ROYAL INSTITUTE
OF TECHNOLOGY

A Study of the ITU-T G.729 Speech Coding Algorithm - Implementation on the Capella ASIC

Master's Thesis

by

Johannes Celander

Department of Access Signal Processing
at Ericsson AB

Department of Signals, Sensors and Systems
at Royal Institute of Technology

Supervisor
M.Sc. Tore André
AL/EAB Access Signal Processing
Ericsson Broadband Access
Stockholm, Sweden

Examiner
Ph.D. Jonas Samuelsson
Sound and Image Processing Lab
Signals, Sensors & Systems
Royal Institute of Technology (KTH)
Stockholm, Sweden

Abstract

This Master Thesis is done at Ericsson, Access Signal Processing, in Älvjso, Sweden. The Access Signal Processing unit is developing an experimental system for Voice over Internet Protocol (VoIP). The system may use the standardized ITU-T G.729/A speech coder.

The main objective of this Master's Thesis is to study the ITU-T G.729/A speech coding algorithm, and implement it on an Application Specific Integrated Circuit (ASIC) called Capella. Capella is based on the Flexible ASIC concept in which several customized Digital Signal Processors (DSP) cores are embedded in one ASIC. The Flexible ASIC concept also consists of a set of development tools which is used when implementing the G.729/A.

This report also contains a brief overview of speech coding in general, an explanation of the G.729/A algorithm specifically and also outlines the main elements of different VoIP solutions.

The implementation of the G.729/A is based on the ANSI-C code provided by ITU-T. Since the provided code is not intended for implementation purposes it needs to be modified and optimized to work on a DSP. After the respective modifications the G.729A functioned correctly on the Capella. However, further optimization work is required to enable multiple channels on one Capella, which is desirable in Ericsson's experimental VoIP solution.

Acknowledgements

First and foremost, I would like to thank my supervisor at Ericsson, Tore André, who has been a great support throughout my thesis work and always had his door open for questions. A special thank you also to Mahfooz Khedri for his general support and interest in my work, to Anders Lindblad for his help in installing the Flextools and support with UNIX, as well as to Kenneth Johansson for his assistance in the search for function pointers. A thank you also to Jonas Rosenberg for inheriting me his Sunblade computer. My gratitude also to the entire ASP-unit.

I also would like to thank my examiner and supervisor at the Royal Institute of Technology, Jonas Samuelsson, for the time and effort he put into this report and for broadening my views with his constructive criticism.

Finally, I would like to thank my wonderful girlfriend, Corinna, for her support during this work. Thank you for correcting my English and for powering this project by making me a lunch-box every day.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Thesis Background

Ericsson's Access Signal Processing unit is developing an experimental system for Voice over Internet Protocol (VoIP) which is intended to be significantly better than the currently existing VoIP solutions on the market. In order to symbolize the emphasis on a high speech quality solution the experimental system is internally called Telephony over Internet Protocol (ToIP). One possible definition of ToIP is given below.

> "ToIP is different from Voice over IP, because it is intended as a carrier-class service, fully featured and with quality guarantees, to be offered to large number of users, eiter overlapping or substituting classical POTS services" [26].

The experimental ToIP system will use the G.711 speech coder and possibly, as an additional coder, the ITU-T [1] G.729.

The idea of Ericsson's experimental ToIP system is depicted in Figure 1. The thought is that telephone calls will go out via the copper lines to a local telephone station in ordinary fashion. However, then the idea is to encode the speech, place it into packets and send it over Ethernet. In addition to the G.711 and G.729 speech coders, the ToIP system also consists of a telephony interface and an echo-canceler. The speech packages are sent over Ethernet with three protocols: Real Time Protocol (RTP), User Data Protocol (UDP) and Internet Protocol (IP). To provide an overview of the aspects in VoIP the main issues are handled in section 4.

The G.729 is considered as an optional coder due to its low bit-rate performance and high quality of speech. However, since the G.729 is patent protected, one of its disadvantages is that extra cost occur since parts of the G.729 algorithm are patent protected. In the implementation, the G.729 was replaced by the low-complexity version G.729A. The G.729 and G.729A coders are described in section 3, and Section 2 provides a background of speech coding in general.

The G.729A implementation platform for this project is called Capella. Capella is an Application Specific Integrated Circuit (ASIC) consisting of eight customized Digital Signal Processor (DSP) cores. The Capella ASIC is based on the Flexible ASIC concept in which several DSP cores are embedded in one ASIC. The Flexible ASIC concept also includes a set of development tools which were used when implementing the G.729.

The Flexible ASIC concept is described in section 5 and the characteristics of the Capella ASIC are stated in section 6.

1.2 Thesis Goal

The stated goal of this thesis is to study the G.729/A speech coding algorithm and implement it on the Capella ASIC. It is desired that the G.729/A coder is optimized so that several codecs can be run on one Capella, thereby enabling the coder to operate in multi-channel environments. The implementation process can be divided into stages, where the first and most important stage is to modify the, by ITU-T provided, C-source code so that the algorithm functions correctly on Capella. Thereafter, optimization should be performed

[1] ITU-T Telecommunication Standardization Sector of ITU

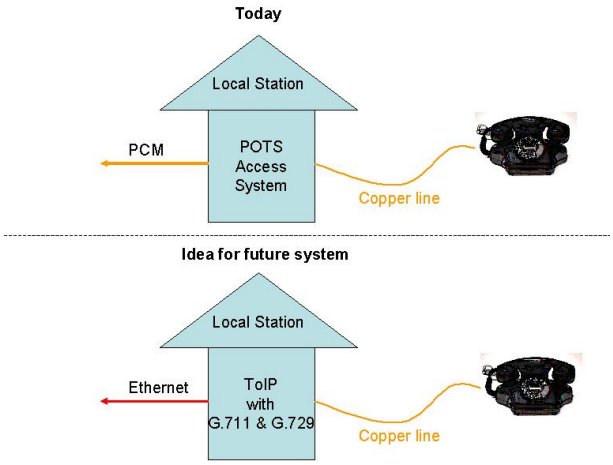

Figure 1: ToIP.

in both, C and Assembly, as far as time allows. If possible, the thesis should also give an estimation of the additional workload required to optimize the coder for multiple channels.

Another part of this thesis is to understand the main aspects of VoIP and investigate different market solutions and research areas. Also, different speech-coding standards are analyzed.

1.3 Thesis Outline

This thesis consists of nine sections. Following the introduction is Section 2 which gives a background to speech coding. Section 3 describes the ITU-T G.729 and G.729A algorithms. In Section 4, an overview of VoIP is given, followed by section 5 where the Flexible ASIC concept is described. Section 6 states the performance characteristics of Capella. Section 7 explains the software development tools used in this project. The actual implementation process is handled in Section 8. Finally, a conclusion is given in Section 9.

A reader familiar with speech coding and Flexible ASIC may directly move on to Section 8 where the implementation is described and simulation results are provided.

1.4 List of Abbreviations

Table 1: List of abbreviations

Abbreviations	
ACELP	Algebraic CELP
ACF	Autocorrelation function
ADM	Adaptive Delta Modulation
ANSI	American National Standards Institute
APC	Adaptive Predictive Coding
AR	Autoregressive
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
BER	Bit Error Rate
CCITT	Comite Consultatif International Telegraphique et Telephonique
CDM	Common Data Memory
CELP	Code-Excited Linear Prediction
CKF	Compiler Known Function
CPM	Common Program Memory
CU	Computational Unit
DAM	Diagnostic Acceptability Measure
DSP	Digital Signal Processor
DFT	Discrete Fourier Transform
DRT	Diagnostic Rhyme Test
ETSI	European Telecommunications Standards Institute
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FLACC	Flexible ASIC C Compiler
FS	Federal Standard
GSM	Group Special Mobile
IETF	Internet Engineering Task Force
IIR	Infinite Impulse Response
IIS	Internet Integrated Service
ISD	Instruction Set Descriptions
ISPP	Interleaved Single-Pulse Permutation
IP	Internet Protocol
ITU	International Telecommunication Union
LDM	Local Data Memory
LP	Linear Prediction
LPC	Linear Predictive Coding
LSF	Line Spectral Frequencies
LSP	Line-Spectral Pairs
LTP	Long-Term Prediction
MELP	Mixed Excitation Linear Prediction
MGCP	Media Gateway Control Protocol
MIPS	Millions of Instructions Per Second

continued on next page

Table 1: *continued*

Abbreviations	
MOS	Mean Opinion Score
MPE	Multi-Pulse Excitation
MP-MLQ	Multipulse-Maximum Likelihood Quantization
MSE	Mean Square Error
NA	North America
PCB * Printed Circuit Board	
PPDPCM	Pitch-Predictive Differential Pulse Code Modulation
POTS	Plain Old Telephone Service
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RAM	Random Access Memory
RTCP	Real-Time Control Protocol
RTP	Real-Time Transport Protocol
RPE	Regular-Pulse Excitation
RSVP	Resource Reservation Protocol
SIP	Session Initial Protocol
SS7	Signaling System
TCP	Transport Control Protocol
TIA	Telecommunications Industy Association
UDP	User Datagram Protocol
VLB	Very Low Bit-Rate
VQ	Vecor Quantizer
VSELP	Vector Sum Excited Linear Prediction
3GPP	Third Generation Partnership Project
Vocoder	Voice Coder

2 Speech Coding Overview

2.1 Introduction

The purpose of speech coding is to represent a speech signal with as few bits per second as possible. Thus, speech coding may also be referred to as speech compression. Like many other technical matters, speech coding research has its roots in military applications. In the 1950's, the U.S. Army started developing a voice coder (vocoder) for speech transmission over highly distorted analog lines. The purpose was to decrease the transmission bandwidth needed. In addition, it also was a way of enabling secure voice communication. One of the first vocoders was the "US Federal Standard 1015", developed by the U.S. Army in 1976. This vocoder is also known as the LPC-10.

During the same time, the international standardisation organ Comite Consultatif International Telegraphique et Telephonique (CCITT) developed the G.711 standard for digital telephony systems. CCITT was later replaced by ITU-T which has ever since continued developing speech coding standards [14].

Today speech coding is widely used in digital communications. One example of an application that uses advanced speech coding is the cell-phone.

In order to understand the techniques used in speech coding, basic knowledge of the human speech production system is helpful.

2.2 Speech Signal Properties

2.2.1 Human Speech Production

Acoustically, speech is a fluctuation of air pressure. In the human speech production system the lungs provide the air flow which is then transformed into speech by the vocal tract. The air flow is forced through the *glottis* which is the space between the vocal cords, through the *larynx* to the cavities of the vocal tract. Leaving the oral and nasal cavities, the air flow exits through the mouth and nose. The main vocal organs are depicted in Figure 2 [2].

