* processor is responsible for performing the operations of the correlation routine. In other words, its main purpose is to transfer the data received from inputs *i0* and *i1* to the MAC unit and trigger this unit according to the operation to be performed. In other words, this processor controls the datapath. Figure 4.13 presents a simplified diagram of the *ltp_lag_proc* processor:

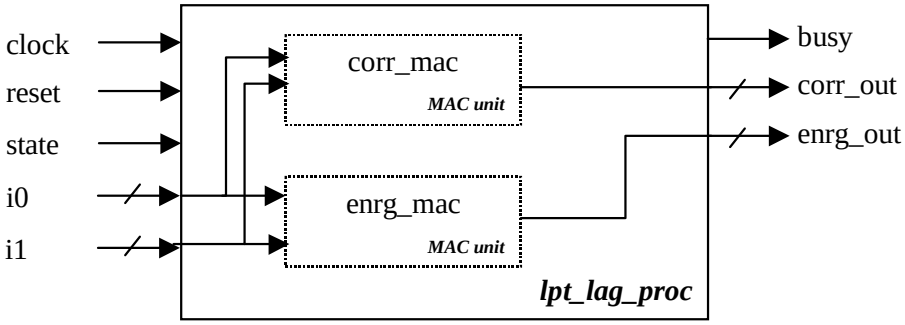

FIGURE 4.13 – Simplified diagram of *ltp_lag_proc* processor

Note that the *ltp_lag_proc* has a *state* input. This indicates that this processor does not own a state machine. It controls the datapath according to the external state. This state is received from the *ltp_lag* processor, which is the main processor. In this situation, the *ltp_lag* processor is mainly responsible for controlling the state machine, and receiving data from memory. Figure 4.14 presents a simplified diagram that illustrates the relation between the *ltp_lag_proc* and the *ltp_lag* processors:

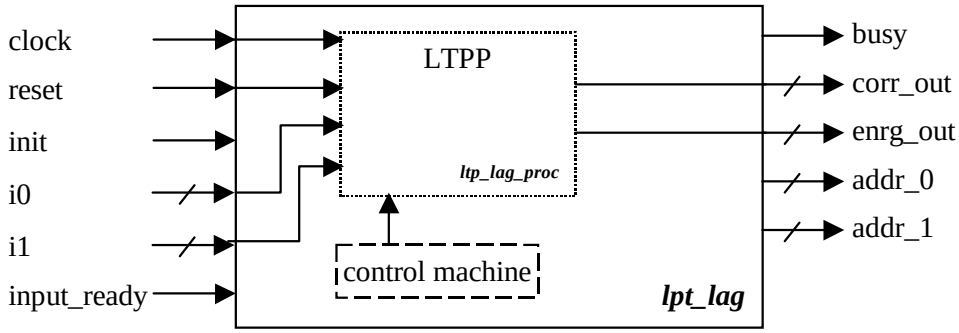

FIGURE 4.14 – Simplified diagram of *ltp_lag* processor (implementation 1)

The *ltp_lag_proc* processor is used just in implementation *one*. In the other implementations, this processor is merged into the *ltp_lag* processor, i.e. the

ltp_lag_proc description is merged into the *ltp_lag* description. Therefore, in implementations *two* and *three*, the *ltp_lag* processor is not only responsible for the state machine, but also for controlling the datapath.

The state machine of the algorithm within the *lag_proc* processor is not very simple (when comparing it to the state machine of the MAC unit, for example), as may be observed in figure 4.15. This is a simplified state machine of the implementation *three*. The state machines of the other implementations are similar to this one.

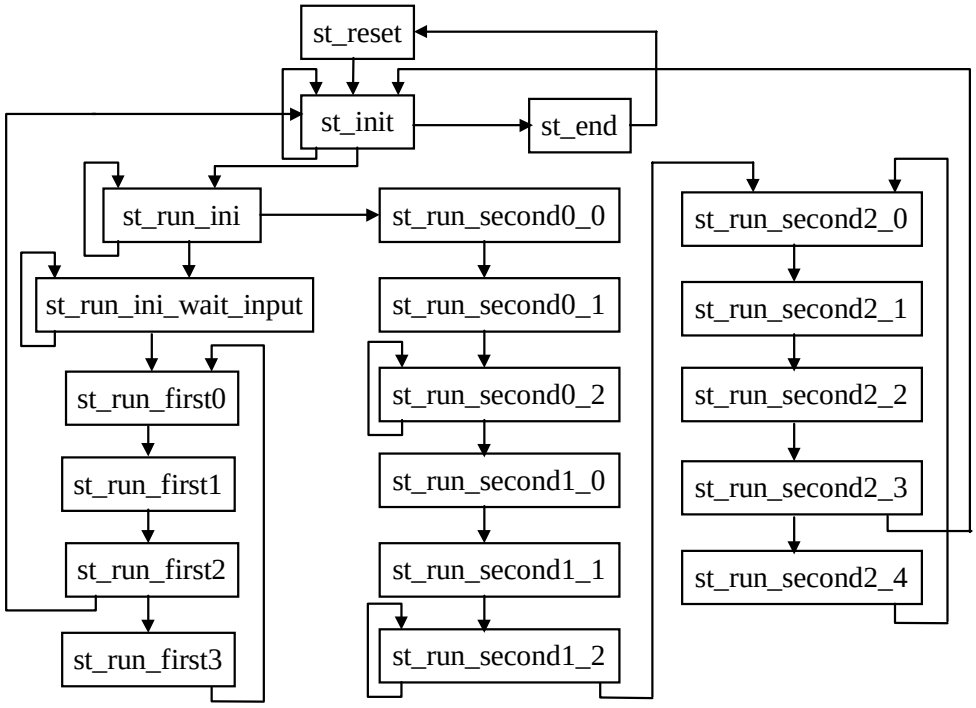

FIGURE 4.15 – State machine of implementation *three*

As may be observed in figure 4.15, the state machine has 20 states, and many possible paths. This complexity contributes to the low maximum frequency achieved in the FPGA synthesis, as discussed below.

Table 4.20 indicates the FPGA synthesis results (number of cells and maximum frequency) of each unit. For the synthesis, Altera's EPF10K30RC240-3 device was used.

TABLE 4.20 – FPGA synthesis of units

Unit	***Description***	***Number of cells***	***Maximum frequency***
add_1bs	1-bit adder	3	125.00 MHz
m_a_1bs	1-bit multiplier	5	125.00 MHz
mul_bs	16-bit multiplier	48	125.00 MHz
mac_bs	16-bit MAC unit	243	56.81 MHz
ltp_lag_proc	LTP processor	849	40.81 MHz

Table 4.21 presents the FPGA synthesis results (number of cells and maximum frequency) of the implementations (i.e. the results for the *ltp_lag* processor). For the synthesis, Altera's EPF10K30RC240-3 device (implementations *one* and *two*) and EPF10K40RC240-3 device (implementation *three*) were selected.

TABLE 4.21 – FPGA synthesis of implementations

Implementation	*Number of cells*	*Maximum frequency*
1	1653	12.54 MHz
2	1597	16.31 MHz
3	2250	7.99 MHz

This section has presented some details of the VHDL implementations. The next section provides details of the implementation results.

4.4.5 VHDL implementation results

The three implementations were simulated using Altera's MaxplusII 9.26 software. Table 4.22 table presents the number of cycles needed to compute the correlation lag routine for one frame, the maximum frequency, the computation time, and the speed-up of implementation *three* in relation to the other implementations:

TABLE 4.22 – Performance of the VHDL implementation

Implementation	*Number of Cycles*	*Maximum clock frequency*	*Computation time (s)*	*Speed-up of implem. 3*
1	104,958,902	12.54 MHz	8.369928	5.3109
2	102,334,904	16.31 MHz	6.274366	3.9812
3	12,601,338	7.99 MHz	1.575985	–

In terms of average computation time, the most efficient description was implementation *three*. Comparing the description style of these implementations gives some important indications about efficient FPGA-synthesis-oriented description. It is clear that the major advantage of implementation *three* over the other implementations is its reduced number of cycles needed to compute the correlation lag routine. Although this implementation has the lowest clock frequency, the reduced number of cycles compensates this problem. Therefore, this implementation has the lowest computation time.

The reduced number of cycles may be attributed to two reasons. First, implementation *three* employs optimized operators from Altera's standard operators library in place of the bit-serial MAC unit. The use of bit-serial arithmetic may be advantageous for low-power and low area applications, but it requires a high clock frequency, at least many times higher than that of parallel arithmetic. This high frequency may not be achievable on FPGAs. Among the implementations, the maximum clock frequency of implementations *one* and *two* was not many times higher than the maximum clock frequency of implementation *three*. Therefore, implementation *three* presented the lowest computation time.

A second reason for the reduced number of cycles in implementation *three* is its simplified state machine. When compared to the other implementations, implementation *three* presents the lowest number of states. Although the synthesis of this state machine clearly requires a large number of FPGA cells, the reduced number of states leads to a reduced number of cycles.

In comparison with implementations on other platforms, the VHDL implementation *three* is not very successful. Table 4.23 presents this comparison, featuring the clock frequency, the computation time, and the speed-up (in relation to the VHDL implementation *three*) of each implementation:

TABLE 4.23 – Comparison among LTP module implementations

Implementation	***Clock frequency (MHz)***	***Computation time (s)***	***Speed-up (to hardware impl.)***
PC C version (PII)	350.00	0.308725	5.1086
PC C version (PIII)	750.00	0.136255	11.5749
PC MMX version (PII)	350.00	0.035364	44.5973
PC MMX version (PIII)	750.00	0.027034	58.3391
56309 integer version	66.00	0.721177	2.1869
C31 integer version	25.00	1.047873	1.5051
56309 floating-point version	66.00	6.782478	0.2325
C31 floating-point version	25.00	2.033406	0.7756
VHDL version (impl. 3)	7.99	1.575985	–

Figure 4.16 illustrates the comparison among implementations:

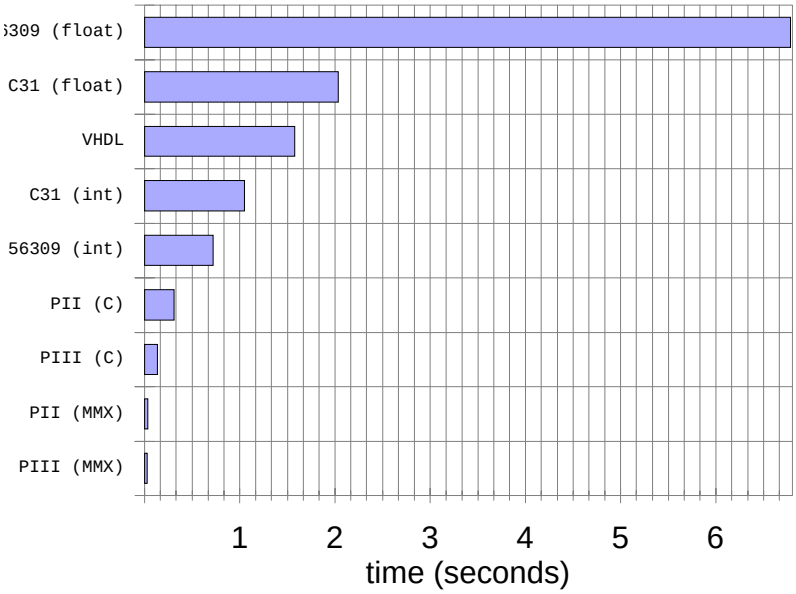

FIGURE 4.16 – Computation time comparison (LTP module, all implementations)

The table 4.23 and the figure 4.16 reveal that, except for the DSPs floating-point versions, the VHDL version is slower than all remaining implementations. There are many reasons that contribute to this unfavorable comparison of the VHDL version

performance with respect to the general-purpose processor (e.g. Pentium II, III, and its variations). One reason is the low clock frequency of the FPGA in comparison with the clock frequency of other implementations. Although the number of cycles required for computation on the VHDL (12,601,338) is lower than, for example, the number of cycles required for the C31 integer version (26,196,824), the low clock frequency achieved on the FPGA synthesis raises its computation time. The normalization of all computation times for the Pentium II clock frequency (350 MHz) clarifies the situation. Table 4.24 presents this normalization:

TABLE 4.24 – Comparison among LTP implementations (normalized for 350 MHz)

Implementation	***Normalized computation time (s)***	***Speed-up to hardware implementation***
PC C version (PII)	0.308725	0.116621
PC MMX version (PII)	0.035364	1.018092
C31 floating-point version	0.145243	0.247886
C31 integer version	0.074848	0.481025
56309 floating-point version	1.278982	0.028150
56309 integer version	0.135993	0.264747
VHDL version (impl. 3)	0.036004	–

This table is represented graphically by figure 4.17. It presents the normalization ordered by normalized computation time:

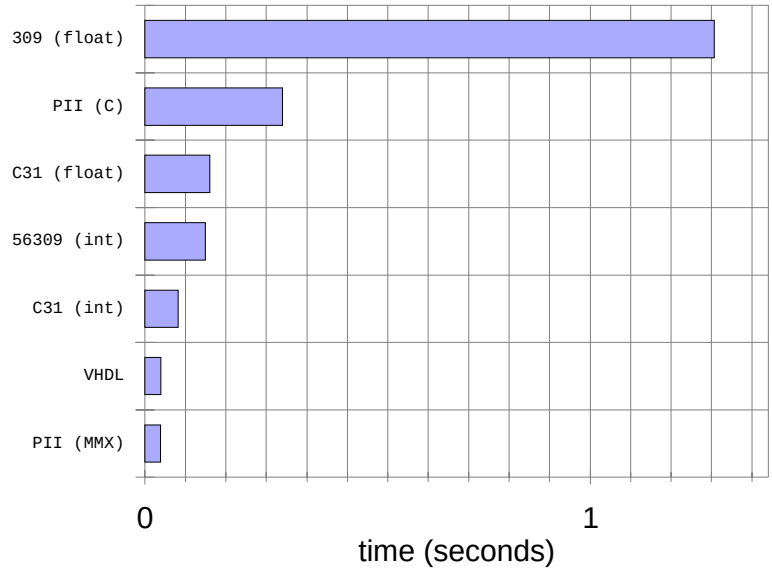

FIGURE 4.17 – Normalized computation time (LTP module, all implementations)

Table 4.24 and figure 4.17 show that, as long as the clock frequency are normalized, the VHDL implementation is one of the best implementations, only slightly worse than the PC MMX version on Pentium II. The normalization of the computation times for the Pentium III clock frequency (750 MHz) reveals that the VHDL implementation may be even better than the Pentium III MMX implementation. Table 4.25 presents this normalization for 750 MHz:

TABLE 4.25 – Comparison among LTP implementations (normalized for 750 MHz)

Implementation	***Normalized computation time (s)***	***Speed-up to hardware implementation***
PC C version (PIII)	0.136255	0.123311
PC MMX version (PIII)	0.027034	0.621506
C31 floating-point version	0.067780	0.247886
C31 integer version	0.034929	0.481025
56309 floating-point version	0.596858	0.028150
56309 integer version	0.063464	0.264747
VHDL version (impl. 3)	0.016802	–

As may be observed in the table 4.24 and the table 4.25, the major problem of the VHDL implementation is its low clock frequency. The main reason for this low clock frequency may be the description style, which may not be appropriate for the LTP algorithm. Thus, the behavioral description of the LTP leads to a synthesized hardware having a lower clock frequency than that of a dedicated ASIC, which has its architecture optimized for the algorithm. In summary, the choice between synthesizing the VHDL description into an FPGA or mapping this description, for example, into an ASIC using standard-cell is very important, since it may define the ultimate performance of the implementation of the algorithm.