The lung pressure in the larynx pushes the air flow through the vocal cords, which in turn start vibrating, through which a periodic pressure wave is produced. The *fundamental frequency*, f_0, is an essential characteristic of periodic signals. It is controlled by the variation of the tension in the vocal cords. For men the average fundamental frequency is about 110Hz, and for women and children it is 200 and 300Hz respectively. *Pitch* is defined as the perceived frequency of sound. It is not a quantity that can be measured directly. Although pitch and fundamental frequency represent different concepts, the two are often referred to as being the same.

Sounds which are generated through the vibration of the vocal cords are known as *voiced sounds*. Examples of voiced sounds are /u/,/d/,/w/,/i/ and /e/. By pressing one's fingers on both sides of the upper throat, one can feel the vibrations of the vocal cords when pronouncing voiced sounds. On the contrary, *unvoiced sounds* are generated when the vocal cords are completely open. Unvoiced sounds are, for instance, /s/ and /f/.

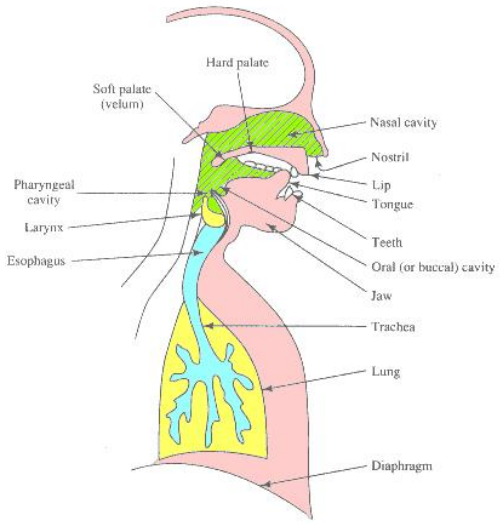

Figure 2: Human speech production.

2.2.2 Speech Sounds

One way of analyzing speech is to use *phonetics*, which is the study of speech sounds and their production. Every language has limited linguistic units called *phonemes*. A phoneme is a sound and several consecutive phonemes create words. For example, the word dog consists of the three phonemes *d*/*ao*/*g*. Most languages have 20-50 phonemes, each phoneme representing a sound. The 42 phonemes in the English language are listed in Table 2 [15].

A *phone* is an acoustic realization of a phoneme. If the realization of a phoneme is context dependent, it is called an *Allophone*. In most languages, the phonemes can be divided into two groups: vowels and consonants. The set of vowels and consonants is language specific. The hierarchy of the English phonemes is depicted in Figure 3 [25].

2.2.3 Speech-Signal Waveform Characteristics

The range of frequencies that humans are able to hear is called the *audio spectrum*. The bandwidth in the audio spectrum ranges hereby from 20 Hz to 20 kHz, although most humans have a much narrower bandwidth for hearing, especially with increasing age. Frequencies above the human hearing limit are called ultrasonic, while sounds below the limit are called infrasonic.

Speech-signal waveform characteristics are constant for short time-periods. A speech signal is commonly assumed to be wide sense stationary and ergodic in the autocorrelation for segments of 10-30 ms of speech. In speech analysis, it is therefore common to extract short-time segments of speech, called *frames* to enable simple speech modeling. For smoother transaction between analysis frames, the latter are extracted with overlap, which

Table 2: English phonemes

Phonemes	Word Examples	Description
iy	f**ee**l,**e**ve,m**e**	front close unrounded
ih	f**i**ll,h**i**t,l**i**d	front close unrounded
ae	**a**t,c**a**rry,g**a**s	front close unrounded
aa	f**a**ther,**ah**,c**a**r	back open unrounded
ah	c**u**t,b**u**d,**u**p	open-mid back unrounded
ao	d**o**g,l**aw**n,c**au**ght	open mid-back round
ay	t**ie**,**i**ce,b**i**te	diphtong with quality: aa + ih
ax	ag**o**,c**o**mply	central close mid
ey	**a**te,d**ay**,t**a**pe	front close-mid unrounded
eh	p**e**t,b**e**rry,t**e**n	front open-mid unrounded
er	t**ur**n,f**ur**,met**er**	central open-mid unrounded
ow	g**o**,**o**wn,t**o**ne	back close-mid rounded
aw	f**ou**l,h**ow**,**our**	diphtong with quality: aa + uh
oy	t**o**y,c**oi**n,**oi**l	diphtong with quality: ao + ih
uh	b**oo**k,p**u**ll,g**oo**d	back close-mid unrounded
uw	t**oo**l,cr**ew**,m**oo**	back close round
b	**b**ig,a**b**le,ta**b**	voiced bilavial plosive
p	**p**ut,o**p**en,ta**p**	voiceless bilavial plosive
d	**d**ig,i**d**ea,wa**d**	voiced alveolar plosive
t	**t**alk,sa**t**	voiceless alveolar plosive
t	me**t**er	alveolar flap
g	**g**ut,an**g**le,ta**g**	voiced velar plosive
k	**c**ut,**k**en,ta**k**e	voiceless velar plosive
f	**f**ork,a**f**ter,i**f**	voiceless labiodental fricative
v	**v**at,o**v**er,ha**v**e	voiced labiodental fricative
s	**s**it,ca**s**t,to**ss**	voiceless alveolar fricative
z	**z**ap,la**z**y,ha**z**e	voiced alveolar fricative
th	**th**in,no**th**ing,tru**th**	voiceless dental fricative
dh	**th**en,fa**th**er,scy**th**e	voiced dental fricative
sh	**sh**e,cu**sh**ion,wa**sh**	voiceless postalveolar fricative
zh	**g**enre,a**z**ure	voiced postalveolar fricative
l	**l**id	alveolar lateral approximant
l	e**l**bow,sai**l**	velar lateral approximant
r	**r**ed,pa**r**t,fa**r**	retroflex approximant
y	**y**acht,**y**ard	palatal sonorant glide
w	**w**ith,a**w**ay	labiovelar sonorant glide
hh	**h**elp,a**h**ead,**h**otel	voiceless glottal fricative
m	**m**at,a**m**id,ai**m**	bilabial nasal
n	**n**o,e**n**d,pa**n**	alveolar nasal
ng	si**ng**,a**ng**er	velar nasal
ch	**ch**in,ar**ch**er,mar**ch**e	voiceless alveolar affricate: t + sh
jh	**j**oy,a**g**ile,ed**g**e	voiced alveolar affricative: d + zh

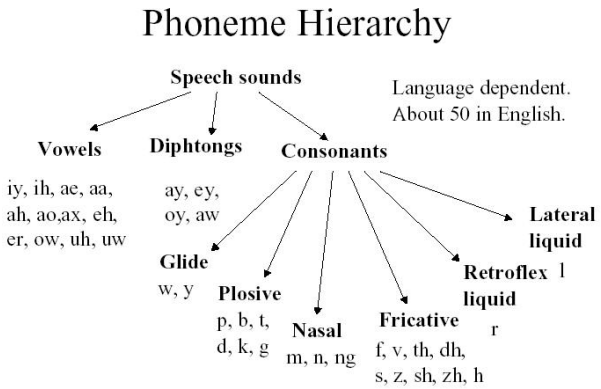

Figure 3: Phoneme hierarchy

is illustrated in Figure 4 [25]. The extracted frames are usually windowed with a rectangular, Hanning or Hamming window before further analysis is conducted.

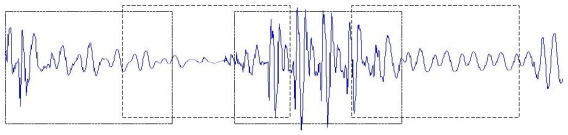

Figure 4: Overlapping frames illustration

Figure 5 shows the waveform and spectogram of the word ”Ericsson” articulated by this thesis' author. The spectogram is a classical tool for speech analysis. It consists of DFTs of overlapping and windowed frames, and shows the distribution of energy in frequency and time. In the spectogram of Figure 5, interesting characteristics of voiced sounds can be observed. The maxima of the voiced segments, seen as white stripes, are due to resonant frequencies of the vocal tract, and are called *formants*. The formant frequencies depend on the shape and dimensions of the vocal tract; each shape is characterized by a set of formant frequencies. Different sounds are produced through varying shapes of the vocal tract. Thus, the spectral properties of the speech signal vary over time with the variation of the shape of the vocal tract [24]. The unvoiced regions in the spectogram are more solidly filled because the energy of unvoiced sound is spread more evenly over the audio spectrum.

The periodicity of voiced sounds can also be seen from the *power spectrum*, taken from a short speech segment. Figure 6 depicts the voiced sound /ih/ and the unvoiced sound /s/, with both, waveform and spectrum plots [25]. The time-domain periodicity of the phoneme /ih/, for instance, can be observed in the spectrum plot as spectral peaks. The spectral peaks are harmonics of the fundamental frequency. The spectrum plots also contain the 14th-order Linear Prediction (LP) *spectral envelope* displayed with a (red) dashed line. Linear prediction in speech coding is discussed in more detail later in this section.

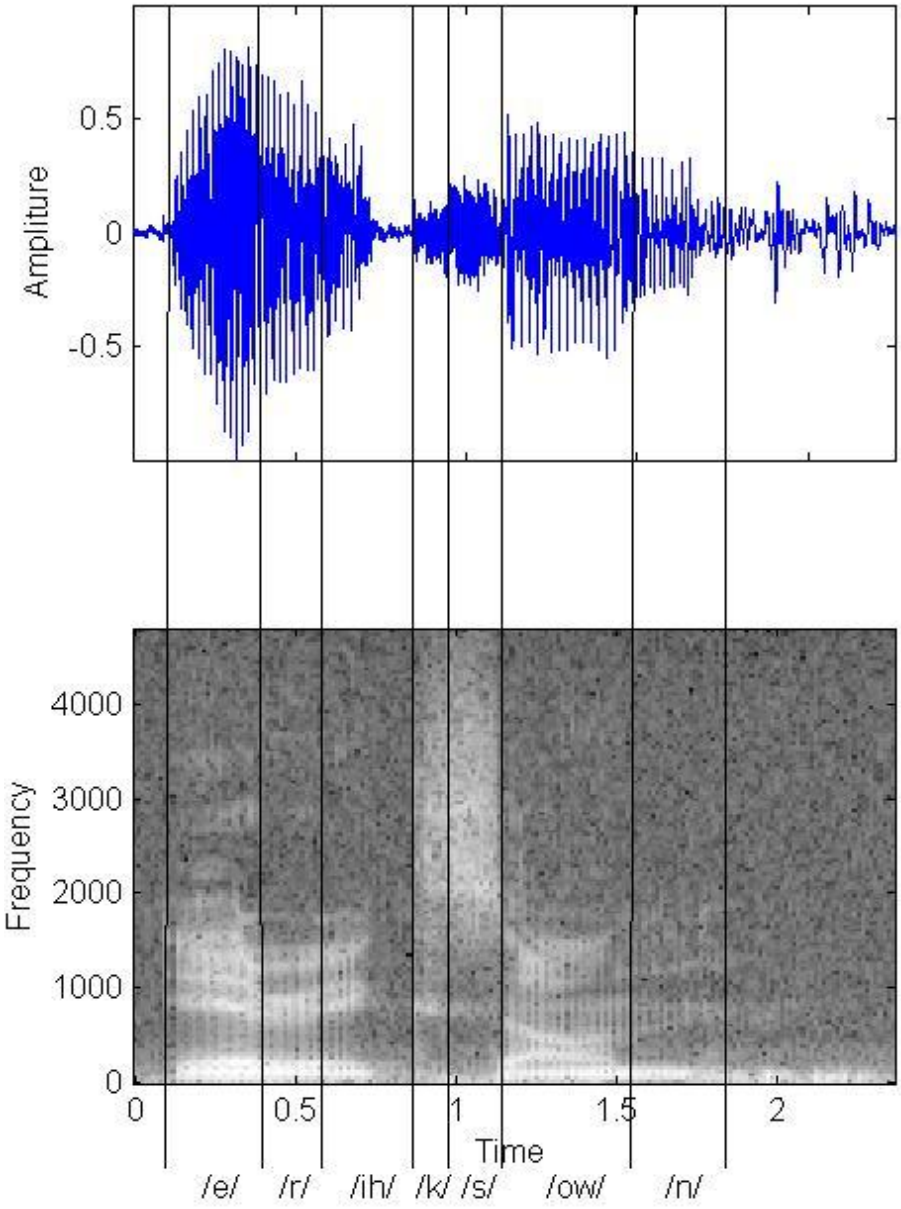

Figure 5: The word ”Ericsson” with waveform in the upper plot and spectogram in the lower

In the spectral envelope for voiced speech, the formants can be identified as peaks. The phoneme /ih/ has formants at approximately 300, 2300 and 3000 Hz. The formant's bandwidth describes the width of the formants peaks. Formants and their bandwidths are the most important spectral features. They characterize the sounds of a speaker. In

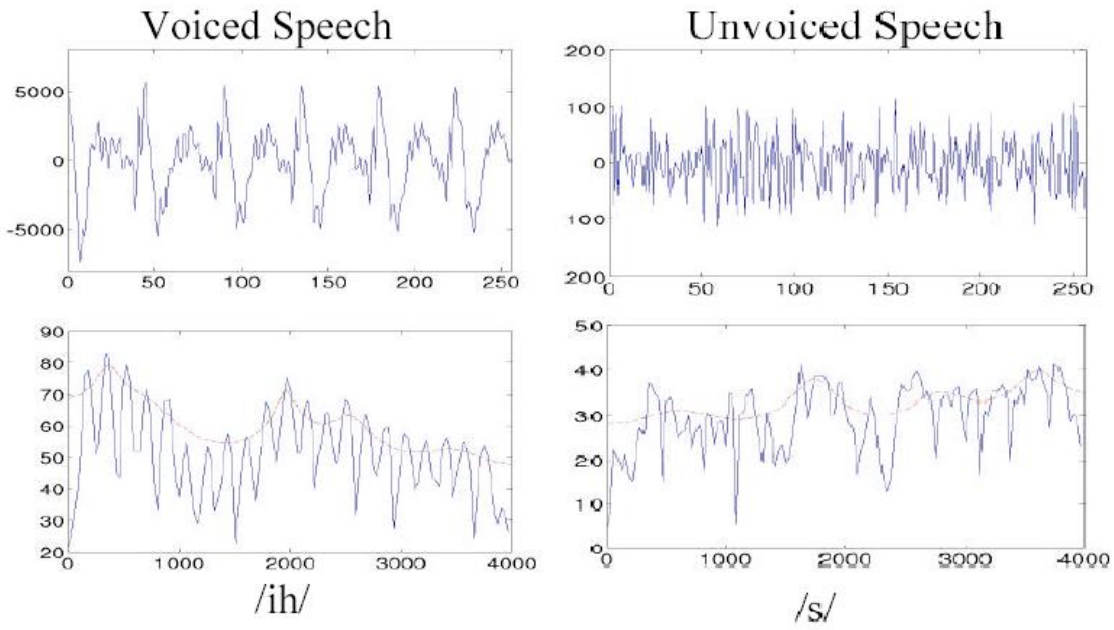

Figure 6: Waveforms and spectrums, for the voiced sound /ih/ and the unvoiced /s/. The dashed line in the spectrum shows the LPC envelope.

statistical terms, a speech signal may be classified as a non-stationary random process. It is commonly assumed that a speech signal is wide sense stationary and ergodic in autocorrelation for short-time periods from 10 to 30 ms.

The autocorrelation for a discrete-time deterministic signal is given by

$$R_{xx} = \sum_{m=-\infty}^{\infty} x[m]x[m+k] \tag{1}$$

The Discrete Fourier Transform (DFT) of the autocorrelation function yields the power spectrum of the signal, according to the equation

$$S_{xx} \triangleq \sum_{n=-\infty}^{\infty} R_{xx}(n)e^{-j\omega n} \tag{2}$$

This shows that the short-term autocorrelation for a frame is a function of the power spectral envelope, which in turn is directly related to the shape of the vocal tract. Thus, the short-term autocorrelation is a function of the shape of the vocal tract.

2.3 Speech Analysis

The speech-analysis techniques reviewed at this point, play an important role in many speech coders, including the G.729.

2.3.1 Pitch Estimation

The single, most important characteristic of voiced speech is the *pitch*. The goal of pitch estimation is to retrieve the pitch, the perceived fundamental frequency, from a voiced segment of speech. There are two main approaches for pitch estimation. The first approach is the autocorrelation method which makes use of the voiced-speech periodicity in the time domain. The second approach is to analyze the fundamental frequency's harmonics in frequency domain. Pitch estimation is a difficult task and to reduce errors, pre-processing of the signal is essential. The pre-processing intends to remove noise through filtering, thereby simplifying the signal for pitch analysis.

Autocorrelation Method

With the autocorrelation method the pitch is estimated by detecting the maximum value of the autocorrelation function in the region of interest [15]. The statistical autocorrelation of a sinusoidal random process

$$x[n] = \cos(\omega_0 n + \phi) \tag{3}$$

is given by

$$R[m] = E[x[n]x[n+m]] = \frac{1}{2}\cos(\omega_0 m) \tag{4}$$

This has maxima for $m = lT_0$, the pitch period and its harmonics. Thus, the pitch period can be estimated by finding the maximum value of the autocorrelation. A wide-sense stationary periodic process, which voiced speech is assumed to be, has an autocorrelation which also has its maxima at $m = lT_0$. Figure 7 depicts the Auto Correlation Function (ACF) for the voiced phoneme /ih/, which is extracted from the speech-signal "Ericsson", articulated earlier. The maximum amplitude of the ACF is at 208 samples and with sampling frequency of 22,050 Hz, this gives a pitch of 110 Hz ($pitch = f_s/samples$). When finding the maximum of the ACF, the first peak with the absolute maximum at $m = 0$ is excluded.

Pitch estimation is a difficult task and involves several error factors.

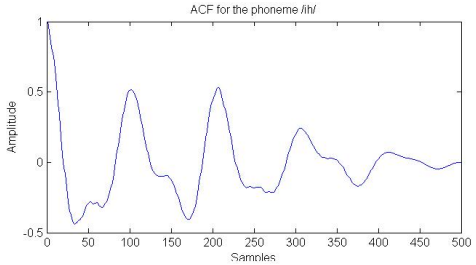

Figure 7: ACF for the voiced phoneme /ih/

The main error factors in pitch estimation are:

- **Sub-Harmonic Errors**
  Sub-harmonics of the fundamental period T_0 appear at $2T_0$,$3T_0$..., and can wrongly be identified as the fundamental period.
- **Noisy Conditions**
  For noisy conditions, with low SNR, pitch estimation is unreliable.
- **Vocal Fry**
  For some speakers the pitch is not continuous and it may change drastically, even halve [15].

2.3.2 Source-Filter Model

The *source-filter model* is based on the human speech production system and is the most commonly used model for speech synthesis. With the information of the pitch period and vocal tract parameters, speech can be synthesized to replicate naturally spoken speech. In the source-filter model, depicted in Figure 8 (according to [24]), a speech signal is sep-

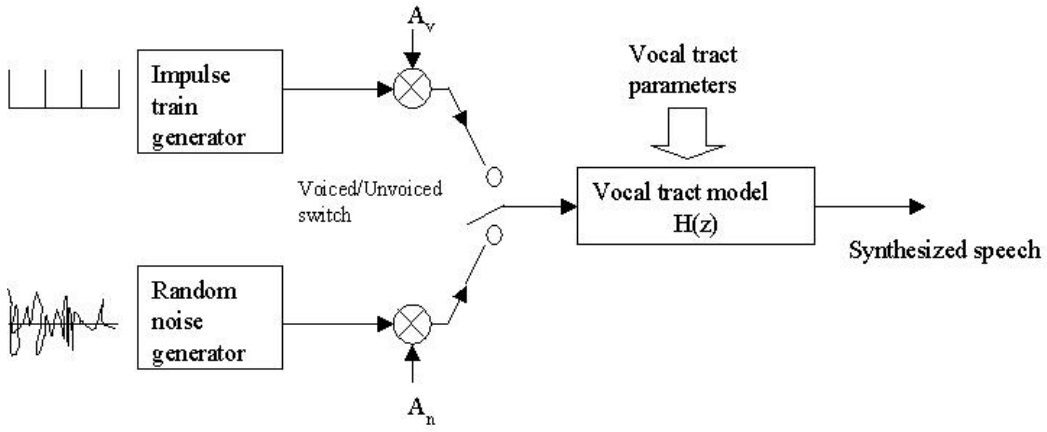

Figure 8: Source-filter model of speech production

arated into two components: the *excitation* and the *vocal tract parameters*. For voiced speech, the excitation is an impulse train with periods corresponding to the fundamental frequency (or pitch). For unvoiced speech, the excitation consists of white noise. For certain sounds, a mixture of voiced and unvoiced excitation is required. This is achieved by scaling and adding the pulse train to the white-noise excitation. Typically, this is required for segments which contain a transition between voiced and unvoiced speech. Also, some phonemes are both, voiced and unvoiced. The voiced/unvoiced switch in Figure 8, requires information about whether the current speech is voiced or unvoiced. Designing a voiced/unvoiced detector algorithm can be even more difficult than designing the pitch estimator. Two possible approaches for designing voiced/unvoiced detector are listed below:

- **Zero-Crossing Count**
  The zero-crossing count is an indicator of the frequency at which the energy is concentrated in the signal spectrum. Voiced speech is periodic and usually shows a low zero-crossing count. Unvoiced speech on the contrary shows a high zero-crossing count. The zero-crossing count of silence is expected to be lower than for unvoiced speech, but comparable to that of voiced speech.