These bad results for the FPGA implementation also raise the question concerning the model used for embedded application, i.e. whether it is profitable to create a custom FPGA-synthesized description in VHDL, or build the system as software upon an embedded microprocessor. The answer would be given by comparing the FPGA implementation with an embedded-processor implementation, as mentioned in section 4.4.2. Although this implementation on embedded processor was not developed in this work, it would provide a larger basis for comparison among implementation platforms.

5 Conclusion

In this work, the MPEG-4 AAC standard was studied, resulting in a tutorial text covering all aspects of the AAC encoder. Besides, some principles of Psychoacoustics were presented in the text, thereby enlightening the study of AAC's perceptual model.

The MPEG-4 AAC is a coding algorithm within the audio part of the ISO MPEG-4 standard for multimedia contents coding and transmission. MPEG-4 AAC is especially designed for low bit-rate coding of general audio (e.g. speech and music). It is an improvement over previous coding algorithms created by the MPEG group, such as the MPEG-1 Layer III. The main components of the AAC encoder are the filterbank, the perceptual model (also called Psychoacoustic module), and the quantization module. The filterbank converts input samples from the time-domain to the spectrum domain (thereby generating spectral coefficients) through a modified cosine transform. The perceptual model uses this spectral-domain data to ensure high quality of the codified audio signal, by providing information about bit-allocation and maximum allowed noise for the quantization process. The spectral data is quantized by AAC quantization process, which is composed by four subblocks: quantization of scalefactors, quantization of spectral coefficients, noiseless coding, and rate/distortion control loop.

The perceptual model is based on guiding rules derived from the field of Psychoacoustics. Masking mechanisms of the human auditory system allows for quantization noise to be added to the codified signal without affecting its quality. Quantization noise is added to an audio signal when it is recoded using fewer bits. Therefore, the masking mechanisms determine the allowed bit reduction that does not degrade the perceived audio quality.

Much of this work was devoted to the implementation of the MPEG-4 AAC encoder. In this work, three different platforms were used and four implementations were developed. The choice of the platforms was guided by current market features, such as availability, portability, cost, etc. For example, while general-purpose processors are largely available, their unitary cost tends to be high, at least much higher than the unitary cost of DSP processor and they are not portable as DSP processor. Therefore, the chosen platforms were general-purpose processors (Intel Pentium II and III), DSP processors (Texas C31 and Motorola DSP56309), and hardware implementation (Altera Flex10k FPGA).

A public-domain version of the MPEG-4 AAC encoder in C language was used as a basis for the development of the implementations. This version of the encoder is very *tutorial*. That is, it exists to guide developers to create their own encoder. Therefore, it is not an efficient implementation of the encoder and it does not reflect the optimizations performed by the audio coding industry in the efforts to design efficient audio encoders, i.e. fast encoders that ensure high coding quality and efficiency.

This C-language code was employed in an analysis of computation time (code profiling). This analysis revealed that the most time-consuming modules of the AAC algorithm were the long-time prediction (LTP) and the Psychoacoustic module. Optimizations on these modules were performed in order to enhance the computation time of the algorithm. Therefore, the implementation work was directed to the development of enhanced (i.e. faster) versions of these modules in several platforms.

The LTP module was developed for all platforms. The implementation results provided some interesting results for comparisons among platforms. For example, this comparison showed that the PC implementations had performance advantages over implementations on other platforms (DSP and FPGA-synthesized hardware). Moreover, the implementations indicated that Assembly optimizations on computation-intensive loops and critical time-consuming routines contribute to significant performance increase. However, comparisons that took normalization of the clock frequency into account showed that the FPGA-synthesized implementation was one of the most efficient implementation, due to its reduced number of cycles needed in the computation of the LTP module. Since DSP processors also have relatively low clock frequency, this was the case for one of them (the C31), too. Therefore, these comparisons revealed that the clock frequency is an important feature of the target platform, as long as computation time is concerned.

Future work using this master thesis as a basis could make the DSP and VHDL implementations more efficient in terms of computation time. In the case of the DSP implementation, this may be achieved by *hand-optimizing* the Assembly code generated by C compiler. However, since Assembly mnemonics vary from processor to processor, this means that a target DSP processor should be chosen at first. Besides, the choice of a target DSP processor should be based on a comparison among different DSP processors, considering their suitability to the development of audio coding algorithms.

In the case of the VHDL implementation, restructuring the control machine of the descriptions by a simplification of the algorithm could provide a more efficient synthesized circuit. Moreover, using other development tools (e.g. standard-cells) could provide better results than the FPGA results.

As another suggestion, embedded processors could be employed for implementation and comparisons, since they may be as low-costing and portable as DSP processors, and as flexible as general-purpose processors.

Appendix 1 Average computation time analysis data

This appendix provides information about the data used for the average computation time analysis, such as the name and characteristics of the sound files used for computation analysis, and tables.

1 Sound files

The following table presents the sound files used for average computation time analysis:

Artists/composer	Song	Start	End	Duration	Samples
M. P. Mussorgsky	Pictures at the Exhibition: Promenade	00:00:07.729	00:00:13.883	00:00:06.154	271,425
M. P. Mussorgsky	The Great Gate of Kiev	00:00:00.296	00:00:06.544	00:00:06.247	275,530
Wagner	The ride of the Valkyries	00:00:00.426	00:00:07.391	00:00:06.964	307,152
Vivaldi	The four seasons: Spring I	00:01:55.995	00:02:01.315	00:00:05.319	234,576
Beethoven	Fifth Symphony (I)	00:00:00.341	00:00:10.371	00:00:10.029	442,288
Tchaikovsky	The Nutcracker (Dance of the reeds)	00:01:05.777	00:01:09.069	00:00:03.291	145,176
Grieg	Piano Concerto (I)	00:00:02.655	00:00:06.998	00:00:04.342	191,484
Pachelbel	Canon	00:01:47.640	00:01:52.911	00:00:05.271	232,470
Louis Armstrong	What a wonderful world	00:00:06.132	00:00:09.941	00:00:03.809	167,994
Ray Charles	Georgia on my mind	00:00:30.190	00:00:35.787	00:00:05.597	246,862
Frank Sinatra	Night and Day	00:00:10.132	00:00:15.150	00:00:05.018	221,298
Glenn Miller	In the mood	00:00:11.089	00:00:17.518	00:00:06.428	283,502
Morris Albert	Feelings	00:00:08.724	00:00:13.331	00:00:04.606	203,136
The Platters	Only you	00:00:04.842	00:00:10.121	00:00:05.278	232,794
Chuck Berry	Sweet little sixteen	00:00:00.405	00:00:04.724	00:00:04.318	190,456
Billie Holiday	Time on my hands	00:00:11.881	00:00:17.589	00:00:05.707	251,687
Tears for Fears	Sowing the seeds of love	00:01:19.591	00:01:28.306	00:00:08.714	384,311
James Brown	I got you (I feel good)	00:00:00.354	00:00:04.372	00:00:04.017	177,174
Boyz II Men	A song for mama	00:00:25.760	00:00:31.808	00:00:06.047	266,708
Scorpions	Wind of change	00:01:39.660	00:01:45.520	00:00:05.859	258,423

2 Tables

The following table presents the average computation time of each AAC module, considering all PC implementations, versions, and parameters.

Module	PII C 320 kbps	PII C 128 kbps	PII C 32 kbps	PII ASM 320 kbps	PIII C 320 kbps	PIII ASM 320 kbps
Init	0.001314	0.001244	0.001214	0.001083	0.000869	0.001141
Channel	0.000021	0.000023	0.000013	0.000025	0.000007	0.000011
Psychoac	0.215356	0.212774	0.214004	0.087016	0.106398	0.045416
Blk_switch	0.000020	0.000019	0.000003	0.000017	0.000011	0.000009
pre-proc	0.000003	0.000003	0.000016	0.000014	0.000007	0.000013
t/f-map	0.022330	0.022270	0.022336	0.023603	0.012204	0.013450
psych2scfac	0.000099	0.000094	0.000101	0.000093	0.000053	0.000161
specLFE	0.000001	0.000003	0.000006	0.000005	0.000001	0.000002
quant&cod	0.000192	0.000229	0.000292	0.000259	0.000143	0.000122
coupling	0.000000	0.000000	0.000000	0.000000	0.000000	0.000001

TNS	0.002813	0.002747	0.002856	0.002715	0.001222	0.001619
LTP	0.565472	0.561258	0.583572	0.081424	0.252519	0.048695
IntStereo	0.000920	0.000784	0.000851	0.000768	0.000473	0.000509
Predict	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
MSstereo	0.000099	0.000109	0.000096	0.000118	0.000085	0.000069
DynRange	0.000002	0.000000	0.000000	0.000000	0.000000	0.000000
AACquant	0.027807	0.182546	0.653176	0.027432	0.014393	0.014613
MSstereoRec	0.000067	0.000112	0.000059	0.000079	0.000058	0.000084
IntStereoRec	0.000068	0.000069	0.000081	0.000091	0.000025	0.000033
SpecUpdt	0.027668	0.027021	0.027256	0.028840	0.014360	0.015934
Write	0.002151	0.001109	0.000511	0.002078	0.001333	0.001330
TOTAL	0.866403	1.012414	1.506443	0.255660	0.404161	0.143212

The following table explains the purpose of each module:

Module	**Purpose**
Init	Initialization
Channel	Determine channel elements
Psychoac	Psychoacoustics
Blk_switch	Block switching processing
pre-proc	Pre-processing
t/f-map	T/F mapping
Psych2scfac	adapt ratios of psychoacoustic module to codec scale factor bands
SpecLFE	Set upper spectral coefficients to zero for LFE
Quant&cod	Quantization and coding
Coupling	Coupling Coordinate calculation
TNS	Perform TNS analysis and filtering
LTP	LTP predictor info and residual spectrum
IntStereo	Intensity Stereo
Predict	predictor info and residual spectrum
MSstereo	MS stereo
DynRange	Dynamic range parameters
AACquant	AAC quantization and coding module
MSstereoRec	Reconstruct MS Stereo bands for prediction
IntStereoRec	Reconstruct Intensity Stereo bands for prediction
SpecUpdt	Add predicted spectrum, TNS decode, update LTP history buffer
Write	Write out all encoded channels

Appendix 2 Original C source-code

This appendix provides the original C source-code from ISO for specific routines of the AAC encoder.

1 LTP module: lag correlation routine

This section presents the lag correlation routine in the C version:

```
int
pitch(double *sb_samples, short *x_buffer, int flen, int lag0, int lag1, double
*predicted_samples, Float *gain, int *cb_idx)
{
   int i, j, delay, start;
   int offset;
   double corr1, corr2, lag_corr;
   double p_max, energy, lag_energy;


   p_max = 0.0;
   lag_corr = lag_energy = 0.0;
   delay = lag0;

   /* Find the lag. */
  for (i = lag0; i < lag1; i++)
  {
        corr1 = corr2 = 0.0;

        start = 0;
        offset = i;

        if(i < DELAY / 2)
        {
               offset = i - lag0;
              start = flen / 2 + i;
        }
        else
        {
               offset = i - DELAY / 2;
              start = 0;
        }

        if(start || offset == 0)
        {
               energy = 0.0f;
               for (j = start; j < flen; j++)
               {
                      corr1 += x_buffer[NOK_LT_BLEN - offset - j - 1] * sb_samples[flen
- j - 1];
                      energy   +=   x_buffer[NOK_LT_BLEN   -   offset   -   j   -   1]   *
x_buffer[NOK_LT_BLEN - offset - j - 1];
               }
        }
        else    /* start == 0 && offset != 0 */
        {
               /* No need to compute the whole energy. */
               energy   -=   x_buffer[NOK_LT_BLEN   -   offset]   *   x_buffer[NOK_LT_BLEN   -
offset];
               energy += x_buffer[NOK_LT_BLEN - offset - flen] * x_buffer[NOK_LT_BLEN -
offset - flen];

        for (j = 0; j < flen; j++)
               corr1 += x_buffer[NOK_LT_BLEN - offset - j - 1] * sb_samples[flen - j -
1];
        }

        if (energy != 0.0)
               corr2 = corr1 / sqrt (energy);
```

```
        else
               corr2 = 0.0;

        if (p_max < corr2)
        {
               p_max = corr2;
              delay = i;
              lag_corr = corr1;
              lag_energy = energy;
        }
   }

   /* Compute the gain. */
   if(lag_energy != 0.0)
        *gain =  lag_corr / (1.010 * lag_energy);
   else
        *gain = 0.0;

   /* ..... */

}
```