- **Signal Energy Comparison**
  For voiced data the energy is much higher than for silence. The energy of unvoiced data is usually lower than for voiced sounds but higher than for silence.

The *vocal tract model* of Figure 8 is a filter representing the vocal tract and the shape of the lips. The filter parameters can be estimated by linear prediction which is discussed in the following section. The source-filter model assumes that speech can linearly be separated into the excitation signal and vocal tract response. With the z-transform of the excitation signal denoted as $E(z)$ and the vocal tract model as $H(z)$, the z-transform of the synthesized signals is given by

$$S(z) = E(z)H(z) \tag{5}$$

The source-filter model is useful in speech coding since it can synthesize speech with the knowledge of only a handful of parameters.

2.3.3 Linear Predictive Coding

Linear Predictive Coding (LPC) is a well-suited perceptual match between a mathematical expression and the human speech characteristics [13]. LPC has proven to be a fast and simple way to estimate the main parameters of speech signals and thereby found widespread use in speech signal-processing applications.

A linear predictor, with the order M, that predicts the next sample of a stochastic process $x(n)$ is give by

$$\hat{x}(n) = \sum_{k=1}^{M} a_k x(n-k) \tag{6}$$

This is called a one-step predictor. a_k are called the *predictor coefficients*. The prediction error is then defined by

$$e(n) = x(n) - \hat{x}(n) \tag{7}$$
$$e(n) = x(n) - \sum a_k x(n-k) \tag{8}$$

Since this is an auto-regressive (AR) process, LPC analysis is also called *auto-regressive* modeling and the predictor coefficients are called *AR-parameters*. The z-transform of the prediction error is given below.

$$E(z) = X(z)(1 - \sum a_k z^k) \tag{9}$$
$$E(z) = X(z)A(z) \tag{10}$$

$A(z)$ is called a *whitening filter*. The reason for this is that by filtering the speech signal $x(n)$ through $A(z)$ the error signal $e(n)$, which is close to white noise is derived. The inverse

of $A(z)$ is called the *production filter*. Filtering the error signal through the production filter yields the original signal. This is illustrated in Figure 9.

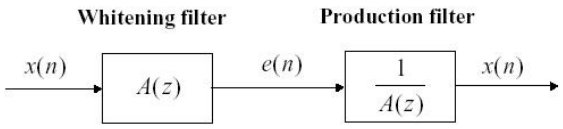

Figure 9: Production and whitening filter

In order to find the optimal predictor coefficients, the variance of the error signal has to be minimized. This is achieved by differentiating Equation 11 with respect to the predictor coefficients and setting this equal to zero.

$$\alpha^{(m)} = E[e^2(n)] \tag{11}$$

This yields Equation 12, which is known as the *Yule-Walker* equation.

$$\frac{\partial \alpha^{(m)}}{\partial a_j} = 0 \Leftrightarrow \sum_{k=1}^{M} a_k R_k(j-k) = R_X(j), \quad j = 1, .., M \tag{12}$$

If $M = 3$, the Yule-Walker in matrix form is given by

$$\begin{pmatrix} R(0) & R(1) & R(2) \\ R(1) & R(0) & R(1) \\ R(2) & R(1) & R(0) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} R(1) \\ R(2) \\ R(3) \end{pmatrix} \tag{13}$$

There exist two efficient algorithms for solving the Yule-Walker equations and finding the optimal predictor coefficients: the *autocorrelation method* and the *covariance method*. In practice, the autocorrelation method is the most favored since it returns poles with values inside the unit circle. Thus, it guarantees that the production filter is stable. The autocorrelation method uses the computationally fast *Levinson-Durbin* Algorithm. For detailed information on the autocorrelation- and covariance method, the interested reader may consult [13] or [15].

LPC Spectral Analysis

The production filter, $H(z)$, with a gain G is defined by

$$H(z) = \frac{G}{A(z)} \tag{14}$$

$H(z)$ is used as the vocal tract model in the source-filter model, discussed in the previous section. By plotting the spectrum for $H(z)$, peaks can be observed that correspond to roots of the denominator. The 14th-order LPC spectrum of the phoneme /ih/, plotted in red, is depicted in Figure 6. From the figure, it can be seen how well the envelope matches the original signal. This is the key to success in LPC.

The choice of prediction order is a balance between spectral detail and estimation error. Also, the prediction order depends on the sampling frequency, f_s. On average, there

is one formant per kHz of bandwidth which needs to be modeled by corresponding poles. The poles are conjugated pairs, thus two predictor coefficients per kHz of bandwidth are required. For improving flexibility of the LPC, two to four extra coefficients are added.

In speech coding it is popular to use *Line Spectral Frequencies* (LSF), which is an equivalent representation of the predictor coefficients. The latter can be seen as a transformation of the predictor coefficients. The LSFs have sensitivity advantages in quantization.

2.4 Speech Coding Performance Attributes

The goal of a speech coder is to represent a speech signal in digital form with as few bits as possible and with as high speech quality as possible. Since these attributes are however conflicting, a speech coder is chosen by application requirements. The main attributes concerning a speech coder are:

- **High Speech Quality**
  Naturally, achieving high speech quality is essential for a speech coder. Measuring speech quality is difficult since hearing perception differs between people. High speech quality in the ears of one person, might be perceived as low quality for another. The most widely used subjective test for speech quality is the *Mean Opinion Score (MOS)*. The MOS has a rating scale from one to five: (5) excellent, (4) good, (3) fair, (2) poor, (1) bad. An MOS-rating of four is defined as *toll* quality, in which case the reconstructed speech cannot be distinguished from the original speech. There also exist objective speech quality performance measurement techniques like, for instance, the Diagnostic Rhyme Test (DRT) and Diagnostic Acceptability Measure, see [13]. The DRT is the most popular of these two. A good speech coder has a DRT rating in the range 85-90.

- **Low Bit Rate**
  As mentioned, the primary objective of a speech coder is to reduce the bit-rate needed for speech representation. Today, bit-rates of speech coders range from the 64-kbit/s G.711 to 2.4 kbit/s for the state of the art *New low-rate U.S. Federal Standard*. Most commonly, speech coders operate at a fixed bit-rate although some coders use variable rates such as the *3G AMR* speech coder.

- **Small Delay**
  The delay of a speech coder consists of four parts. The first one is the algorithm delay which is the accumulation of a frame plus possible look ahead. Second is the transmission delay. The third delay occurs in multiplexing, in the case of a multiple user channel. The fourth and final delay is the computational delay required by the DSP to run the coding and decoding algorithms.

  The acceptable limit of delay is application dependent. In a system without an echo-canceler, the delay threshold can be as low as 10 ms. It is irritating and difficult to speak when hearing one's own voice. With echo cancelation the acceptable limit of delay depends on the interaction level of the conversation. For a normal conversation, a delay of 500 ms may be tolerable, while for a highly interactive conversation the delay threshold may be 150 ms [28].

- **Robustness Against Channel Errors**
  There are two types of channel errors. The first one is random errors which occur as a consequence of channel noise. This is specified as Bit Error Rate (BER). The counter-measure against random errors is channel coding. In channel coding, redundancies are added to the signal to make it more robust against channel errors. The second type of channel errors are called burst errors. These errors are common in radio channels and occur due to causes like, for instance, fading. In order to fight burst errors, detection schemes are implemented.

2.5 Speech Coding Techniques

Figure 10 demonstrates one way of classifying speech coders according to coding method and domain [13]. In the figure, the bit-rate increases to the right, i.e. vocoders have the lowest bit-rate and waveform coders the highest. The downward pointing arrow on the left corresponds to coding in time-domain while the arrow on the right points to coding techniques in frequency domain.

In the following text the four different speech coder types are explained. Last in this section, a table of speech-coding standards and their characteristics is provided.

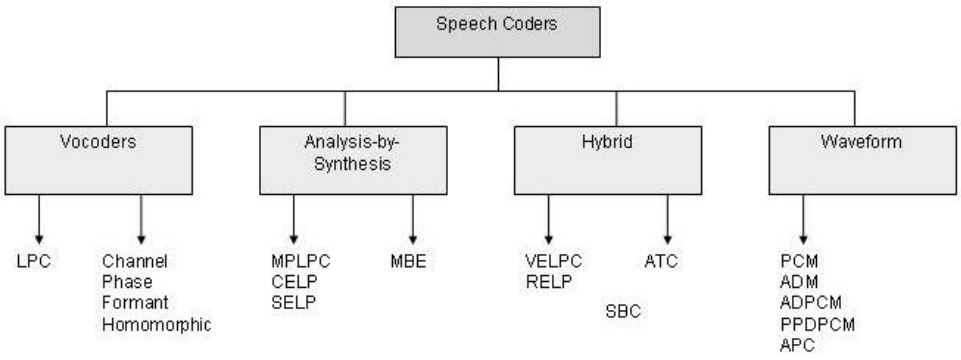

Figure 10: Speech coding techniques

2.5.1 Waveform Coders

The world's first speech coding standard, the G.711 64kbit/s PCM, is a waveform coder. Waveform coders faithfully attempt to preserve the time domain waveform. Thus, waveform coders performs as well on non-speech signals.

The waveform coders are the best coders with regard to speech quality, however, with the downside of a high bit-rate. The simplest waveform coding technique is PCM. More advanced techniques are used in Adaptive Delta Modulation (ADM), Adaptive Differential Pulse Code Modulation (ADPCDM), Pitch-Predictive Differential Pulse Code Modulation (PPDPCM) and Adaptive Predictive Coding (APC). For further information on waveform coders the reader may, for example, consult [13].

2.5.2 Voice Coders

Voice coders, or *vocoders*, were the first speech coders to allow significant compression to low bit-rates. These coders imitate, or model, the human speech production system with emphasis on the vocal tract.

Vocoders are based on the source-filter model and LPC analysis discussed earlier. In the encoder, *analysis* is conducted to extract the vocal tract parameters and the period of the excitation signal. These data are then quantized and sent over the transmission channel. With the received parameters of the vocal tract filter and the excitation signal period; the decoder performs *synthesis* of the speech by filtering an excitation signal, with the specified period, through the vocal tract filter. The speech produced by simple vocoders has a tendency to sound unnatural and robotic.

2.5.3 Hybrid Coders

Hybrid coders are a combination of waveform coders and vocoders. The speech production model is similar to the one used by vocoders but the excitation signal differs. Hybrid coders use a mid-range bit-rate and produce well-sounding synthesized speech.

2.5.4 Analysis-by-Synthesis Coders

Analysis-by-Synthesis coders can be seen as an improved form of vocoders. In the encoder, analysis-by-synthesis coders synthesize all possibilities with a given codebook structure, and find the best *perceptual* match to the original speech by comparing each synthesized signal to the original one with a perceptually weighted least mean square comparison. The parameters representing the best excitation signal and corresponding production filters, are then send over to the decoder. An analysis-by-synthesis encoder can essentially be seen as an encoder with a built-in decoder, where all possible signals are synthesized and the parameters of the synthesized signal which sounds the most like the original speech, is then transmitted to the decoder.

The reason why analysis-by-synthesis coders are so successful, and in many cases better than other coders, is because of the perceptual distortion measure used to find the best excitation signal. The design of the perceptual weighting filter, which is supposed to resemble human hearing, is critical for the success of the analysis-by-synthesis coder. [13].

2.5.5 Code Excited Linear Predictive Coding

CELP-coding, which is used in the G.729, is one of the most popular coding techniques for low bit-rate speech coders (for bit-rates below 10 kbit/s). The CELP-algorithm is a version of the analysis-by-synthesis technique described earlier. The new concept of CELP is that the excitation signal is taken from a *codebook* with stored excitation vectors. Therefore, the name code-excited linear prediction. Also, another major difference between a CELP-coder and the LPC-coder, discussed earlier, is that the CELP-coder does not need to classify voiced or unvoiced speech.

In CELP-coding, a speech signal is synthesized by a *long-* and a *short-term predictor*. The long-term predictor accounts for fine structures in the spectrum, i.e. the pitch. In

CELP-coding, the long-term predictor is also called an *adaptive codebook*. The adaptive-codebook entries are segments of the recently synthesized signal [15]. The short-term predictor accounts for the spectral envelope, i.e. the formants, and is the same type of production filter used in vocoders. Figure 11 depicts how speech is synthesized in a CELP-encoder [27]. Thus, the fixed innovation comes from the fixed codebook which is then filtered through the long-term production filter followed by the short-term production filter. Another way to see this is that the excitation vectors from the adaptive and fixed codebook are added to construct the excitation signal which is then filtered through the short-term production filter.

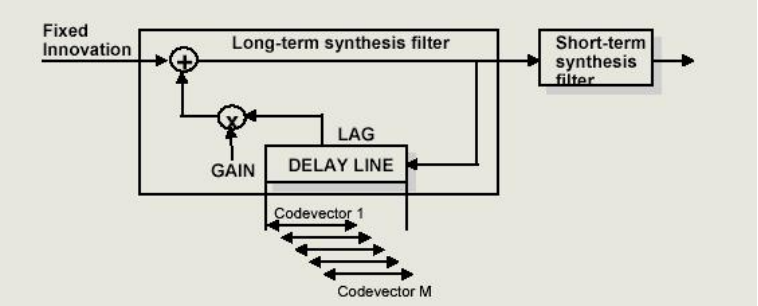

Figure 11: Long-term prediction / Adaptive codebook

By subtracting the long-term and short-term predictions of the speech signal, the redundancy of the signal is removed. The remaining signal is called the *innovation* or *excitation* signal. A set of signals which approximates the possible excitation sequences forms the codebook. Thus, a CELP- encoder algorithm consist of three stages:

1. Finding the best long-term and short-term linear predictors.
2. Filtering the predictor filters with all possible excitations and finding the sequence in the codebook that minimizes the perceptually-weighted mean squared error. This is an analysis-by-synthesis procedure. The perceptual weighting takes into account how humans perceive sound or speech and is absolutely essential to the success of CELP-coding.
3. Send information to the decoder about the long- and short-term predictor coefficients and the binary codebook word corresponding to the sequence chosen from the codebook.

The decoder then looks up the excitation sequence from the codebook with the received codebook word and drives the excitation sequence through the long- and short-time production filters [13]. The CELP-coding principle is depicted in Figure 12 [8].

CELP Complexity

A CELP-coding algorithm is complex and therefore computationally demanding. Assuming a CELP-coder that uses a 10-bit word as an index to the codebook, would give a codebook consisting of $2^{10} = 1024$ excitation sequences. Applying each of these 1024 excitation

sequences to the long- and short-term production filters and comparing each synthesized signal with the original is the core of the encoder. Since this process needs to be performed for each speech frame it is computationally very complex. In the G.729 algorithm, this is done twice for every 10-ms frame.

A-CELP

Algebraic-CELP (A-CELP), is a special case of CELP with an efficient search of the codebook. The term algebraic means the use of algebra or mathematic rules to create the excitation codevectors. These rules are addition and shifting. With ACELP there is no need to store the entire codebook, which leads to significant memory saving [4].

In Algebraic Structure-CELP (AS-CELP), the excitation pulse amplitudes are limited to the fixed values of +1 and -1. In AS-CELP, the possible positions for each pulse are restricted, which is controlled by a track table for each pulse [27].

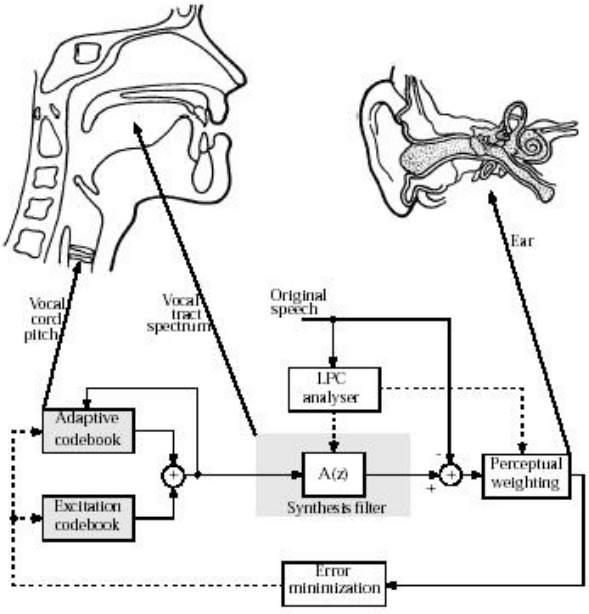

Figure 12: CELP encoder

2.5.6 Speech Coding Standards

Table 3 lists the most-commonly used speech coders and their characteristics starting from the first standard, G.711, until today's state of the art 3G-AMR speech coder.

Table 3: Speech coding standards

Year Finalized[a]	Standard	Type	Bit Rate[a] (kbit/s)	Complexity (MIPS)	Quality[c] (MOS)	Applications
1972	ITU-T G.711	PCM	64	0.01[c]	4.3	for PSTN
1984	DoD FS 1015	LPC-10	2.4	7[c]	2.3	Secure communication
1987	ETSI GSM FR 6.10	RPE-LTP	13	5-6[c]	3.5-3.9	Digital mobile radio
1990	ITU-T G.726	VBR-ADPCM	16, 24, 32, 40	2[c]	4.1	General purpose
1990	TIA IS-54	VSELP	7.95	14[c]	3.5	NA TDMA digital cellular telephony
1990	ETSI GSM HR 6.20	VSELP	5.6	14[c]	3.4	GSM cellular system
1990	RCR STD-27B	VSELP	6.7			Japanese cellular system
1991	FS 1016	CELP	4.8	16[c]	3.2	Secure communication
1992	ITU-T G.728	LD-CELP	16	30[b]	4.0	General Purpose
1993	TIA IS-96	VBR-CELP	8.5,4,2,0.8	15[c]	2.3	NA CDMA digital cellular telephony
1995	ITU-T G.723.1	MP-MLQ / ACELP	5.3/6.3,	11[c]	4.0/3.7	Multimedia communications videophones
1995	ITU-T G.729/A	CS-ACELP	8	20[c]/10.8[b]	4.0/3.8	General purpose
1996	ETSI GSM EFR	ACELP	12.2	-	-	General purpose
1996	TIA IS-641	ASELP	7.4	-	-	NA TDMA digital cellular telephony
1997	DoD FS MELP	MELP	2.4	40[c]	3.2	Secure communication
1999	ETSI AMR	AMR-ACELP	12.2,10.2,7.95, 7.40,6.70,5.90, 5.15,4.75	-	-	General purpose

[a]Based on data from [4]

[b] Based on data from Vocal Technologies LTD (www.vocal.com); Algorithms implemented on ADSP-21xx

[c]Based on data from[18]. For MIPS values, the DSP is not provided.
Although values correspond well with the ones provided by [b].

3 Description of the G.729/A Speech-Coding Algorithm

3.1 Introduction

In this thesis' work, the ITU-T G.729A speech coder standard was implemented. The G.729A coder is a low complexity version of the G.729 coder. The differences between the two algorithms are relatively small, therefore in this section the G.729 coder will be described first and at the end of this section the differences between the G.729 and the G.729A will be stated.

The following description of the G.729 coder is mainly based on the documentation of the standard [16]. The goal is to describe the coder in a simple and understandable way. A reader seeking detailed specifics may refer to the standard document [16].

3.2 General Description of the Coder

The G.729 uses a 16-bit Pulse Code Modulation (PCM) signal as input which is achieved by the following means. The analog speech signal is telephone-bandwidth filtered through a 300-3400 Hz filter and sampled at 8 kHz. This signal is then quantized to a 16-bit PCM-signal which then serves as input to the G.729 speech coder. The G.729 operates at 8 kbit/s and on 10-ms frames, corresponding to 80 samples. The characteristics of the coder were previously displayed in Table 3.

The G.729 speech coding algorithm uses Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP). For each frame of 10 ms, the input speech signal is analyzed and the CELP parameters are extracted, encoded and transmitted. The CELP parameters include LPC-filter coefficients, indexes of the adaptive and fixed codebook and their respective gains.

The decoder takes the received parameters, extracts the excitation signal from the fixed and adaptive (long-term synthesis filter) codebook and forms the short-term synthesis filter. The excitation signal is then filtered through the short-term synthesis filter to form the reconstructed speech. The last step in the decoding process is to enhance the reconstructed speech through a post-filter which emphasizes the formant structure of the signal.

3.3 Encoder

The block diagram of the G.729 encoder is shown in Figure 13 and the signal of the encoder is depicted in Figure 14 [16]. At first, the input speech signal is passed through the pre-processing block where the signal is filtered through a 140-Hz high-pass filter and scaled. On the pre-processed signal, a 10th-order LP analysis is conducted once per every frame of 10 ms. The ten LP-coefficients are then converted into Line Spectrum Pairs (LSP), which is the same as LSF, discussed in section 2.3.3, and then quantized with a two-stage vector quantizer with 18 bits.

Every frame of 10 ms is divided into two subframes of 5 ms (40 samples) each, with the purpose of reducing the complexity of the codebook searches and optimizing the tracking of pitch and gain parameters. For the first subframe interpolated LP-coefficients are used in the synthesis filter. For the second subframe the synthesis filter consists of original and quantized LP-coefficients.

Then, for each subframe analysis-by-synthesis is performed to retrieve the CELP-parameters for the best synthesized signal.