2 Psychoacoustical module

This section presents some extracts of the C version of the Psychoacoustical module:

```
/* ..... */

double sprdngf(double b1, double b2)
{
   double tmpx,tmpy,tmpz;

   tmpx = (b2 >= b1) ? 3.0*(b2-b1) : 1.5*(b2-b1);
   tmpz = 8.0 * psy_min( (tmpx-0.5)*(tmpx-0.5) - 2.0*(tmpx-0.5),0.0 );
   tmpy = 15.811389 + 7.5*(tmpx + 0.474)-17.5 *sqrt(1.0 + (tmpx+0.474)*(tmpx+0.474));

   return( tmpy < -100.0 ? 0.0 : pow(10.0, (tmpz + tmpy)/10.0) );
}


/* ..... */


void psy_step2(double sample[][BLOCK_LEN_LONG*2],
       PSY_STATVARIABLE_LONG *psy_stvar_long,
       PSY_STATVARIABLE_SHORT *psy_stvar_short,
       FFT_TABLE_LONG *fft_tbl_long,
       FFT_TABLE_SHORT *fft_tbl_short,
       int ch
       )
{
   int w,i,j,k,l,h,n,d,ik,k2,n4;
   double t,s,c,dx,dy;
   double *xl,*yl;

   /* FFT for long */
   xl = (double *)malloc( sizeof(double) * BLOCK_LEN_LONG * 2 );
   yl = (double *)malloc( sizeof(double) * BLOCK_LEN_LONG * 2 );

   psy_stvar_long->p_fft[ch] += BLOCK_LEN_LONG;

   if(psy_stvar_long->p_fft[ch] == BLOCK_LEN_LONG * 3)
        psy_stvar_long->p_fft[ch] = 0;

   /* window *//* static int it = 0; */
   for(i = 0; i < BLOCK_LEN_LONG*2; ++i){
        xl[i] = fft_tbl_long->hw[i] * sample[ch][i];
        yl[i] = 0.0;
   }

   n = BLOCK_LEN_LONG*2;
   n4 = n/4;
```

```
   for (i = 0; i < n; ++i) {    /* bit inverse */
        j = fft_tbl_long->brt[i];
        if (i < j) {
               t = xl[i];  xl[i] = xl[j];  xl[j] = t;
               t = yl[i];  yl[i] = yl[j];  yl[j] = t;
        }
   }
   for (k = 1; k < n; k = k2) {    /* translation */
        h = 0;  k2 = k + k;  d = n / k2;
        for (j = 0; j < k; j++) {
               c = fft_tbl_long->st[h + n4];
               s = fft_tbl_long->st[h];
               for (i = j; i < n; i += k2) {
                      ik = i + k;
                      dx = s * yl[ik] + c * xl[ik];
                      dy = c * yl[ik] - s * xl[ik];
                      xl[ik] = xl[i] - dx;  xl[i] += dx;
                      yl[ik] = yl[i] - dy;  yl[i] += dy;
               }
               h += d;
        }
   }

   for(w = 0; w < BLOCK_LEN_LONG; ++w){
        psy_stvar_long->fft_r[ch][w+psy_stvar_long->p_fft[ch]]
                  = sqrt(xl[w]*xl[w] + yl[w]*yl[w]);
        if( xl[w] > 0.0 ){
               if( yl[w] >= 0.0 )
                      psy_stvar_long->fft_f[ch][w+psy_stvar_long->p_fft[ch]]   =   atan(
yl[w] / xl[w] );
               else
                      psy_stvar_long->fft_f[ch][w+psy_stvar_long->p_fft[ch]]   =   atan(
yl[w] / xl[w] )+ M_PI * 2.0;
        } else if( xl[w] < 0.0 ) {
               psy_stvar_long->fft_f[ch][w+psy_stvar_long->p_fft[ch]]  =  atan(  yl[w]  /
xl[w] ) + M_PI;
        } else {
               if( yl[w] > 0.0 )
                      psy_stvar_long->fft_f[ch][w+psy_stvar_long->p_fft[ch]]  =  M_PI  *
0.5;
               else if( yl[w] < 0.0 )
                      psy_stvar_long->fft_f[ch][w+psy_stvar_long->p_fft[ch]]  =  M_PI  *
1.5;
               else
                      psy_stvar_long->fft_f[ch][w+psy_stvar_long->p_fft[ch]]  =  0.0;  /*
tmp */
        }
   }
   free(xl);
   free(yl);


   /* added by T. Araki (1997.10.16) */
   /* FFT for short */
   xl = (double *)malloc( sizeof(double) * BLOCK_LEN_SHORT * 2 );
   yl = (double *)malloc( sizeof(double) * BLOCK_LEN_SHORT * 2 );

   for(l = 0; l < MAX_SHORT_WINDOWS; ++l){
        /* window */
        for(i = 0; i < BLOCK_LEN_SHORT*2; ++i){
               xl[i]    =    fft_tbl_short->hw[i]    *    sample[ch][OFFSET_FOR_SHORT    +
BLOCK_LEN_SHORT * l + i];
               yl[i] = 0.0;
        }

        n = BLOCK_LEN_SHORT*2;
        n4 = n/4;

        for (i = 0; i < n; ++i) {    /* bit inverse */
               j = fft_tbl_short->brt[i];
               if (i < j) {
                      t = xl[i];  xl[i] = xl[j];  xl[j] = t;
                      t = yl[i];  yl[i] = yl[j];  yl[j] = t;
               }
        }
        for (k = 1; k < n; k = k2) {    /* translation */
```

```
                h = 0;  k2 = k + k;  d = n / k2;
                for (j = 0; j < k; j++) {
                        c = fft_tbl_short->st[h + n4];
                        s = fft_tbl_short->st[h];
                        for (i = j; i < n; i += k2) {
                                ik = i + k;
                                dx = s * yl[ik] + c * xl[ik];
                                dy = c * yl[ik] - s * xl[ik];
                                xl[ik] = xl[i] - dx;  xl[i] += dx;
                                yl[ik] = yl[i] - dy;  yl[i] += dy;
                        }
                        h += d;
                }
        }

        for(w = 0; w < BLOCK_LEN_SHORT; w++){
                psy_stvar_short->fft_r[l][w] = sqrt(xl[w]*xl[w] + yl[w]*yl[w]);

                if( xl[w] > 0.0 ){
                        if( yl[w] >= 0.0 )
                                psy_stvar_short->fft_f[l][w] = atan( yl[w] / xl[w] );
                        else
                                psy_stvar_short->fft_f[l][w] = atan( yl[w] / xl[w] )+ M_PI
* 2.0;
                } else if( xl[w] < 0.0 ) {
                        psy_stvar_short->fft_f[l][w] = atan( yl[w] / xl[w] ) + M_PI;
                } else {
                        if( yl[w] > 0.0 )
                                psy_stvar_short->fft_f[l][w] = M_PI * 0.5;
                        else if( yl[w] < 0.0 )
                                psy_stvar_short->fft_f[l][w] = M_PI * 1.5;
                        else
                                psy_stvar_short->fft_f[l][w] = 0.0; /* tmp */
                }
        }
    }
    free(xl);
    free(yl);
/* added by T. Araki (1997.10.16) end */
}


/* ..... */


void psy_step4(PSY_STATVARIABLE_LONG *psy_stvar_long,
        PSY_STATVARIABLE_SHORT *psy_stvar_short,
    PSY_VARIABLE_LONG *psy_var_long,
    PSY_VARIABLE_SHORT *psy_var_short,
    int ch
    )
{
    int w,i;
    double r,f,rp,fp;

    for(w = 0; w < BLOCK_LEN_LONG; ++w){
         r = psy_stvar_long->fft_r[ch][psy_stvar_long->p_fft[ch]+w];
         f = psy_stvar_long->fft_f[ch][psy_stvar_long->p_fft[ch]+w];
         rp = psy_var_long->r_pred[w];
         fp = psy_var_long->f_pred[w];

         if( r + fabs(rp) != 0.0 )
                psy_var_long->c[w] = sqrt( psy_sqr(r*cos(f) - rp*cos(fp))
                      +psy_sqr(r*sin(f) - rp*sin(fp)) )/ ( r + fabs(rp) ) ;
         else
                psy_var_long->c[w] = 0.0; /* tmp */
         }

         /* added by T. Araki (1997.10.16) */
         for(i = 0; i < MAX_SHORT_WINDOWS; ++i){
                for(w = 0; w < BLOCK_LEN_SHORT; ++w){
                        r = psy_stvar_short->fft_r[i][w];
                        f = psy_stvar_short->fft_f[i][w];
                        rp = psy_var_short->r_pred[i][w];
                        fp = psy_var_short->f_pred[i][w];

                        if( r + fabs(rp) != 0.0 )
```

```
                                psy_var_short->c[i][w]    =    sqrt(    psy_sqr(r*cos(f)    -
rp*cos(fp))+ psy_sqr(r*sin(f) - rp*sin(fp)) )/ ( r + fabs(rp) ) ;
                        else
                                psy_var_short->c[i][w] = 0.0; /* tmp */
                }
        }
        /* added by T. Araki (1997.10.16) end */
}


/* ..... */


void psy_step6(PARTITION_TABLE_LONG *part_tbl_long,
        PARTITION_TABLE_SHORT *part_tbl_short,
        PSY_VARIABLE_LONG *psy_var_long,
        PSY_VARIABLE_SHORT *psy_var_short
        )
{
   int b,bb,i;
   double ecb,ct;
   double sprd;

   for(b = 0; b < part_tbl_long->len; ++b){
        ecb = 0.0;
        ct = 0.0;
        for(bb = 0; bb < part_tbl_long->len; ++bb){
                sprd = sprdngf(part_tbl_long->bval[bb],part_tbl_long->bval[b]);
                ecb += psy_var_long->e[bb] * sprd;
                ct += psy_var_long->c[bb] * sprd;
        }
        if (ecb!=0.0) {
                psy_var_long->cb[b] = ct / ecb;
        } else {
                psy_var_long->cb[b] = 0.0;
        }
        psy_var_long->en[b] = ecb * part_tbl_long->rnorm[b];
   }

   /* added by T. Araki (1997.10.16) */
   for(i = 0; i < MAX_SHORT_WINDOWS; ++i){
        for(b = 0; b < part_tbl_short->len; ++b){
                ecb = 0.0;
                ct = 0.0;
                for(bb = 0; bb < part_tbl_short->len; ++bb){
                        sprd = sprdngf(part_tbl_short->bval[bb],part_tbl_short->bval[b]);
                        ecb += psy_var_short->e[i][bb] * sprd;
                        ct += psy_var_short->c[i][bb] * sprd;
                }
                if (ecb!=0.0) {
                        psy_var_short->cb[i][b] = ct / ecb;
                } else {
                        psy_var_short->cb[i][b] = 0.0;
                }
                psy_var_short->en[i][b] = ecb * part_tbl_short->rnorm[b];
        }
   }
   /* added by T. Araki (1997.10.16) end */
}
```