In Figure 13, the excitation signal consists of contributions from both, the adaptive and fixed codebook. The adaptive codebook then simulates the pitch, and therefore voiced sounds. The fixed codebook simulates unvoiced sounds. In reality, the adaptive codebook is an adaptive filter through which the fixed excitation signal is filtered. The following two sections deal with the fixed and adaptive codebook in detail.

3.3.1 Fixed Codebook

The fixed codebook has an algebraic structure and is called *interleaved single-pulse permutation* (ISPP) design [13]. Each 40-sample fixed codebook excitation vector has only four pulses, denoted i_0, i_1, i_2 and i_3, with, as previously mentioned, two possible amplitudes: +1 and -1. The fixed codebook is displayed in Table 4. Each excitation codevector is a sum of four pulses, given by

$$c(n) = s_0\delta(n - m_0) + s_1\delta(n - m_1) + s_2\delta(n - m_2) + s_3\delta(n - m_3) \qquad n = 0, ..., 39 \quad (15)$$

where $\delta(n)$ is the unit impulse at time instant n.

Table 4: G.729 fixed codebook

Pulse	Sign	Positions
i_0	$s_0 :\overset{+}{-} 1$	m_0:0, 5, 10, 15, 20, 25, 30, 36
i_1	$s_1 :\overset{+}{-} 1$	m_1:1, 6, 11, 16, 21, 26, 31, 36
i_2	$s_2 :\overset{+}{-} 1$	m_2:2, 7, 12, 17, 22, 27, 32, 37
i_3	$s_3 :\overset{+}{-} 1$	m_3:3, 8, 13, 18, 23, 28, 33, 38 4, 9, 14, 19, 24, 29, 34, 39

The fixed codebook is searched in four nested loops, one for every pulse. However, to reduce complexity, the fourth loop is not entirely searched since a full search in the fourth loop would produce slightly higher speech quality, but also require high computational effort. Since the four excitation pulses are unable to have the same position, the search is named *conjugate-structured*.

As can be seen in Table 4, the index to the fixed codebook consists of two parts: sign and position. Each pulse requires one bit per sign, thus, a total of 4 bits is needed to encode the signs. For the first three pulses, i_0, i_1, i_2, 3 bits are needed to encode the pulse positions. For the fourth pulse, i_3, 4 bits are required. This yields a total of 13 bits to index the pulse positions. Thus, altogether $4 + 13 = 17$ bits are needed to transmit the fixed codebook index (without gain), which in fact is done once every subframe.

3.3.2 Adaptive Codebook

The adaptive codebook, which in fact is an adaptive pitch filter is given by

$$P(z) = \frac{1}{1 - \beta z^{-T}} \quad (16)$$

where T is the pitch delay in the current subframe and β is the adaptive gain. In order to achieve high speech quality a fractional pitch delay with 1/3 sample time resolution is used. The first step in the adaptive codebook search is an open loop pitch estimation, based on the perceptually weighted speech signal, which is done once per frame. Then, to find the gain and delay for the adaptive codebook, closed-loop pitch analysis is done for each subframe by searching around the pitch estimate. The pitch delay found in the search is encoded with 8 bits for the first subframe and 5 bits from the second subframe.

3.3.3 Gain Quantization

The gains of the fixed and adaptive codebook are vector quantized with 7 bits. The VQ consists of two stages, a 3-bit two-dimensional first stage followed by a 4-bit two-dimensional second stage. The VQ also uses a *conjugate structured* codebook.

3.4 Bit Allocation

Table 5 shows the bit allocation for one 10-ms, 80 sample frame. An interesting note is that 62 bits out of 80 bits in the bitstream are located to the excitation signal. This yields 6.2 kbit/s out of 8 kbit/s.

Table 5: G.729 bit allocation

Parameter	**Codeword**	**Subframe 1**	**Subframe 2**	**Total per frame**
Line Spectrum pairs	$L0, L1, L2, L3$			18
Adaptive-codebook delay	$P1, P2$	8	5	13
Pitch-delay parity	$P0$	1		1
Fixed-codebook index	$C1, C2$	13	13	26
Fixed-codebook sign	$S1, S2$	4	4	8
Codebook gains (stage 1)	$GA1, GA2$	3	3	6
Codebook gains (stage 2)	$GB1, GB2$	4	4	8
Total				80

3.5 Decoder

Figure 15 depicts the block diagram of the G.729 decoder [16]. The first step in the decoding process is to extract the parameter indices from the received bitstream. These indices are then used to look up the values of the parameters corresponding to a 10-ms frame. These parameters are

- LSP coefficients
- Two fractional pitch delays
- Two fixed codebook vectors
- Two sets of adaptiveand fixed-codebook gains

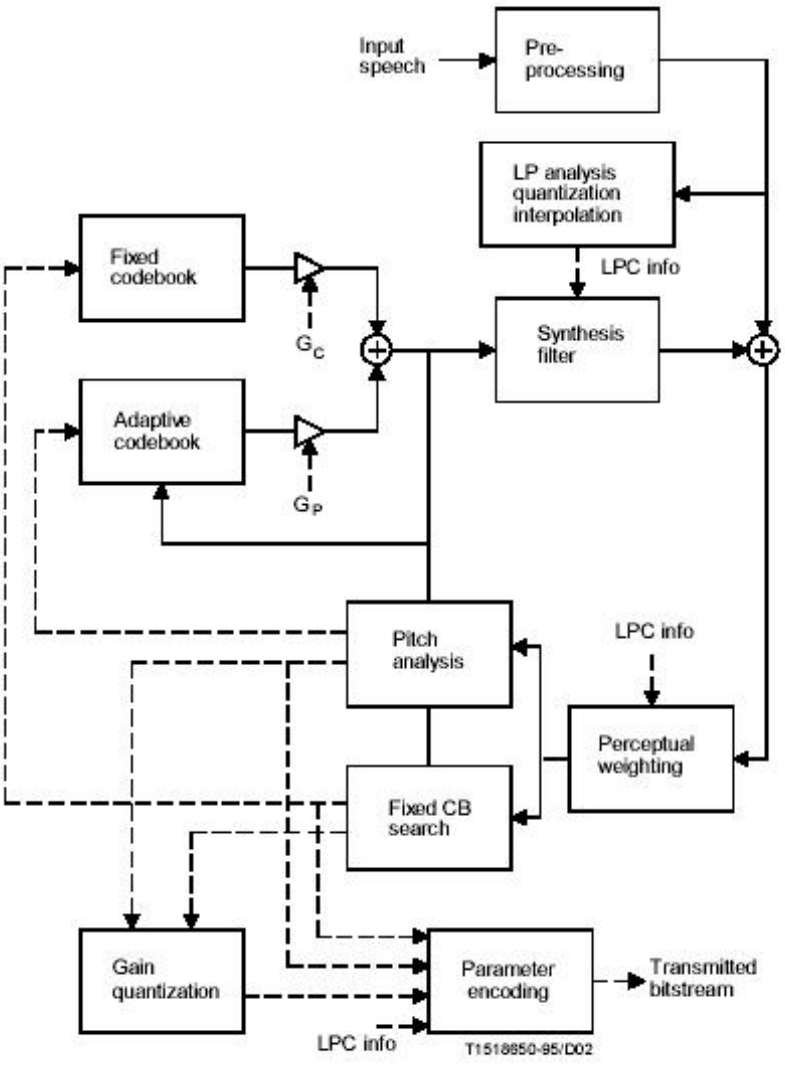

Figure 13: Encoding principle of the G.729

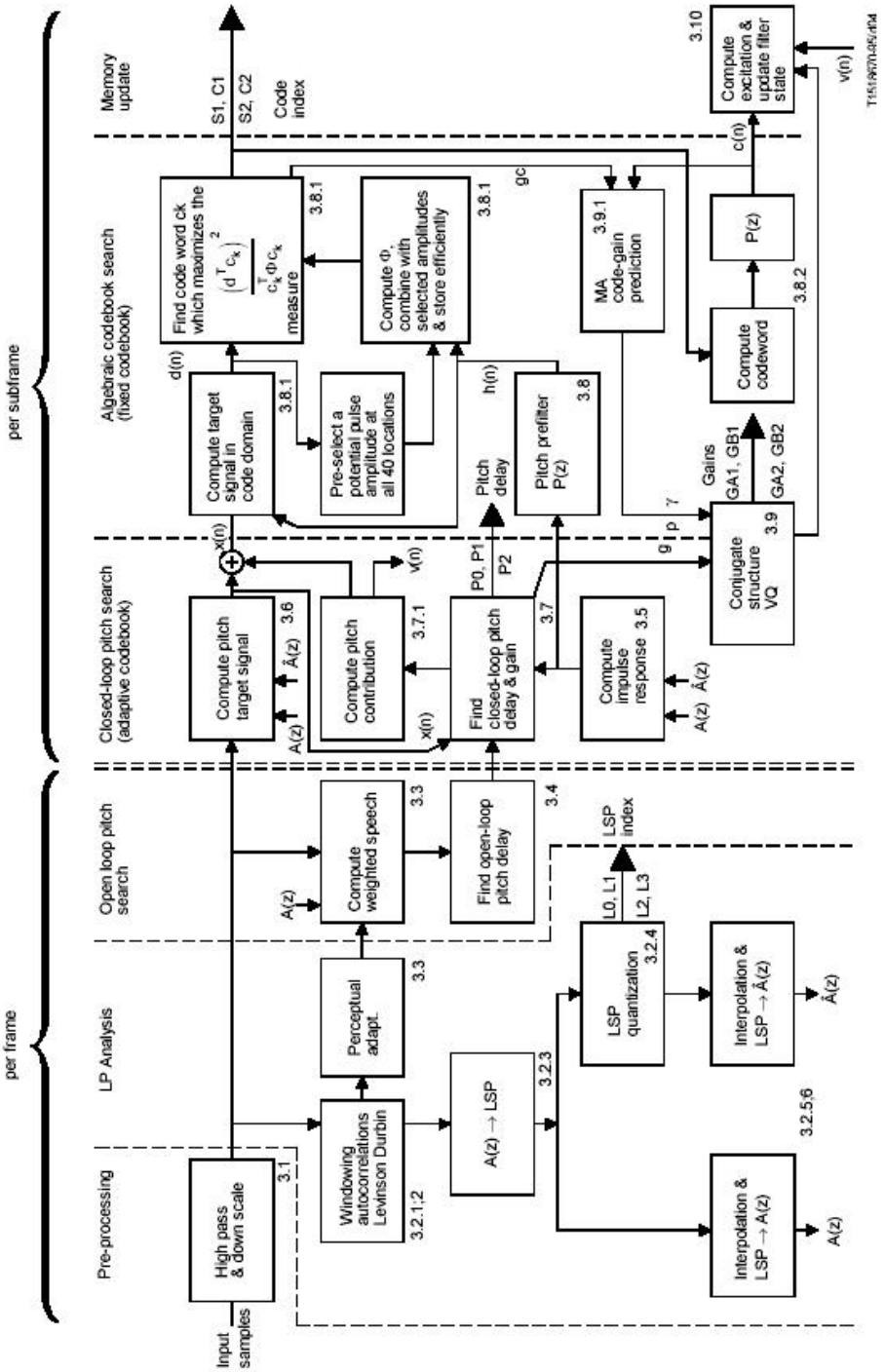

Figure 14: Signal flow of the G.729 encoder

For each subframe the LSP-coefficients are interpolated and converted to LPC-filter coefficients. Then, for each subframe, the excitation signal is constructed by adding the adaptive and fixed-codebook vectors multiplied with their respective gain. The excitation is then filtered through the LPC- (or short-term) filter. Finally the reconstructed speech is further enhanced by a post-filter.