Appendix 3 Assembly code

1 LTP module: lag correlation routine

```
.686
.MMX
.MODEL flat, C
.DATA
i1                          dword   0
start                       dword   0

corr1                       qword   0
energy                      qword   0
corr2                       real8   0.0
p_max                       real8   0.0

lag_corr                    real8   0.0
lag_energy                  real8   0.0

gain_const                  real8   1.010

var32                       dword   0
var_real                    real8   0.0


.CODE

DELAY_2                             =       1024
NOK_LT_BLEN                         =       3072
LOOP_LOST_PRECISION                 =       8
LOOP_SHIFT                          =       LOOP_LOST_PRECISION + 32

CorrLag PROC uses eax ebx ecx edx esi edi,
       sb_samples:          ptr,    \
       x_buffer:            ptr,    \
       flen:                dword,  \
       lag0:                dword,  \
       lag1:                dword,  \
       delay:               ptr,    \
       gain:                ptr

       mov          eax, delay
       mov          ecx, gain
       mov          dword ptr [eax], 0    ; delay = 0
       fldz
       fstp         qword ptr [ecx]       ; *gain = 0

       mov          eax, lag0
       mov          i1, eax               ; i1 = lag0

   outer_loop:

       ; reg_alloc : eax = i1

       cmp          eax, lag1
       jge          end_corr_lag

   cmp_delay:

       pxor         mm7, mm7              ; corr1 = 0

       mov          ecx, flen
       cmp          eax, DELAY_2          ; if(i < DELAY /2)
       jge          cmp_delay_else

   ; reg_alloc : eax = i1, ebx = offset, ecx = flen, edx = start

       mov          ebx, eax
       sub          ebx, lag0             ; offset = i - lag0;

       mov          edx, ecx              ; start = flen / 2 + i;
```

```
        sar             edx, 1
        jns             short @cmp_delay1
        adc             edx, 0
@cmp_delay1:
        add             edx, eax
        jmp             short cmp_start_offset

cmp_delay_else:

    ; reg_alloc : eax = i1, ebx = offset, ecx = flen, edx = start

        mov             ebx, eax
        add             ebx, -DELAY_2           ; offset = i - DELAY / 2;
        xor             edx, edx                ; start = 0;

cmp_start_offset:

    ; reg_alloc : eax = i1, ebx = offset, ecx = flen, edx = start, esi = j, mm7 = energy

        mov             eax, i1
        test            edx, edx                ; if(start || offset == 0)
        jne             short cmp_start_offset_then
        test            ebx, ebx
        jne             cmp_start_offset_else

cmp_start_offset_then:

        pxor            mm6, mm6                ; energy = 0;

    ; reg_alloc : ebx = offset=>tmp2, ecx = flen, edx = start=>tmp1, esi = j, mm6 =
energy, mm7 = corr1

    ; tmp1 = &x_buffer         + NOK_LT_BLEN - offset - 1 - j
    ; tmp2 = &sb_samples       + flen - 1 - j

        mov             esi, edx                ; j = start
        mov             edx, x_buffer           ; tmp1 = &x_buffer
        mov             eax, esi
        sal             eax, 1
        sub             edx, eax                ; tmp1 = &x_buffer - j
        mov             eax, ebx
        sal             eax, 1
        sub             edx, eax                ; tmp1 = &x_buffer - j - offset
        mov             eax, NOK_LT_BLEN
        sal             eax, 1
        add             edx, eax        ; tmp1 = &x_buffer - j - offset + NOK_LT_BLEN
        mov             ebx, sb_samples         ; tmp2 = &sb_samples
        mov             eax, ecx
        sal             eax, 1
        add             ebx, eax                ; tmp2 = &sb_samples + flen
        mov             eax, esi
        sal             eax, 1
        sub             ebx, eax        ; tmp2 = &sb_samples + flen - j
        sub             edx, 8          ; tmp1 = &x_buffer - offset + NOK_LT_BLEN -j - 1
        sub             ebx, 8          ; tmp2 = &sb_samples + flen -j - 1

loop_corr_1:                            ; for (j = start; j < flen; j++)

        cmp             esi, ecx
        jge             loop_corr_1_end


        ; corr1 += x_buffer[NOK_LT_BLEN - offset - j - 1]
        ;                               * sb_samples[flen - j - 1];

        movq            mm0, qword ptr [edx]  ; mm0 <= x_buffer
        movq            mm1, qword ptr [ebx]  ; mm1 <= sb_samples

        pmaddwd         mm0, mm1
        psrad           mm0, LOOP_LOST_PRECISION
        paddd           mm7, mm0


        ; energy += x_buffer[NOK_LT_BLEN - offset - j - 1]
        ;                               * x_buffer[NOK_LT_BLEN - offset - j - 1];

        movq            mm0, qword ptr [edx]  ; mm0 <= x_buffer
```

```
        movq            mm1, mm0                ; mm1 <= x_buffer
        pmaddwd         mm0, mm1
        psrad           mm0, LOOP_LOST_PRECISION
        paddd           mm6, mm0

        add             esi, 4
        sub             ebx, 8
        sub             edx, 8
        jmp             loop_corr_1

loop_corr_1_end:

        movq            mm0, mm7
        psrlq           mm0, 32
        paddd           mm7, mm0
        movq            mm0, mm6
        psrlq           mm0, 32
        paddd           mm6, mm0

        jmp             energy_check

cmp_start_offset_else:


        ;       else /* start == 0 && offset != 0 */
        ;       {

        ; reg_alloc : ebx = offset=>tmp2, ecx = flen, edx = start=>tmp1, esi = j, mm6 =
energy, mm7 = corr1

      ; tmp1 = &x_buffer             + NOK_LT_BLEN - offset (- j)
      ; tmp2 = &sb_samples    + flen - 1 (- j)

        mov             esi, 0                  ; j = 0
        mov             edx, x_buffer           ; tmp1 = &x_buffer
        mov             eax, ebx
        sal             eax, 1
        sub             edx, eax                ; tmp1 = &x_buffer - offset
        mov             eax, NOK_LT_BLEN
        sal             eax, 1
        add             edx, eax                ; tmp1 = &x_buffer - offset + NOK_LT_BLEN
        mov             ebx, sb_samples         ; tmp2 = &sb_samples
        mov             eax, ecx
        sal             eax, 1
        add             ebx, eax                ; tmp2 = &sb_samples + flen
        sub             ebx, 8                  ; tmp2 = &sb_samples + flen - 1

                        ; energy -= x_buffer[NOK_LT_BLEN - offset]
                        ;               * x_buffer[NOK_LT_BLEN - offset];

        pxor            mm2, mm2
        movq            mm6, energy
        psrlq           mm6, LOOP_LOST_PRECISION

        movq            mm0, qword ptr [edx]    ; mm0 <= x_buffer
        punpcklwd       mm0, mm2
        movq            mm1, mm0                ; mm1 <= x_buffer
        pmaddwd         mm0, mm1
        psrad           mm0, LOOP_LOST_PRECISION
        psubd           mm6, mm0

               ; energy += x_buffer[NOK_LT_BLEN - offset - flen]
               ;                * x_buffer[NOK_LT_BLEN - offset - flen];
        mov             eax, ecx
        sal             eax, 1
        sub             edx, eax        ; tmp1 = &x_buffer - offset + NOK_LT_BLEN - flen
        movq            mm0, qword ptr [edx]    ; mm0 <= x_buffer
        punpcklwd       mm0, mm2
        movq            mm1, mm0        ; mm1 <= x_buffer
        pmaddwd         mm0, mm1
        psrad           mm0, LOOP_LOST_PRECISION
        paddd           mm6, mm0
        add             edx, eax        ; tmp1 = &x_buffer - offset + NOK_LT_BLEN
        sub             edx, 8          ; tmp1 = &x_buffer - offset + NOK_LT_BLEN -j - 1

loop_corr_2:
        ; for (j = 0; j < flen; j++)
```

```
        cmp             esi, ecx
        jge             loop_corr_2_end


    ; corr1 += x_buffer[NOK_LT_BLEN - offset - j - 1]
    ;                         * sb_samples[flen - j - 1];

        movq            mm0, qword ptr [edx]  ; mm0 <= x_buffer
        movq            mm1, qword ptr [ebx]  ; mm1 <= sb_samples
        pmaddwd         mm0, mm1
        psrad           mm0, LOOP_LOST_PRECISION
        paddd           mm7, mm0
        add             esi, 4
        sub             ebx, 8
        sub             edx, 8
        jmp             loop_corr_2

loop_corr_2_end:

        movq            mm0, mm7
        psrlq           mm0, 32
        paddd           mm7, mm0

energy_check:

        pxor            mm5, mm5
        punpckldq       mm7, mm5
        punpckldq       mm6, mm5

        movq            mm5, mm7
        psrad           mm5, 32
        punpckldq       mm7, mm5
        psllq           mm7, LOOP_LOST_PRECISION
        movq            corr1, mm7
        psllq           mm6, LOOP_LOST_PRECISION
        movq            energy, mm6
        emms

        fild            energy          ; if (energy != 0.0)
        fldz
        fcomip          st, st(1)
        je              energy_zero

energy_calc:


                        ; corr2 = corr1 / sqrt (energy);

                                        ; st(0) = energy
        fsqrt                           ; st(0) = sqrt(energy)
        fild            corr1           ; st(0) = corr1, st(1) = sqrt(energy)

        fdivrp          st(1), st(0)    ; st(0) = corr2
        fst             corr2

        fld             p_max           ; if (p_max < corr2)
        fcomip          st, st(1)
        jae             outer_loop_end

                        ; reg_alloc : ebx = &delay, ecx = &gain
                        ; st(0) = corr2
        fstp            p_max           ; p_max = corr2;

        mov             ebx, delay
        mov             ecx, gain
        mov             eax, i1         ; delay = i;
        mov             dword ptr [ebx], eax

        fild            corr1           ; lag_corr = corr1;
        fst             lag_corr
        fild            energy          ; st(0) = energy, st(1) = lag_corr
        fst             lag_energy      ; lag_energy = energy;
        fmul            gain_const      ; *gain = lag_corr / (((short)1.010) * lag_energy);
        fdivp           st(1), st(0)    ; st(0) = gain
        fstp            real8 ptr [ecx]
```

```
        jmp             outer_loop_end

energy_zero:

        mov             ebx, delay
        mov             ecx, gain

outer_loop_end:

        mov             eax, i1
        inc             eax
        mov             i1, eax
        jmp             outer_loop

end_corr_lag:

        mov             ebx, delay
        mov             ecx, gain
        mov             eax, dword ptr [ebx]  ; delay

        fld             qword ptr [ecx]       ; gain
        fstp            var_real


        ret

CorrLag ENDP

_DATA segment
scr_msg1@       label   byte
   ;    s@+0:
   db   "value = %d",10,0
   align        4
scr_msg2@       label   byte
   ;    s@+0:
   db   "energy = %08lx",10,0
   align        4
scr_msg3@       label   byte
   ;    s@+0:
   db   "value = %e",10,0
   align        4
scr_msg4@       label   byte
   ;    s@+0:
   db   "HERE!!!",10,0
   align        4
_DATA   ends

extrn           printf:near
extrn           _turboFloat:word

END
```

Appendix 4 VHDL implementation

The following section presents the third VHDL implementation.

1 Third implementation

ltp_lag.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity ltp_lag is
   generic (data_width : integer:= 16);
   port(
               reset:  in std_logic;
               clock:  in std_logic;
               init:   in std_logic;  -- Begin mathematical processing
               busy:   out std_logic; -- Signaling
               input_ready:   in std_logic;
               i0, i1:        in std_logic_vector(data_width-1 downto 0);
               corr_out, enrg_out: out std_logic_vector(2*data_width-1 downto 0);
               addr0, addr1:  out std_logic_vector(11 downto 0)
   );
end entity;

architecture rtl of ltp_lag is
   constant ADDRESS_WIDTH:           integer := 13;

   type   state_type   is   (st_reset,   st_ini,   st_run_ini,   st_run_ini_wait_input,
st_run_first_0,   st_run_first_1,   st_run_first_2,   st_run_first_3,   st_run_second0_0,
st_run_second0_1,       st_run_second0_2,       st_run_second1_0,       st_run_second1_1,
st_run_second1_2,       st_run_second2_0,       st_run_second2_1,       st_run_second2_2,
st_run_second2_3,  st_run_second2_4, st_end);

   signal state:              state_type;
   signal reg0, reg1:         std_logic_vector(data_width-1 downto 0);
   signal corr_int, enrg_int: std_logic_vector(2*data_width-1 downto 0);
   signal corr, enrg:         std_logic_vector(2*data_width-1 downto 0);

   signal enable_corr_mul, enable_enrg_mul: std_logic;
   signal enable_corr_add, enable_enrg_add: std_logic;

   signal NOK_LT_BLEN, LAG1, DELAY_2, FLEN, FLEN_2: std_logic_vector(ADDRESS_WIDTH-1
downto 0);
begin
   fsm: process(clock)
       variable start, offset, j, lag:      std_logic_vector(ADDRESS_WIDTH-1 downto 0);
   begin
       if reset = '1' then
              state         <= st_reset;
       elsif(clock'event and clock = '1') then
              case state is
                     when st_reset =>      -- Initial state
                            if init = '1' then
                                   state <= st_ini;
                            end if;

                            reg0    <= (others => '0');
                            reg1    <= (others => '0');


                            start   := (others => '0');
                            offset  := (others => '0');
                            j       := (others => '0');
                            lag     := (others => '0');

                     when st_ini =>        -- Waits for init command
```

```
                        if lag >= LAG1 then
                                state <= st_end;
                        else
                                state  <= st_run_ini;

                                if lag < DELAY_2 then
                                        offset  := lag;
                                        start   := FLEN_2 + lag;
                                else
                                        offset  := lag - DELAY_2;
                                        start   := (others => '0');
                                end if;
                        end if;
                when st_run_ini =>
                        reg0    <= i0;
                        reg1    <= i1;

                        j       := start;
                        if lag <= DELAY_2 then
                                -- start || offset == 0
                                addr0   <= NOK_LT_BLEN - offset - j - 1;
                                addr1   <= flen - j - 1;

                                state   <= st_run_ini_wait_input;
                        else
                                addr0   <= NOK_LT_BLEN - offset;
                                addr1   <= (others => '0');

                                if input_ready = '1' then
                                        state   <= st_run_second0_0;
                                end if;
                        end if;
                when st_run_ini_wait_input =>
                        if input_ready = '1' then
                                state   <= st_run_first_0;
                        end if;
                when st_run_first_0 =>
-- corr1 += x_buffer[NOK_LT_BLEN - offset - j - 1] * sb_samples[flen - j - 1];
-- energy += x_buffer[NOK_LT_BLEN - offset - j - 1] * x_buffer[NOK_LT_BLEN - offset - j
- 1];
                        state   <= st_run_first_1;
                when st_run_first_1 =>
                        state   <= st_run_first_2;
                when st_run_first_2 =>
                        j := j + 1;
                        if j < FLEN then
                                addr0   <= NOK_LT_BLEN - offset - j - 1;
                                addr1   <= flen - j - 1;
                                state   <= st_run_first_3;
                        else
                                lag := lag + 1;
                                state   <= st_ini;
                        end if;
                when st_run_first_3 =>
                        if input_ready = '1' then
                                state   <= st_run_first_0;
                        end if;
                when st_run_second0_0 =>
-- energy -= x_buffer[NOK_LT_BLEN - offset] * x_buffer[NOK_LT_BLEN - offset];
                        state   <= st_run_second0_1;
                when st_run_second0_1=>
                        state   <= st_run_second0_2;
                when st_run_second0_2=>
                        reg0    <= i0;
                        reg1    <= i1;

                        addr0   <= NOK_LT_BLEN - offset - flen;
                        addr1   <= (others => '0');
                        if input_ready = '1' then
                                state   <= st_run_second1_0;
                        end if;
                when st_run_second1_0=>
-- energy += x_buffer[NOK_LT_BLEN - offset - flen] * x_buffer[NOK_LT_BLEN - offset -
flen];
                        state   <= st_run_second1_1;

                when st_run_second1_1=>
```

```
                        state   <= st_run_second1_2;
                    when st_run_second1_2=>
                        reg0    <= i0;
                        reg1    <= i1;

                        j := start;
                        addr0   <= NOK_LT_BLEN - offset - j - 1;
                        addr1   <= flen - j - 1;

                        if input_ready = '1' then
                            state   <= st_run_second2_0;
                        end if;
                    when st_run_second2_0=>
-- corr1 += x_buffer[NOK_LT_BLEN - offset - j - 1] * sb_samples[flen - j - 1];
                        state   <= st_run_second2_1;
                    when st_run_second2_1 =>
                        state   <= st_run_second2_2;
                    when st_run_second2_2 =>
                        reg0    <= i0;
                        reg1    <= i1;
                        state   <= st_run_second2_3;
                    when st_run_second2_3 =>
                        j := j + 1;
                        if j < FLEN then
                            addr0   <= NOK_LT_BLEN - offset - j - 1;
                            addr1   <= flen - j - 1;
                            state   <= st_run_second2_4;
                        else
                            lag     := lag + 1;
                            state   <= st_ini;
                        end if;
                    when st_run_second2_4 =>
                        if input_ready = '1' then
                            state   <= st_run_second2_0;
                        end if;
                    when st_end =>
                        if init = '1' then
                            state <= st_reset;
                        end if;
            end case;
        end if;
    end process fsm;