The signal flow of the decoder is displayed in Figure 16 [16].

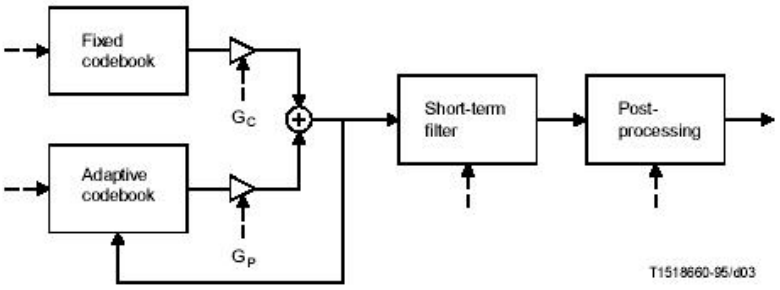

Figure 15: Decoding principle of the G.729

3.6 G.729A

The G.729A is a reduced complexity version of the G.729, i.e it requires much less processing power. The G.729A is bitstream compatible with the G.729. Thus, the speech can be encoded with an G.729 encoder and decoded with a G.729A decoder, or vice versa.

One important note though, the synthesized speech of the G.729A is not as good as from the G.729 in some cases. The G.729A has a MOS of 3.7, which is *not* toll-quality, compared to the G.729 which has a toll MOS of 4.1. Therefore, one might question the suitability of using the G.729A in a ToIP system which is intended to provide a high quality service. However, according to [1] the G.729A only suffers in performance during multiple tandem encoding (encoding and decoding the signal repeatedly) and is not a reason for anyone to not use the G.729A compared to the G.729.

The G.729A has only some small differences compared to the G.729. The differences in the encoder and decoder are stated in the following two sections. The reader interested in detailed changes in the C source code should refer to document [17] and [31].

3.6.1 Encoder Differences Between G.729 and G.729A

The modules which have been simplified are:

- Perceptual weighting filter
- LP to LSP transformation
- Search for adaptive-codebook delay

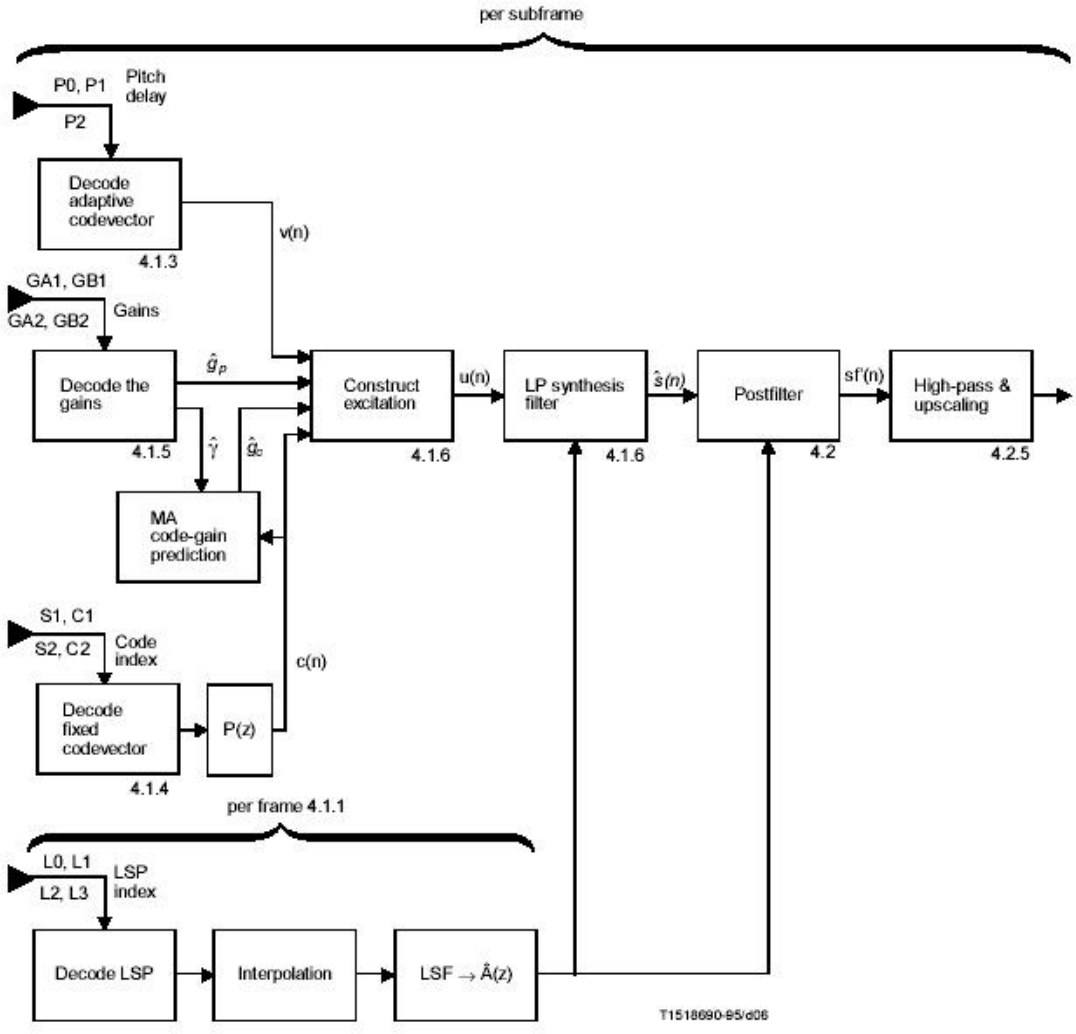

Figure 16: Signal flow of the G.729 decoder

- Fixed codebook search
- Memory updates between frames

3.6.2 Decoder Differences Between G.729 and G.729A

The decoder is almost the same for G.729 and G.729A. The differences and simplifications lay in the post-processing block, depicted in Figure 15. In the post-processing block, the following parts are simplified:

- Long-term post filter
- Short-term post filter
- Compensation filter
- Adaptive gain control

4 Voice over IP Overview

4.1 Introduction

This section shall give an overview of the concept Voice over Internet Protocol (VoIP), which is essential for understanding the broader background of this thesis project. Hereby, elements of possible VoIP solutions are examined and some of the parts considered for Ericsson's ToIP solution are pointed out.

VoIP or IP Telephony enables the transmission of voice and fax through data networks by making use of the Internet Protocol (IP). VoIP combines hereby the efficiency of a packet-switched network with the voice quality of a circuit-switched network. By transmitting voice over IP, an initial analog voice is converted into a digital data stream which is composed of data packets. After the data packets are routed through the data network between the different users, a backward conversion into voice occurs.

4.1.1 VoIP Advantages

There are a number of advantages attached to the transmission of voice over IP. Firstly, this kind of transmission enables cost savings, in that already existing networks can be used, which may especially benefit companies with a large amount of international voice traffic. Also private individuals may profit from lower rates for international VoIP calls in comparison to making calls over POTS (Plain Old Telephone Service).

The IP protocol enables the transformation of a wide range of data. In contrast to POTS, IP packets are routed from one user to another via different routes and with different delays. Voice is hereby converted into data streams in different ways, depending on the different products. While only standardized protocols should be employed, voice conversion can take place with or without compression.

4.1.2 VoIP Advantages

As will be discussed later in this chapter, voice transmission via IP is however especially perceptive to lengthy delays, packet losses and out-of-order packets, all of which may result in a substantial quality decrease of the transmitted voice. The Internet is still characterized by a high level of network instability and interrupted connections and breaking links are a frequent occurrence. While delays, which rapidly increase with increasing network traffic, have less effect on the usage of email or the transfer of files, in telephony, delays in excess of 400 ms have a substantial effect on the quality. While the use of Internet Telephony still remains impossible for most business applications, Intranet Telephony mostly overcomes the previously mentioned problems and is increasingly applied in many business environments. In general, Intranets exhibit higher transmission rates and also enable better user management [29].

4.1.3 VoIP Routing Possibilities

There are several ways to implement VoIP.

- **Partly over PSTN**
  A call can be routed partly over Public Switched Telephone Network (PSTN)-based

and partly over internet. A VoIP application can hereby communicate with a PSTN-base telephone, and vice versa. In the ToIP experimental system currently under development at Ericsson, a call is intended to go between two PSTN based telephones although the call is transmitted over a I.P. network in local station. As depicted in Figure 1.

- **Solely over the internet**
  Also a direct communication between two VoIP applications is possible.

While IP Telephony should offer the same standard of services provided by the commonly used PSTN and ISDN telephones, it may also be able to offer a wide range of new services. Examples may include universal messaging, an integrated web browser or an FTP client [29].

4.2 VoIP Components

Regarding the technology requirements of an IP Telephony solution, four stages can be identified: signaling, encoding, transport and gateway control.

4.2.1 Signal System 7

When a user dials a number, signaling is required to establish the call by firstly determining the recipient's call status - busy or available. The PSTN uses hereby the Signaling System 7 (SS7), a particular set of protocols, for call setup, teardown and maintenance. SS7 operates as a packet-switched network.

An example of an VoIP network which uses an SS7-to-IP gateway is given in Figure 17. In this case, SS7 is responsible for the call control on both sides of the PSTN and H.323 or Session Initial Protocol (SIP) controls calls in the IP network. Circuit-to-voice conversion is guaranteed through the media gateway [21].