    NOK_LT_BLEN <= conv_std_logic_vector(3072, ADDRESS_WIDTH);
    LAG1        <= conv_std_logic_vector(2048, ADDRESS_WIDTH);
    DELAY_2     <= conv_std_logic_vector(1024, ADDRESS_WIDTH);
    FLEN        <= conv_std_logic_vector(2048, ADDRESS_WIDTH);
    FLEN_2      <= conv_std_logic_vector(1024, ADDRESS_WIDTH);

    corr_out    <= corr;
    enrg_out    <= enrg;

    operators: process(clock)
    begin
        if(clock'event and clock = '1') then
            if enable_corr_mul = '1' then
                corr_int <= reg0 * reg1;
            end if;
            if enable_enrg_mul = '1' then
                enrg_int <=    reg0 * reg0;
            end if;
            if enable_corr_add = '1' then
                corr <= corr + corr_int;
            end if;
            if enable_enrg_add = '1' then
                enrg <= enrg + enrg_int;
            end if;
        end if;
    end process;


    with state select
        enable_corr_mul <=    '1'           when   st_run_first_0,
                              '1'           when   st_run_second2_0,
                              '0'           when   others;

    with state select
```

```
        enable_enrg_mul <=    '1'           when   st_run_first_0,
                              '1'           when   st_run_second0_0,
                              '1'           when   st_run_second1_0,
                              '0'           when   others;

    with state select
        enable_corr_add <=    '1'           when   st_run_first_1,
                              '1'           when   st_run_second2_1,
                              '0'           when   others;

    with state select
        enable_enrg_add <=    '1'           when   st_run_first_1,
                              '1'           when   st_run_second0_1,
                              '1'           when   st_run_second1_1,
                              '0'           when   others;

    with state select
        busy <=               '0'           when   st_reset,
                              '0'           when   st_end,
                              '1'           when   others;

end rtl;
```

Appendix 5 MATLAB files

This chapter presents some of the MATLAB files used in the study of the MPEG-4 AAC standard.

1 Windowing

These files may be used to generate windows and window sequences.

winlong.m

```
function x = winlong(winshape, winshape_prev)

win_length = 2048;
x = zeros(1, win_length);

if winshape_prev == 0
   x(1:1024) = sin(pi/win_length * (0:1023 + 0.5));
end
if winshape_prev == 1
   a = kaiser(2048, 4);
   x(1:1024) = a(1:1024);
end

if winshape == 0
   x(1025:2048) = sin(pi/win_length * (1024:2047 + 0.5));
end
if winshape == 1
   x(1025:2048) = a(1025:2048);
end
```

winshrt.m

```
function x = winshrt(winshape, winshape_prev)

win_length = 256;
x = zeros(1, win_length);

w1 = zeros(1, 128);
w2 = zeros(1, 256);

if winshape_prev == 0
   w1(1:128) = sin(pi/win_length * (0:127 + 0.5));
end
if winshape_prev == 1
   a = kaiser(256, 6);
   w1(1:128) = a(1:128);
end

if winshape == 0
   w2(1:128) = sin(pi/win_length * (0:127 + 0.5));
   w2(129:256) = sin(pi/win_length * (128:255 + 0.5));
end
if winshape == 1
   a = kaiser(256, 6);
   w2(1:128) = a(1:128);
   w2(129:256) = a(129:256);
end

x(1:128) = w1;
x(129:256) = w2(129:256);
```

winstart.m

```
function x = winstart(winshape, winshape_prev)

win_length = 2048;
```

```
x = zeros(1, win_length);

if winshape_prev == 0
   x(1:1024) = sin(pi/win_length * (0:1023 + 0.5));
end
if winshape_prev == 1
   a = kaiser(2048, 4);
   x(1:1024) = a(1:1024);
end
x(1025:1472) = 1.0;
x(1601:2048) = 0.0;

if winshape == 0
   x(1473:1600) = sin(pi/256 * (128:255 + 0.5));
end
if winshape == 1
   a = kaiser(256, 6);
   x(1473:1600) = a(129:256);
end
```

winstop.m

```
function x = winstop(winshape, winshape_prev)

win_length = 2048;
x = zeros(1, win_length);

x(1:448) = 0.0;
if winshape_prev == 0
   x(449:576) = sin(pi/256 * (0:127 + 0.5));
end
if winshape_prev == 1
   a = kaiser(256, 6);
   x(449:576) = a(1:128);
end
x(577:1024) = 1.0;

if winshape == 0
   x(1025:2048) = sin(pi/win_length * (1024:2047 + 0.5));
end
if winshape == 1
   a = kaiser(2048, 4);
   x(1025:2048) = a(1025:2048);
end
```

wingraph.m

```
function win_graph

clear all; close all;

winshape = 0;
win_length = 2048;
x = zeros(1, 7 * win_length);

hold on;


x   = winlong(winshape, winshape);   subplot(2, 2, 1); plot(x); axis([0 2100 0 1]);
title('long window');
x   = winshrt(winshape, winshape);   subplot(2, 2, 2); plot(x); axis([0 2100 0 1]);
title('short window');
x   = winstart(winshape, winshape); subplot(2, 2, 3); plot(x); axis([0 2100 0 1]);
title('start window');
x   = winstop(winshape, winshape);   subplot(2, 2, 4); plot(x); axis([0 2100 0 1]);
title('stop window');
```

my_seq.m

```
function my_seq

clear all; close all;

winshape = 0;
win_length = 2048;
x = zeros(1, 7 * win_length);
```

```
hold on;

x  = winlong(winshape, winshape); plot(x);
n= 1; x = [zeros(1, n*win_length*0.5) winlong(winshape, winshape)]; plot(x);
n= 2; x = [zeros(1, n*win_length*0.5) winstart(winshape, winshape)]; plot(x);
n= 3; x = [zeros(1, n*win_length*0.5) winshrts(winshape, winshape)]; plot(x);
n= 3 + 0/8; x = [zeros(1, n*win_length*0.5) winshrt(winshape, winshape)]; plot(x);
n= 3 + 1/8; x = [zeros(1, n*win_length*0.5) winshrt(winshape, winshape)]; plot(x);
n= 3 + 2/8; x = [zeros(1, n*win_length*0.5) winshrt(winshape, winshape)]; plot(x);
n= 3 + 3/8; x = [zeros(1, n*win_length*0.5) winshrt(winshape, winshape)]; plot(x);
n= 3 + 4/8; x = [zeros(1, n*win_length*0.5) winshrt(winshape, winshape)]; plot(x);
n= 3 + 5/8; x = [zeros(1, n*win_length*0.5) winshrt(winshape, winshape)]; plot(x);
n= 3 + 6/8; x = [zeros(1, n*win_length*0.5) winshrt(winshape, winshape)]; plot(x);
n= 3 + 7/8; x = [zeros(1, n*win_length*0.5) winshrt(winshape, winshape)]; plot(x);
n= 4; x = [zeros(1, n*win_length*0.5) winstop(winshape, winshape)]; plot(x);
n= 5; x = [zeros(1, n*win_length*0.5) winlong(winshape, winshape)]; plot(x);
```

2 Psychoacoustic model

crit_band.m

```
function crit_band

close all;
clear all;

F = 0:0.5:20;

B_M = 25 + 75.*(1+1.4.*F.^2).^0.69;      % Munich critical bandwidth
B_C = 24.7.*(1 + 4.37.*F);               % Cambridge critical bandwidth
B_1_3 = 232.*F;

figure; hold on;
plot(F, B_M, 'green :');
plot(F, B_C, 'blue -.');
plot(F, B_1_3, 'red -');

xlabel('frequency (kHz)');
ylabel('critical bandwidth');
legend('Munich', 'Cambridge', '1/3-octave');
```

spreadf.m

```
function spreadf

close all;
clear all;

DEF = 0.1;

b1 = 1:DEF:20;
b2 = 1:DEF:20;

spr = [];
for k1 = 1:length(b1)
   tmp = [];
   for k2 = 1:length(b2)
      tmp = [tmp spreading_function(b1(k1), b2(k2))];
   end;
   spr = [spr [tmp]'];
end

figure;
mesh(b1, b2, spr); view(-37.5, 80);
title('spreading function'); xlabel('frequency (bark)'); ylabel('frequency (bark)');
zlabel('spreading (dB/bark)');

figure;
plot(b1, spreading_function(5,b1)); axis([0 9 0 1]);
title('spreading function at 5 bark'); xlabel('frequency (bark)'); ylabel('normalized
spreading');


function spr = spreading_function(b1, b2)
```

```
if b2 >= b1
   tmpx = 3.0 .* (b2-b1);
else
   tmpx = 1.5 .* (b2-b1);
end
tmpz = 8.0 .* min((tmpx-0.5).^2 - 2.0 .* (tmpx-0.5), 0.0);
tmpy = 15.811389 + 7.5 .* (tmpx + 0.474) - 17.5 .* sqrt(1.0 + (tmpx + 0.474).^2);
if tmpy < - 100.0
   spr = 0.0;
else
   spr = 10.^((tmpz + tmpy)./10.0);
end
```

Appendix 6 Resumo em português

"ESTUDO DO ALGORITMO DE CODIFICAÇÃO DE ÁUDIO DO PADRÃO MPEG-4 AAC (CODIFICADOR DE ÁUDIO AVANÇADO) E COMPARAÇÃO ENTRE IMPLEMENTAÇÕES DE MÓDULOS DO ALGORITMO"

1 Resumo

A codificação de áudio é utilizada para comprimir sinais de áudio, dessa forma reduzindo a quantidade de bits necessária para transmitir ou armazenar sinais de áudio. Isso é útil quando a larga de banda de uma rede ou a capacidade de armazenamento é limitada. Algoritmos de compressão de áudio são baseados em um processo de codificação e decodificação de áudio. No processo de codificação, o sinal de áudio não comprimido é transformado em uma representação codificada, dessa forma comprimindo o sinal de áudio. Após isso, eventualmente necessita-se restabelecer o sinal de áudio codificado (por exemplo, para a reprodução desse sinal), o que ocorre através da decodificação do sinal codificado. O decodificador recebe a cadeia de bits e a reconverte em um sinal não comprimido.

O padrão ISO-MPEG é um padrão para a codificação em alta qualidade e baixas taxas de transmissão para vídeo e áudio. A parte de áudio do padrão é composta de algoritmos para a codificação em alta qualidade e baixas taxas de transmissão de áudio, isto é, algoritmos que reduzem a taxa de transmissão original, ao mesmo tempo que garantem a alta qualidade do sinal de áudio. Os algoritmos de codificação de áudio consistem em MPEG-1 (composto de três camadas diferentes), MPEG-2, MPEG-2 AAC e MPEG-4.

Este trabalho apresenta um estudo do algoritmo de codificação de áudio MPEG-4 AAC (Codificador de Áudio Avançado). Além disso, ele apresenta uma implementação do algoritmo em diferentes plataformas e comparações entre as implementações. As implementações são em linguagem C, em Assembly do Intel Pentium, em linguagem C usando processador DSP e em HDL. Já que cada implementação têm o seu nicho de aplicação, cada uma delas é válida como solução final. Além disso, outro objetivo deste trabalho é a comparação entre essas implementações, considerando-se custos estimados, tempo de execução e vantagens e desvantagens de cada uma delas.

Palavras-chaves: MPEG-4 AAC, Codificação de Áudio, Codificadores Perceptuais, Psicoacústica, PC, MMX, DSP, VHDL.

2 Introdução

Em diversas situações, a largura de banda de rede ou a capacidade de armazenamento disponível é bastante limitada. Por exemplo, em uma rede, diversos usuários podem estar tentando enviar grandes quantidades de dados. Nesse contexto, a redução da quantidade de informação é extremamente necessária. A principal aplicação da codificação de áudio é a redução da quantidade de bits necessário na transmissão ou no armazenamento de um sinal de áudio. Por isso, através de um algoritmo, o codificador de áudio comprime o sinal antes de enviá-lo ou armazená-lo.

Algoritmos de compressão de áudio são baseados em um processo de codificação e decodificação. O processo de codificação é um passo em que o sinal modulado em pulso (PCM) é transformado em uma representação codificada. O objetivo do codificador é comprimir o sinal de áudio. Assim, por causa desse conjunto de parâmetros criado pelo codificador, o sinal de áudio comprimido necessita de menos bits para a representação da informação de áudio. Esse conjunto de parâmetros é uma cadeia de bits que pode ser transmitida em uma rede ou armazenada em um meio de armazenamento.

Eventualmente, necessita-se reestabelecer o sinal de áudio codificado (por exemplo, para a reprodução desse sinal). Para realizar essa tarefa, o sinal de áudio codificado é decodificado. Nesse processo, ele perde o seu formato comprimido e retorna ao formato de áudio original. O decodificador recebe a cadeia de bits e a reconverte em uma representação PCM através de síntese de áudio, baseando-se nos parâmetros obtidos da cadeia de bits.

O processo de compressão e decompressão é desempenhado pelo codificador e pode ser sem ou com perdas. A compressão sem perdas fornece uma reconstrução exata (em termos de bits) do sinal de áudio original no processo de decodificação, isto é, o sinal de áudio original e o sinal decodificado apresentam a mesma sequência de bytes. Por isso, nenhuma informação de áudio é perdida no processo de codificação. A compressão com perdas não garante que o sinal decodificado seja uma réplica do sinal original. Nesse caso, o sinal decodificado pode ser perceptualmente similar ao sinal original.

Os algoritmos de compressão com perdas empregam em geral modelos psicoacústicos para a codificação de sinais de áudio [MOO 96]. Os modelos psicoacústicos aproveitam as características e limitações do sistema auditivo humano – como mascaramento – para codificar o sinal. Por isso, eles exercem uma degradação inaudível (imperceptível) da qualidade do sinal original. Através dessa degradação, uma quantia de informação de áudio é removida do sinal original e, através disso, diminui-se a quantidade de bits necessária para representar a informação de áudio restante.

3 Escolha das implementações

Ao se escolher uma plataforma para a implementação de um sistema eletrônico, há diversas opções que podem ser vislumbradas. Uma primeira opção é a implementação de software que roda em um processador de uso geral. Um exemplo interessante é o desenvolvimento de software para a arquitetura IBM-PC, já que ela é amplamente difundida no mercado. Entretanto, mesmo no desenvolvimento de software para IBM-PC, ainda há escolhas a serem realizadas. Por exemplo, pode-se desenvolver uma aplicação que utilize apenas o conjunto de instruções padrão PC (8086). Ou então pode-se utilizar instruções especiais multimídia, como as do conjunto de instruções Intel MMX.

Em muitas situações, a utilização de computadores pessoais como plataforma final é impraticável. Por exemplo, ao se projetar um sistema de gravação digital portátil, um processador de uso geral como o Intel Pentium não pode ser usado, pois o seu custo unitário é muito alto para esta aplicação. Nesse caso, dependendo da escala de produção, outras opções devem ser contempladas. Considerando-se média escala de produção, o uso de um processador de sinais digitais (DSP) pode ser vislumbrado, pois o seu custo unitário é menor que o custo de um porcessador de uso geral como o Intel Pentium. Entretanto, ao se considerar uma larga escala de produção, uma aplicação de hardware digital pode ser a melhor escolha. Os assim chamados sistemas embutidos representam a integração funcional entre hardware e software para uma aplicação específica. Nesse caso, a descrição de hardware – que, em geral, pode ser realizada com o apoio de uma linguagem de descrição de hardware (HDL) – pode ser sintetizado em um conjunto de portas programáveis no campo (FPGA), ou detalhado em um processador totalmente customizado. Em geral, processadores totalmente customizados têm um alto custo. Entretanto, através da larga escala de produção, o seu custo é reduzido a um custo adequado para o mercado de eletrônica de consumo

O objetivo deste trabalho é a implementação de blocos do algoritmo MPEG-4 AAC, considerando-se as opções de implementação mencionadas há pouco. Como cada implementação possui o seu próprio nicho de aplicação, cada uma delas é válida como solução final. Além disso, outro objetivo do trabalho é a comparação entre essas implementações, considerando-se custos estimados, tempo de execução e vantagens e desvantagens de cada uma delas. As implementações realizadas neste trabalho são:

Implementação em linguagem C. Essa é a primeira implementação e está fortemente baseada no código-fonte fornecido pela ISO. Esse código fonte é uma implementação tutorial do algoritmo MPEG-4 AAC e ele é utilizado para a análise e comparação com outras implementações.

Implementação em Assembly do processador Intel Pentium. Compiladores de linguagens de alto nível (p. ex. linguagem C) tentam otimizar o código-fonte em todos os estágios da compilação. Entretanto, nem sempre é possível que um compilador

otimize cada aspecto do código, pois compiladores são ferramentas genéricas que têm a finalidade de gerar código executável garantidamente correto. Em muitas situações, algumas otimizações específicas não são vistas pelo compilador. Consequentemente, o desenvolvimento de software em Assembly pode aumentar o desempenho de algoritmos matemáticos intensivos (o que é caso do MPEG-4 AAC), fazendo com que o código executável seja mais eficiente que o código compilado em C correspondente. Por outro lado, a programação em Assembly é bastante dependente de máquina, isto é, o código Assembly escrito para o processador X (com o seu conjunto de instruções específico) não necessariamente rodará em um processador Y. Por isso, a escolha do processador alvo é muito importante. O processador Intel Pentium foi escolhido, já que ele é amplamente utilizado no mercado e tem um conjunto de instruções especial para multimídia. Assim, essa implementação roda em um processador Intel Pentium com suporte a multimídia.

Implementação em processador DSP. Enquanto que as duas primeiras implementações tinham como alvos aplicações em computador pessoal e processadores de uso geral, a implementação em DSP tem como alvo dispositivos dedicados, em que a utilização de processadores de uso geral não é aconselhada, do ponto de vista financeiro. Essa implementação tenta tirar vantagem de características de processadores DSP (como desempenho otimizado em computação matemática intensiva) para aumentar o desempenho do algoritmo MPEG-4 AAC.

Implementação em HDL. A última implementação é uma solução em hardware dedicado ao algoritmo AAC. Em outras palavras, é um projeto de uma arquitetura unicamente dedicada à implementação de um conjunto de módulos AAC. Essa implementação deve ser bem balanceada, isto é, ela deve ser otimizada em termos de tempo de computação, mas também deve ser realística em termos de área e potência.

4 Módulos do MPEG-4 AAC

O ISO-MPEG é um padrão de codificação de vídeo e áudio em alta qualidade e baixas taxas de transferência. Ele foi criado através dos esforços do grupo MPEG (Moving Pictures Experts Group), estabelecido pela ISO/IEC. O ISO-MPEG 4 Audio é composto de várias ferramentas. Entre elas, está o Codificador de Áudio Avançado (AAC).

O codificador AAC recebe amostras de áudio PCM e as transfere para o banco de filtros e para o modelo perceptual. Através de uma transformada do cosseno modificada, o banco de filtros converte as amostras em domínio temporal para o domínio espectral. O modelo perceptual, por sua vez, fornece informações sobre a alocação de bits (número de bits utilizados no processo de codificação) e máximo ruído permitido. Essa informação é utilizada no processo de codificação para assegurar a alta qualidade do sinal codificado nas taxas de transmissão mais baixas.

Após o banco de filtros, os dados espectrais passam por algumas ferramentas, de modo a aperfeiçoar a qualidade ou eficiência de codificação. A ferramenta de formatação de ruído temporal (TNS) é utilizada para o controle do ruído de quantização no domínio temporal. Para isso, utiliza-se um processo de filtragem no domínio espectral. O processo de acoplamento de intensidade é utilizado em canais estéreo para reduzir o número de bits necessários para a codificação de dois canais. Isso é realizado através da codificação de dois canais como um canal único. A substituição de ruído perceptual (PNS) é utilizada na codificação de dados espectrais ruidosos, que são codificados como sendo ruído branco ao invés de serem codificados como coeficientes espectrais; para isso, utiliza-se uma indicação de nível de energia do ruído. Enquanto que a ferramenta PNS é utilizada para sinais ruidosos, a ferramenta de predição a longo prazo (LTP) é utilizada para aperfeiçoar a codificação de sinais tonais. Para isso, aplica-se um processo de predição nas amostras do domínio temporal e transmite-se apenas o erro (indicado pela diferença entre o sinal predito e o sinal real). A decisão centro/lados (M/S) é utilizada em sinais estéreo e determina se canais estéreo devem ser codificados no formato usual esquerdo-e-direito, ou no formato centro-e-lados.

Os dados espectrais são quantizados pelo processo de quantização AAC, que é composto por quatro sub-blocos: quantização dos fatores de escala, quantização dos coeficientes espectrais, codificação sem ruído e laço de controle de taxa e distorção. Esses blocos utilizam informações fornecidas pelo modelo perceptual para assegurar alta qualidade e eficiência de codificação nos dados espectrais codificados.

No último passo, os dados espectrais quantizados e as informações adicionais das ferramentas AAC são codificadas na cadeia de bits. Um processo de checagem CRC é empregado adicionalmente à cadeia de bits, de modo a reduzir a sua vulnerabilidade a erros de transmissão. Finalmente, a cadeia de bits está pronta para a transmissão ou o armazenamento.

5 Implementação em linguagem C

A implementação em C consiste no código-fonte fornecido pela ISO-MPEG (versão 990224). Esse código-fonte está completamente escrito em C e ele foi concebido de maneira a ser o mais portável possível. Ele foi utilizado como base na análise do tempo médio de computação e nas implementações e otimizações em outras arquiteturas. Nenhuma modificação ou otimização foi realizada nesse código; apenas adaptações necessárias para fazê-lo compilar na plataforma PC.

O tempo de computação médio para cada quadro foi determinado. O código-fonte C original foi compilado em um IBM-PC utlizando-se o compilador Borland C++ 5.0. Dois processadores foram utilizados nessa análise: o Intel Pentium II 350 MHz e o Intel Pentium III 750 MHz. Vinte arquivos de entrada em qualidade de CD (16-bit 44-kHz stereo) foram utilizados, com duração sonora de 3,291 s a 10,029 s. A soma de

todos os arquivos corresponde a 113,0150 s de som utlizados na análise. Para a análise, a taxa de codificação de 320 kbps foi utlizada.

O tempo de computação no Pentium II foi de 0,836625 s, enquanto que no Pentium III foi de 0,365897 s. A aceleração ficou em 2,2865 vezes. Assim, a execução do algoritmo no processador Pentium III 750 MHz resulta em uma aceleração de 2,2865 vezes, em comparação à execução em Pentium II 350 MHz.

Analisando-se os módulos internos do algoritmo, observou-se que os módulos LTP e Psicoacústico são os módulos que tomam a maior parte do tempo de computação. Percentualmente, eles correspondem a 65,27% e 24,86% do tempo de computação médio, o que, somado, fornece o valor de 90,13%. Por isso, a redução do tempo de computação desses módulos é necessária e, através dela, há uma redução sensível no tempo de computação total do algoritmo.

A análise do módulo LTP mostrou que 96,85% do tempo de computação desse módulo está concentrado numa função chamada *pitch* (tom). Dentro dessa função, a análise mostrou ainda que 99,95% do tempo de computação está concentrado em uma rotina que busca o melhor atraso para o cálculo de correlação. Por isso, a otimização dessa rotina deve reduzir o tempo de computação do módulo LTP de maneira sensível e, da mesma forma, do algoritmo AAC como um todo.

A análise do tempo de computação do módulo Psicoacústico mostrou que o tempo de computação é gasto principalmente em três funções: 42,80% no cálculo da convolução, 32,38% no cálculo da medida de não-previsibilidade e 18,42% na transformada rápida de Fourier (FFT). Ao se somar esses valores, percebe-se que 93,60% do tempo de computação é gasto nessas funções, o que indica a necessidade de otimização dessas funções.

6 Implementação em Assembly

Na implementação em Assembly, a rotina de busca de atraso da correlação (do módulo LTP) foi completamente reescrita em Assembly do processador Intel-PC, utilizando-se instruções MMX. O montador Microsoft MASM 6.15.8803 foi utilizado para montar a rotina.

Originalmente, a rotina utilizava variáveis em ponto-flutuante. Como as instruções MMX trabalham com valores inteiros, o algoritmo foi adaptado para trabalhar com variáveis inteiras. Adicionalmente, foi feita uma análise de precisão, de modo a garantir a correção dos dados.

Os resultados do desempenho dessa implementação, considerando-se a computação de um quadro e sinais estéreo foram os seguintes. O tempo de codificação médio na versão otimizada foi de 0,376642 s (PII 350 MHz) e 0,198002 s (PIII 750 MHz). O tempo de computação médio da rotina de busca de atraso na versão original

foi de 0,198002 s (PII 350 MHz) e 0,198002 s (PIII 750 MHz). Na versão otimizada, esse tempo de computação passou para 0,035364 s (PII 350 MHz) e 0,027034 s (PIII 750 MHz). A aceleração no tempo de computação, em comparação com a implementação original foi de 2,2213 vezes (PII 350 MHz) e 1,8479 vezes (PIII 750 MHz). A aceleração no tempo de computação do módulo LTP, em relação ao módulo na versão original foi de 6,9448 vezes (PII 350 MHz) e 5,1857 vezes (PIII 750 MHz). A aceleração da rotina de busca do atraso de correlação – comparando-se a rotina original com a otimizada – foi de 8,7299 vezes (PII 350 MHz) e 5,0401 vezes (PIII 750 MHz).

As otimizações no módulo Psicoacústico consistiram na codificação à mão em Assembly de rotinas computacionalmente intensivas. Entretanto, continuou-se a usar variáveis em ponto-flutuante, da mesma forma como no código original. Por isso, instruções MMX não foram utilizadas.

Assim, por exemplo, por causa da reescrita da rotina de medida de não-predibilidade, atingiu-se uma aceleração de 5,4753 vezes (para o Pentium II 350 MHz) no tempo de computação médio da rotina.

Além disso, um esquema de tabela de consulta foi utilizado em substituição à função interna de cálculo de espalhamento. Para isso, ao invés de se reutilizar a função a cada vez que um novo valor era exigido, uma simples consulta a uma tabela era suficiente para se obter um novo valor. Além disso, não foi preciso o uso de interpolação e, também, nenhuma perda decorreu do uso desse esquema. Duas tabelas de consultas foram utilizadas: uma para janelas longas e outra para janelas curtas. As tabelas ocuparam 40,5 kbytes (para janelas longas) e 18 kbytes (para janelas curtas). O uso de tabelas levou a uma aceleração de 8,7286 vezes (para Pentium II 350 MHz) no tempo de computação médio da rotina de convolução.

Os resultados da análise do tempo de computação médio para essa implementação (em segundo por quadro), considerando-se todas as otimização (nos módulos LTP e Psicoacústico) foram os seguintes. O tempo de computação médio para o codificador (utilizando-se sinais estéreo) para a versão otimizada foi de 0,217543 s (PII 350 MHz) e 0,143212 s (PIII 750 MHz). A aceleração do tempo de computação do codificador foi de 3,8458 vezes (PII 350 MHz) e 2,5549 vezes (PIII 750 MHz). A aceleração do tempo de computação do módulo Psicoacústico foi de 2,4749 vezes (PII 350 MHz) e 2,3427 (PIII 750 MHz).

7 Implementação em DSP do módulo LTP

Essa seção apresenta a implementação em DSP do módulo LTP. Para isso, apresenta-se os resultados da implementação nos processadores Texas Instruments TMS320C31 e Motorola DSP56309.

No desenvolvimento da implementação no processador em ponto-flutuante Texas Instruments TMS320C31, a ferramenta Code Composer (da Texas Instruments) foi utilizada. Essa ferramenta permite a compilação de código C e ligação entre rotina em C e Assembly. Além disso, ela permite a simulação da execução do código. Essa simulação é otimista, isto é, os resultados da simulação podem apresentar tempo de computação médio melhor (menor) do que o de uma execução real no processador. Para se obter o tempo de computação médio simulado, um processador operando a 25 MHz (o que equivale a um ciclo de relógio de 40 ns) foi escolhido.

Duas implementações da rotina de busca de atraso da correlação (do módulo LTP) foram consideradas. Ambas foram codificadas em linguagem C. A primeira versão em C utilizou variáveis em ponto-flutuante, enquanto que a segunda utilizou variáveis inteiras.

O tempo de computação da rotina em ponto-flutuante foi de 2,0334 s. A seguir apresenta-se a comparação desse resultado com os resultados das implementações em PC. Apresentam a aceleração da implementação em PC (na versão em linguagem C) em relação à implementação em DSP foi de 6,5851 vezes (PII 350 MHz) e 14,9206 vezes (PIII 750 MHz). Quanto à implementação em PC com módulos em linguagem Assembly, a aceleração da implementação em PC (em relação à implementação em DSP) foi de 57,4878 vezes (PII 350 MHz) e 75,2016 (PIII 750 MHz).

Na versão que utiliza variáveis inteiras, o tempo de computação foi de 1,0479 s. A aceleração da implementação em PC (na versão em linguagem C) em relação à implementação em DSP foi de 3,3946 vezes (PII 350 MHz) e 7,6915 vezes (PIII 750 MHz). Em relação à implementação em PC que utiliza módulos em linguagem Assembly, a aceleração da implementação em PC foi de 29,6347 vezes (PII 350 MHz) e 38,7660 (PIII 750 MHz).

Uma outra comparação possível decorre da normalização dos resultados obtidos com o processador C31. Nesse caso, a freqüência de operação de todos os processadores é normalizada. Para isso, extrapola-se a frequência de operação usual do processador C31, de modo que ela se iguale à dos processadores Pentium II e III.

O tempo de computação normalizados do C31, na versão em ponto-flutuante, foi de 0,145243 s (PII 350 MHz) e 0,067780 (PIII 750 MHz). Na versão inteira, esse tempo normalizado passou para 0,074848 s (PII 350 MHz) e 0,034929 s (PIII 750 MHz). Assim, considerando-se a normalização dos tempos de computação, a aceleração da implementação em Pentium, na versão em C, em relação à implementação DSP em ponto-flutuante ficou em 0,4705 vezes (PII 350 MHz) e 0,4975 vezes (PIII 750 MHz). Para a versão com variáveis inteiras, a aceleração da implementação em PC ficou em 0,2424 (PII 350 MHz) e 0,2564 (PIII 750 MHz). Comparando-se com a implementação em PC em linguagem Assembly, a aceleração da implementação em Pentium em relação à implementação DSP em ponto-flutuante ficou em 4,1071 vezes (PII 350 MHz) e 2,5072 (PIII 750 MHz). A aceleração da implementação em Pentium em relação à

implementação DSP que utiliza variáveis inteiras ficou em 2,1165 vezes (PII 350 MHz) e 1,2920 vezes (PIII 750 MHz).

Pode-se observar que, nesse caso, a implementação no processador C31 inteira é melhor que a maior parte das implementações em PC. Entretanto, o desempenho da implementação em Assembly (MMX) ainda é melhor que o da versão em ponto-flutuante do C31.

O mesmo código C utilizado na implementação no processador Texas Instruments TMS320C31 foi utilizado na implementação em Motorola DSP56309. O tempo de computação da rotina em ponto-flutuante foi de 6,782478 s. A comparação desse resultado com os resultados das implementações em PC indica que a aceleração da implementação em PC em relação à implementação em DSP, considerando-se a versão em C, ficou em 21,9693 vezes (PII 350 MHz) e 49,7778 vezes (PIII 750 MHz). Comparando-se a implementação em PC na versão em Assembly com a implementação em DSP, a aceleração ficou em 191,7905 vezes (PII 350 MHz) e 250,8869 (PIII 750 MHz).

O tempo de computação – na versão inteira da implementação em DSP – foi de 0,721177 s. Assim, a aceleração da implementação em PC em relação à implementação em DSP, considerando-se a versão em C, ficou em 2,3360 vezes (PII 350 MHz) e 5,2929 vezes (PIII 750 MHz). Comparando-se a implementação em PC na versão em Assembly com a implementação em DSP, a aceleração ficou em 20,3930 vezes (PII 350 MHz) e 26,6767 (PIII 750 MHz).

A seguir estão os resultados normalizados para o processador DSP56309. O tempo de computação normalizado no DSP56309, considerando-se a versão em ponto-flutuante, foi de 1,278981 s (PII 350 MHz) e 0,596858 s (PIII 750 MHz). Considerando-se a versão com variáveis inteiras, o tempo de computação normalizado ficou em 0,135993 s (PII 350 MHz) e 0,063464 s (PIII 750 MHz). Baseando-se nessa normalização dos tempos de computação, a aceleração da implementação em Pentium (na versão em C) em relação à implementação DSP em ponto-flutuante ficou em 4,1428 vezes (PII 350 MHz) e 4,3804 vezes (PIII 750 MHz). Em relação à versão com variáveis inteiras, a aceleração da implementação em PC ficou em 0,4405 (PII 350 MHz) e 0,4658 (PIII 750 MHz). Quanto à implementação em PC em linguagem Assembly, a aceleração da implementação em Pentium em relação à implementação DSP em ponto-flutuante foi de 36,1662 vezes (PII 350 MHz) e 22,0781 (PIII 750 MHz). A aceleração da implementação em Pentium em relação à implementação DSP que utiliza variáveis inteiras foi de 3,8455 vezes (PII 350 MHz) e 2,347 vezes (PIII 750 MHz).

Na maioria dos casos, mesmo considerando-se a normalização dos resultados, a implementação em PC ainda é melhor que a implementação no processador DSP. Apenas na comparação normalizada entre a versão inteira do DSP56309 e a versão em ponto-flutuante do PC, o desempenho do DSP é melhor.

Geralmente, espera-se que processadores DSP sejam mais eficientes no processamento de algoritmos com computação matemática intensiva. Entretanto, isso não foi observado na comparação entre as implementações. Mesmo que a comparação normalizada tenha fornecido alguns resultados favoráveis aos processadores DSP, em geral, os processadores de uso geral Pentium se saíram melhor na comparação.

Entretanto, quando se considera as necessidades de sistemas embarcados (como baixo custo unitário e baixo consumo de potência), a situação é outra. Os processadores de uso geral normalmente não são adequados para esse tipo de aplicação, pois eles possuem um alto custo unitário e um alto consumo de potência. Por isso, ao se projetar um sistema embarcado, geralmente utiliza-se processadores DSP em lugar de processadores de uso geral.

8 Implementação HDL do módulo LTP

O VHDL é uma linguagem para descrição e simulação de hardware. Ela é bastante adequada para a descrição de circuitos algorítmicos (como é o caso de microprocessadores e autômatos finitos). Além disso, ela é bastante utilizada como entrada para a síntese em FPGAs.

A rotina de busca de atraso da correlação do módulo LTP foi descrita como uma máquina algorítmica em VHDL. A descrição foi sintetizada para a família de dispositivos Flex10k da Altera. O software MaxplusII da Altera foi utilizado para síntese e simulação da descrição.

Três diferentes descrições foram desenvolvidas para este trabalho, sendo que todas elas foram escritas em VHDL. As diferenças entre elas são as seguintes:

- O uso de um processador ltp_lag_proc separado (isto é, descrito como um módulo VHDL separado) ou a sua descrição integrada em um processador principal. Na versão separada, o *ltp_lag_proc* é responsável pelas operações matemáticas, enquanto que o processador central lê os dados da memória e controla os estados do processador ltp_lag_proc. Na versão integrada, há apenas um processador central (*ltp_lag*) que lê os dados e realiza os cálculos.

- O uso de unidades multiplicador-acumulador (MAC) separadas ou o uso de operadores padrões da Altera (disponíveis em bibliotecas ou em assim chamadas mega-funções). As unidades MAC mencionadas são baseadas em somadores e multiplicadores bit-seriais.

As diferenças entre as três implementações realizadas em VHDL são as seguintes. Na primeira implementação, o processador *ltp_lag_proc* está separado do processador principal e utiliza-se unidades MAC. Na segunda implementação, o processador *ltp_lag_proc* está integrado ao processador principal (isto é, há apenas um

processador que realiza todas as operações) e utiliza-se unidades MAC. Na terceira implementação, há apenas um processador principal e utiliza-se operadores da Altera.

As três implementações foram simuladas utilizando-se o software fornecido pela Altera. Os parâmetros levados em conta para a comparação entre as implementações foram o número de ciclos para a computação de da rotina de procura em um quadro, a freqüência máxima de operação e o tempo de computação. Na primeira implementação, o número de ciclos é de 104.958.902 ciclos, a freqüência máxima de operação é de 12,54 MHz e o tempo de computação é de 8.369928 s. Na segunda implementação, o número de ciclos é de 102.334.904 ciclos, a freqüência máxima de operação é de 16,31 MHz e o tempo de computação é de 6,274366 s. Na terceira implementação, o número de ciclos é de 12.601.338 ciclos, a freqüência máxima de operação é de 7,99 MHz e o tempo de computação é de 1,575985 s. O desempenho da terceira implementação é de 5,3109 vezes em relação ao desempenho da primeira implementação e de 3,9812 vezes em relação ao desempenho da segunda implementação. A principal razão para esse desempenho otimizado da terceira implementação é o número reduzido de ciclos desta implementação. Esse número reduzido de ciclos, por sua vez, origina-se na utilização dos operadores da biblioteca da Altera e em uma máquina de estados mais simples que a das duas outras implementações.

Comparando-se essa terceira implementação em VHDL com as implementações em outras plataformas, pode-se observar que a implementação em VHDL é somente mais rápida que as implementações em DSP (tanto para o 56309, como para o C31) que utilizam ponto-flutuante. No entanto, ao se normalizar os resultados de todas as implementações, os resultados do VHDL são melhores. Nesse caso, a implementação em VHDL é somente um pouco menos eficiente que a implementação em MMX no Pentium II. A explicação para isso é o número reduzido de ciclos da implementação em VHDL.

Quanto à baixa freqüência máxima de operação da implementação em VHDL, deve-se mencionar que todas as implementações realizadas em VHDL apresentam um forte caráter comportamental. Por isso, pode-se levantar a hipótese de que esse estilo de descrição comportamental pode não ser o mais apropriado para o algoritmo de busca do módulo LTP. Assim, pode-se supor que esse estilo de descrição atinga uma freqüência máxima de operação mais baixa que a de um ASIC dedicado, que possui uma arquitetura otimizada para o algoritmo. Desta forma, a escolha entre qual estilo de descrição deve ser adotado em um projeto é muito importante, pois isso pode determinar o desempenho final do algoritmo.

9 Conclusões

Neste trabalho, o padrão MPEG-4 AAC foi estudado, resultando em um texto tutorial em inglês que cobre todos os aspectos do codificador AAC. Além disso, alguns

princípios de Psicoacústica foram apresentados no texto, desta forma esclarecendo o estudo do modelo perceptual do codificador AAC.

O MPEG-4 AAC é um algoritmo de codificação contido na parte de áudio do padrão ISO MPEG-4 de codificação e transmissão de conteúdo multimídia. O MPEG-4 AAC foi especialmente especificado para a codificação de áudio genérico (p.ex.: fala e música) em baixas taxas de codificação. Trata-se de um aperfeiçoamento de algoritmos de codificação de áudio mais antigos criados pelo grupo MPEG, como o MPEG-1 camada III. Os principais componentes do codificador AAC são o banco de filtros, o modelo perceptual (também chamado de modelo psicoacústico) e o módulo de quantização. O banco de filtros converte as amostras de entrada a partir do domínio tempo para o domínio espectral (gerando desta forma coeficientes espectrais) através de uma transformada modificada do cosseno. O modelo perceptual utiliza os dados do domínio espectral para garantir alta qualidade do sinal de áudio codificado, provendo desta forma informações sobre alocação de bits e ruído máximo permitido para o processo de quantização. Os dados espectrais são quantizados pelo processo de quantização do AAC, que é composto de quatro subblocos: o bloco de quantização de fatores de escala, o bloco de quantização de coeficientes espectrais, o bloco de codificação sem adição de ruído e o bloco de laço de controle de taxa de transmissão e distorção.

O modelo perceptual baseia-se em regras derivadas do campo da Psicoacústica. Mecanismos de mascaramento do sistema auditivo humano permitem que ruído de quantização seja adicionado ao sinal codificado sem afetar a qualidade. Esse ruído de quantização é adicionado ao sinal de áudio no momento em que ele é recodificado utilizando-se uma quantidade menor de bits. Desta forma, os mecanismos de mascaramento determinam a redução de bits permitida que não degrade a qualidade percebida do sinal.

Uma grande parte deste trabalho foi dedicada à implementação do codificador MPEG-4 AAC. Neste trabalho, três plataformas de implementação diferentes foram utilizadas e quatro implementações foram desenvolvidas. A escolha das plataformas foi guiada pelas características atuais do mercado, como disponibilidade, portabilidade, custos, etc. Por exemplo, ao mesmo tempo que processadores de uso geral estão amplamente disponíveis, o seu custo unitário tende a ser alto (ao menos, muito maior do que o custo unitário de um processador DSP) e eles não são tão portáteis quanto processadores DSP. Por estes motivos, as plataformas escolhidas foram processadores de uso geral (Intel Pentium II e III), processadores DSP (Texas 31 e Motorola DSP56309) e implementação em hardware (Altera Flex 10k FPGA).

Uma versão de domínio público do codificador MPEG-4 AAC em linguagem C foi utilizada como base para o desenvolvimento das implementações. Essa versão do codificador é bastante tutorial. Isto é, ela existe como base para guiar desenvolvedores na criação de seus próprios codificadores. Por isso, ela não representa uma

implementação eficiente do codificador e não reflete as otimizações realizadas pela indústria de áudio em seus esforços para implementar codificadores de áudio eficientes, isto é, codificadores rápidos que assegurem alta qualidade e eficiência de codificação.

O código em linguagem C foi empregado na análise do tempo de computação do codificador. Essa análise revelou que os módulos do algoritmo AAC que mais consumiam o tempo de codificação eram a predição a longo prazo (LTP) e o módulo psicoacústico. As otimizações realizadas nesses módulos permitiram melhorar o tempo de computação do algoritmo. Desta forma, o trabalho de implementação foi direcionado para o desenvolvimento de versões aperfeiçoadas (isto é, mais rápidas) desses módulos em diversas plataformas.

O módulo LTP foi desenvolvido em todas as plataformas mencionadas. Os resultados das implementações apresentaram relações interessantes entre as plataformas. Por exemplo, demonstrou-se que as implementações em PC têm vantagens em termos de desempenho sobre todas as implementações em outras plataformas (DSP e hardware sintetizado em FPGA). Além disso, as implementações demonstraram que otimizações em Assembly para laços de computação intensiva e rotinas críticas contribuem para uma melhora significativa no desempenho. Entretanto, comparações que levaram em conta a normalização da freqüência de relógio demonstraram que a implementação em FPGA-sintetizado foi uma das mais eficientes, devido ao número reduzido de ciclos necessários na computação do módulo LTP. Como os processadores DSP também apresentam uma freqüência de relógio relativamente baixa, esse também foi o caso de um deles (o C31). Por isso, essas comparações revelaram que a freqüência de relógio é uma característica importante em uma plataforma alvo quando se trata de tempo de computação.

Futuros trabalhos que utilizarem esta tese de mestrado como base podem tornar as implementações em DSP e VHDL mais eficientes em termos de tempo de computação. No caso das implementações em DSP, isso pode ser atingido através de otimizações manuais a partir do código Assembly gerado pelo compilador C. Entretanto, como os mnemômicos variam de processador para processador, é necessário que um processador DSP alvo seja escolhido em primeiro lugar. Além disso, essa escolha do processador deve estar baseada em comparações entre diversos processadores DSP, considerando-se a sua adequação para o desenvolvimento de algoritmos de codificação de áudio.

No caso da implementação em VHDL, o reestruturamento da máquina de controle das descrições através de uma simplificação do algoritmo pode resultar em um circuito sintetizado mais eficiente. Além disso, a utilização de outras ferramentas de desenvolvimento (como, p.ex., standard-cells) pode fornecer resultados melhores que os obtidos com FPGAs.

Outra sugestão é a utilização de processadores embebidos nas implementações e comparações, já que eles podem apresentar custos baixos e portabilidade (da mesma

forma como processadores DSP) e podem ser tão flexíveis quanto processadores de uso geral.

References

[ALT 2001] ALTERA CORPORATION. **Flex 10K Embedded Programmable Logic Device Family Data Sheet**. New York, 2001.

[ARM 2001] ARM. **Arm Homepage**. Available at: <http://www.arm.com/>. Visited on Nov. 2001.

[BLE 78] BLESSER, Barry A. Digitalization of Audio: A Comprehensive Examination of Theory, Implementation, and Current Practice. **Journal of the Audio Engineering Society**, New York, v. 26, n. 10, Oct. 1978.

[BOS 99] BOSI, Marina. Filter banks in Perceptual Audio Coding. In: AES INTERNATIONAL CONFERENCE ON HIGH QUALITY AUDIO CODING, 17., 1999. **Proceedings...** New York: Audio Engineering Society, 1999.

[BRA 96a] BRANDENBURG, Karlheinz. Introduction to Perceptual Coding. In: **Collected Papers on Digital Audio Bit-Rate Reduction**. New York: Audio Engineering Society, 1996.

[BRA 96b] BRANDENBURG, Karlheinz; STOLL, Gerhard. ISO-MPEG-1 Audio: A Generic Standard for Coding of High-Quality Digital Audio. In: **Collected Papers on Digital Audio Bit-Rate Reduction**. New York: Audio Engineering Society, 1996.

[BRA 98] BRANDENBURG, Karlheinz; BOSI, Marina ISO/IEC MPEG-2 Advanced Audio Coding: Overview and Applications. In: AES CONVENTION, 103. and 104., 1998. **Proceedings...** New York: Audio Engineering Society, 1998.

[BRA 99] BRANDENBURG, Karlheinz. MP3 and AAC Explained. In: AES INTERNATIONAL CONFERENCE ON HIGH QUALITY AUDIO CODING, 17., 1999. **Proceedings...** New York: Audio Engineering Society, 1999.

[BRU 98] BRUEKERS, Fons et al. Improved Lossless Coding of 1-bit Audio Signals. In: AES CONVENTION, 103. and 104., 1998. **Proceedings...** New York: Audio Engineering Society, 1998.

[CHA 99] CHANG, K.C. **Digital Systems Design with VHDL and Synthesis**. New York: IEEE Computer Society Press, 1999.

[CHE 99] CHEN, Jin; TAI, Heng-Ming. MPEG-2 AAC Coder on a Fixed-Point DSP. In: INTERNATIONAL CONFERENCE ON CONSUMER ELECTRONICS, ICCE, 1999. **Proceedings...** New York: ICCE, 1999.

[COO 99] COOK, Perry (Ed.). **Music, Cognition, and Computerized Sound**. Cambridge: The MIT Press, 1999.

[EDL 98] EDLER, Bernd; PURNHAGEN, Heiko. Concepts for Hybrid Audio Coding Schemes Based on Parametric Techniques. In: AES CONVENTION, 105., 1998. **Proceedings...** New York: Audio Engineering Society, 1998.

[EYR 2001] EYRE, Jennifer. The Digital Signal Processor Derby. **IEEE Spectrum**, New York, v. 38, n. 06, 2001.

[FIS 99] FISHER, Randall J.; DIETZ, Henry G. **Compiling For SIMD Within A Register**. Available at: <http://shay.ecn.purdue.edu/~swar/>. Visited on Nov. 2001.

[GAJ 97] GAJSKY, D. **Principles of Digital Design**. New York: Prentice-Hall, 1997.

[GER 99] GERZON, M.A. et al. The MLP Lossless Compression System. In: AES INTERNATIONAL CONFERENCE ON HIGH QUALITY AUDIO CODING, 17., 1999. **Proceedings...** New York: Audio Engineering Society, 1999.

[GRA 98] GRAY, Robert M.; NEUHOFF, David L. Quantization. In: **IEEE Transactions on Information Theory**, New York, v. 44, n. 6, Oct. 1998.

[GRI 99] GRILL, Bernhard. MPEG-4 General Audio Coder. In: AES INTERNATIONAL CONFERENCE ON HIGH QUALITY AUDIO CODING, 17., 1999. **Proceedings...** New York: Audio Engineering Society, 1999.

[HAR 98] HARTMANN, William M. **Signals, sound, and sensation**. New York: Springer-Verlag, 1998

[HIL 98] HILPERT, Johannes et al. Implementing ISO/MPEG-2 Advanced Audio Coding in Realtime on a Fixed Point DSP. In: AES CONVENTION, 105., 1998. **Proceedings...** New York: Audio Engineering Society, 1998.

[HIL 2000] HILPERT, Johannes et al. Real-Time Implementation of the MPEG-4 Low Delay Advanced Audio Coding Algorithm (AAC-LD) on Motorola

DSP56300. In: AES CONVENTION, 108., 2000. **Proceedings...** New York: Audio Engineering Society, 2000.

[HUA 99] HUANG, Dong-Yan et al. Implementation of the MPEG-4 advanced audio coding encoder on ADSP-21060 SHARC. In: IEEE INTERNATION SYMPOSIUM ON CIRCUITS AND SYSTEMS, 1999. **Proceedings...** New York: Institute of Electrical and Electronics Engineers, 1999.

[IBM 2001] IBM CORPORATION. **PowerPC Homepage**. Available at: <http://www.chips.ibm.com/products/powerpc/>. Visited on Nov. 2001.

[JES 2000] JESPERS, P. Converter performances. **Microelectronic Engineering**, New York, n. 54, 2000.

[LIN 2001] LINCOLN, Bosse. **An Experimental High Fidelity Perceptual Audio Coder**. Available at: <http://ccrma-www.stanford.edu/~bosse/proj/proj.html>. Visited on Nov. 2001.

[LIU 99] LIU, Chi-Min; LEE, Wen-Chieh. A Unified Fast Algorithm for Cosine Modulated Filter Banks in Current Audio Coding Standards. **Journal of the Audio Engineering Society**, New York, v. 47, n. 12, Dec. 1999.

[MIC 2001] MICROSOFT CORPORTATION. **Windows CE Homepage**. Available at: <http://www.microsoft.com/windows/embedded/ce/>. Visited on Nov. 2001.

[MIP 2001] MIPS. **Mips Homepage**. Available at: <http://www.mips.com>. Visited on Nov. 2001.

[MOO 96] MOORE, Brian. Masking the Human Auditory System. In: **Collected Papers on Digital Audio Bit-Rate Reduction**. Audio Engineering Society, 1996.

[MOT 2000] MOTOROLA INC. **DSP56300 Family Manual**. New York, 2000.

[MPG 93] ISO/IEC. **Coding of Moving Pictures and Associated Audio for Digital Storage Media up to 1.5 Mbit/s – Part 3: Audio**, International Standard 11172-3. Zurich, 1993.

[MPG 94] ISO/IEC. **Generic Coding of Moving Pictures and Associated Audio – Part 3**, International Standard 13818-3. Zurich, 1994.

[MPG 97] ISO/IEC. **Generic Coding of Moving Pictures and Associated Audio – Part 7**, International Standard 13818-7. Zurich, 1997.

[MPG 99] ISO/IEC. **Information Technology – Coding of Audiovisual Objects – Part 3**, Final Committee Draft 14496-3. Zurich, 1999.

[MPG 2000] ISO/IEC. **MPEG Audio Page**. Available at: <http://www.tnt.uni-hannover.de/project/mpeg/audio/>. Visited on Nov. 2001.

[NIS 99] NISHIGUCHI, Masayuki. MPEG-4 Speech Coding. In: AES INTERNATIONAL CONFERENCE ON HIGH QUALITY AUDIO CODING, 17., 1999. **Proceedings...** New York: Audio Engineering Society, 1999.

[NOL 97] NOLL, Peter. MPEG Digital Audio Coding. **IEEE Signal Processing Magazine**, New York, v. 14, n. 5, Sept. 1997.

[OPP 98] OPPENHEIM, Alan V. Schafer, Ronald W. **Discrete-Time Signal Processing**. Upper Saddle River: Prentice-Hall Inc, 1998

[ORF 96] ORFANIDIS, Sophocles. **Introduction to Signal Processing**. Upper Saddle River: Prentice-Hall, 1996.

[PAT 96] PATTERSON, David A.; HENNESSY, John L. **Computer Architecture**. 2nd ed. New York: Morgan Kaufmann, 1996.

[PEL 97] PELEG, Alex; WILKIE, Sam; WEISER, Uri. Intel MMX for Multimedia PCs. **Communications of the ACM**, New York, v. 40, n. 1, p. 25–38, Jan. 1997.

[PRE 92] PRESS, William et al. **Numerical Recipes in C**. 2nd ed. Cambridge: Cambridge University Press, 1992.

[PUR 98] PURNHAGEN, Heiko; EDLER, Bernd; FEREKIDIS, Charalampos. Object-Based Analysis/Synthesis Audio Coder for Very Low Bit Rates. In: AES CONVENTION, 103. and 104., 1998. **Proceedings...** New York: Audio Engineering Society, 1998.

[PUR 99] PURNHAGEN, Heiko. An Overview of MPEG-4 Audio Version 2. In: AES INTERNATIONAL CONFERENCE ON HIGH QUALITY AUDIO CODING, 17., 1999. **Proceedings...** New York: Audio Engineering Society, 1999.

[QUA 97] QUACKENBUSH, S.R.; JOHNSTON, J.D. Noiseless coding of quantized spectral components in MPEG-2 Advanced Audio Coding. In: IEEE WORKSHOP ON APPLICATIONS OF SIGNAL PROCESSING TO AUDIO AND ACOUSTICS, 1997. **Proceedings...** New York: Institute of Electrical and Electronic Engineers, 1997.

[SAB 96] SABLATASH, Mike; COOKLEV, Todor. Compression of High-Quality Audio Signals, Including Recent Methods Using Wavelet Packets. **Digital Signal Processing**, New York, v. 6, p. 96 – 107, 1996.

[SCH 99a] SCHEIRER, Eric; VERCOE, Barry L. SAOL: The MPEG-4 Structured Audio Orchestra Language. **Computer Music Journal,** Cambridge, v. 23, n. 2, p. 31-51, Summer 1999.

[SCH 99b] SCHEIRER, Eric; KIM, Youngmoo E. Generalized audio coding with MPEG-4 Structured Audio. AES INTERNATIONAL CONFERENCE ON HIGH QUALITY AUDIO CODING, 17., 1999. **Proceedings...** New York: Audio Engineering Society, 1999.

[SCH 99c] SCHEIRER, Eric; VÄÄNÄNEN, Riitta; HUOPANIEMI, Jyri. AudioBIFS: Describing Audio Scenes with the MPEG-4 Multimedia Standard. **IEEE Transactions on Multimedia**, New York, v. 1, n. 3, p. 237-250, Sept. 1999.

[SPA 2000] SPANIAS, Andreas. Speech Coding: A Tutorial Review. **Proceedings of the IEEE**, New York, v. 82, n. 10, Oct. 1994.

[STO 96] STOLL, Gerhard. ISO-MPEG-2 Audio: A Generic Standard for the Coding of Two-Channel and Multichannel Sound. **Collected Papers on Digital Audio Bit-Rate Reduction**. New York: Audio Engineering Society, 1996.

[TEX 97] TEXAS INSTRUMENTS. **TMS320C3x User's Guide**. New York, 1997.

[VER 98] VERBAKEL, Jan; VAN DE KERKHOF, Leon; MAEDA, Muneyasu; INAZAWA, Yoshizumi. Super Audio CD Format. In: AES CONVENTION, 103. and 104., 1998. **Proceedings...** New York: Audio Engineering Society, 1998.

[WID 89] WIDDER, David Vernon. **Advanced Calculus**. 2nd ed. New York: Dover Publications Inc., 1989.

[WIN 2001] WIND RIVER. **VxWorks RTOS Homepage**. Available at: <http://www.windriver.com/products/html/vxwks54.html>. Visited on Nov. 2001.

[WYL 96] WYLIE, Fred. apt-X100: Low-Delay, Low-Bit-Rate Subband ADPCM Digital Audio Coding. **Collected Papers on Digital Audio Bit-Rate Reduction**. New York: Audio Engineering Society, 1996.