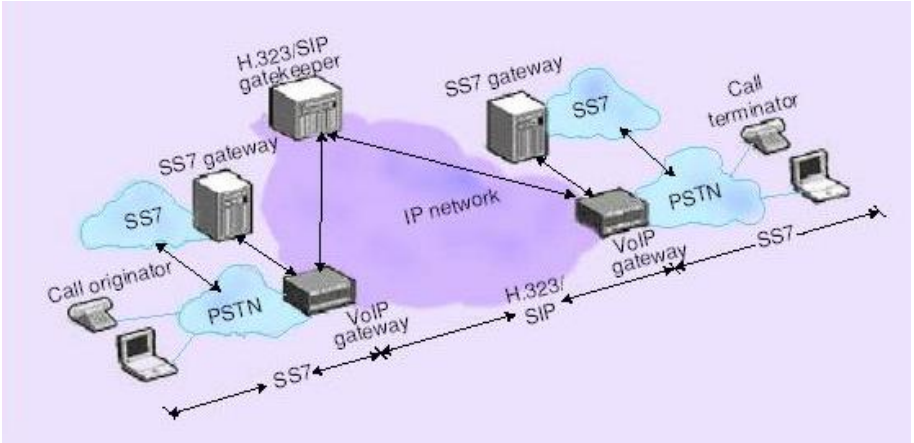

Figure 17: VoIP network with SS-7 to-IP gateway

4.3 Gateway Control

A telephony gateway is a network element that converts packet-based audio formats into protocols which are understandable to PSTN systems. Media Gateway Control Protocol (MGCP) is used for the control of telephony gateways from external call control elements known as media gateway controllers or call agents. MGCP provides a call control system where the call control intelligence is outside the gateways and handled by external call control elements. The MGCP operates under the assumption that these call control elements, call agents, synchronize with each other so that coherent commands are sent to the gateways under their control. MGCP can therefore be seen as a master/slave protocol, where the gateways execute commands sent by the call agents. The implementation of the media gateway control interface with MGCP is based on a set of transactions which is composed of a command and a subsequent mandatory response.

The Megaco protocol (H.248) is similar to the MGCP from an architectural point of view and the controller-to-gateway relationship. However, Megaco/H.248 supports a broader range of networks, such as Asynchronous Transfer Mode (ATM). In contrast to MGCP, Megaco/H.248 enables the transport of multiple commands in one single packet [21] [7].

4.3.1 H.323

Nowadays, the H.323 protocol suite, which is ratified by the International Telecommunication Union-Telecommunication (ITU-T), represents the basis of the majority of products. Figure 18 depicts an H.323 protocol including its functional parts [5]. The H.323 terminal, also known as H.323 client, is an end-user device which enables voice, video and data communication with another H.323 terminal. Examples of clients include multimedia PCs or IP phones. Also a terminal adapter, which establishes a connection between the H.323 network and an analog phone or fax machine can be a client. Address translation as well as call control services are provided by the gatekeeper. The gatekeeper is also responsible for bandwidth control, authentication, authorization as well as accounting. Switched and data networks are connected by a gateway. The connection is hereby established by converting signaling protocols and media transmission formats between the respective terminals. The gateway can also operate with other ITU-networks, including PSTN, ISDN, BISDN and ATM. In addition, the gateway can operate in form of a Multipoint Control Unit (MCU), a conference server which provides centralized signaling [29].

4.3.2 Session Initiation Protocol (SIP)

In contrast to H.323, Session Initiation Protocol (SIP), developed by the Internet Engineering Task Force (IETF), is a signaling protocol particularly designed for the Internet. In the same way as other VoIP protocols, SIP, an application-layer control protocol, is based on a packet telephony network system. SIP enables the initiation, modification and termination of interactive communication sessions between end-points. Such sessions may involve two or more users. Through mapping and redirection services, SIP provides a high degree of user mobility since telecommunication services may now be accessed from different terminals and places. Other attributes of SIP include the ability to determine the media parameters of the targeted end point as well as the latter's availability. After determining

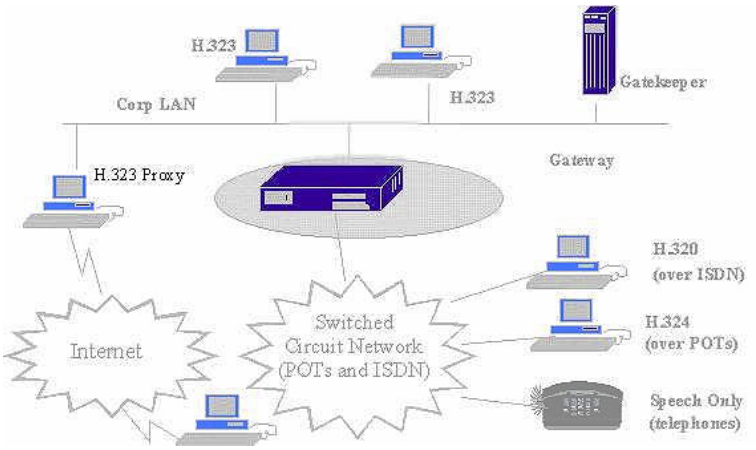

Figure 18: VoIP solution

the end point's willingness to enter into a conversation, SIP initiates a session which it later also terminates.

4.3.3 A Comparison Between H.323 and SIP

H.323 and SIP are fierce competitors in the market for IP telephony signaling. Both, SIP and H.323, have been developed to address session control and signaling functions in a distributed call control environment. Like H.323, SIP works most efficiently with intelligent end-points. However, whereas in the case of SIP network intelligence and services are provided by servers, H.323 employs gatekeepers. Also, in contrast to H.323, which operates on a binary code basis, SIP is ASCII-based.

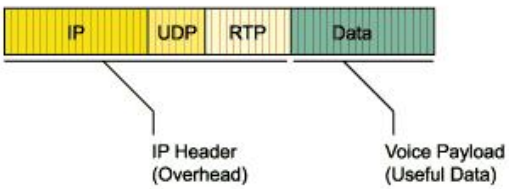

Figure 19: IP header

4.4 RTP and RTCP

Real-Time Transport Protocol (RTP) and Real-Time Control Protocol (RTCP) enable the transport of voice packets, following the voice signaling and encoding. RTP is an end-to-end delivery device providing services for data with real-time attributes, examples of which are interactive audio and video. Payload type identification, sequence numbering, time-stamping and delivery monitoring are some of the services provided. While the majority

of applications runs RTP on top of UDP to use the advantages of its multiplexing and checksum services, RTP can also be used with a number of other underlying network or transport protocols. Multicast distribution, provided by the respective underlying network, enables RTP to support the transfer of data to multiple destinations. Lower-layer service guarantees timely delivery and the quality of service. RTP does neither give a guarantee for delivery nor does it rely on the underlying network to deliver the packets in sequence. The sequence numbers included in RTP enable to determine the location of a packet, which can take place without the decoding of packets in sequence, as well as the reconstruction of the packet sequence of the sender.

RTP has primarily been developed for multimedia conferences with multiple participants. Other applications of RTP include the storage of continuous data, interactive distributed simulation, active badge, and control and measurement.

RTP contains two parts. While the Real-Time Transport Protocol (RTP) ensures the transport of data with real-time attributes, the RTP Control Protocol (RTCP) assumes responsibility for the monitoring of the service quality as well as for the provision of information about the session's participants. The basis of the RTCP is hereby the periodic transmission of control packets to all participants, which requires the multiplexing of the data and control packets by the underlying protocol, for instance, by using separate port numbers with UDP. RTCP performs four functions:

- RTCP provides feedback on the quality of the data distribution, which is closely related to the flow and congestion control functions of other transport protocols.
- RTCP carries a persistent transport-layer identifier for an RTP source called the canonical name or CNAME. Due to the possibility that the SSRC identifier may change in case a conflict is discovered or in case of the restart of a program, receivers require the CNAME to keep track of each participant. CNAME may also be required by receivers to associate multiple data streams from a given participant in a set of related RTP sessions, such as in the case of an audio-video synchronization
- Since the first two functions require that all participants send RTCP packets, there must be a control for the rate in order for RTP to scale up to a large number of participants. Through the process of having each participant send its control packets to all of the others, the number of participants can be determined. On the basis of this number, the rate at which the packets are sent, is calculated.
- RTCP conveys minimal session control information, which is mostly used in sessions with no membership control or parameter negotiation.

4.5 Quality of Service

The ability of networks to deliver good, predictable service over a variety of underlying technologies, such as IP-routed networks, is defined as Quality of Service (QoS).

Due to the real-time attribute of voice applications the latter are not only extremely intolerant to delivery delays of packets, but also to jitter, echo, congestion, packet loss, and miss-ordered packets arrival. Methods to overcome the hostile environment of the IP net and to achieve an acceptable Quality of Service, are therefore essential.

Several methods and sophisticated algorithms have been developed for the evaluation of the QoS. PSQM (ITU P.861), PAMS (BT) and PESQ are some examples [23].

The Resource Reservation Protocol (RSVP) is a protocol which allows the Internet to support QoS. RSVP, which is part of the Internet Integrated Service (IIS) model, contains the ability to reserve resources along a path from the source to the destination, which then in turn allows routers, enabled by RSVP, to schedule and prioritize packets, which guarantees the QoS.

IP-IPv6, which succeeded IPv4, inherently supports QoS. However, in contrast to IPv4, whose packet header requires a mere 20 bytes, the packet header of IPv6 necessitates 40 bytes, which doubles the overhead and may thus result in a greater latency which is especially troubling for vocoders with diminutive packets. However, header compression schemes are existent which are able to, at least partially, offset the extended header overhead.

4.5.1 Packet Loss and Jitters

If a router receives too many packets due to network congestions, some of the packets may be dropped out, known as *packet loss*. In general, packet loss has a substantial effect on the quality of the received audio/video and packet losses in excess of 10 percent are regarded as intolerable. Since voice transmissions are extremely time sensitive, regular retransmission instruments which are based on TCP appear inappropriate. Speech interpolation is one approach commonly used for packet-loss compensation. With interpolation, losses which incur during unvoiced speech segments are corrected through packet repetition, while losses during voiced speech are repaired through the use of pitch-cycle waveform repetition.

Even in the case that packets are transmitted at even intervals, they may reach the recipient with uneven intervals. Such a gap of arrival time is known as *jitter*. Jitters are generally the result of network congestions, because routers have to handle a great number of packets transmitted from many other hosts. Since jitters affect the timing of audio/video playback, also the quality of received audio and video is decreased. The removal of jitters requires the collection of packets, which subsequently have to be stored long enough so that the slowest packet will arrive and be played in the correct sequence. A jitter buffer temporarily stores arriving packets thereby minimizing delay variations. Packets that arrive too late are discarded [21].

4.5.2 Latency

Latency is an essential factor in IP performance, and reducing or mitigating the effects of latency is an important aspect of network performance engineering. Latency, which is defined as the time delay incurred in speech by the IP Telephony system, is generally measured in milliseconds from the moment that the speaker utters a sound until the listener actually hears the sound. This is known as *one-way latency*, while *round-trip latency* is the sum of the two one-way latency figures which make up a telephone call. The round-trip latency for domestic calls over the PSTN within the continental United States, is, for instance, virtually always under 150 milliseconds. At these levels, users are unable to notice the latency. By contrast, international calls, particularly the ones carried via satellite, may

have round-trip latency figures in excess of one second. According to the 1996 ITU Recommendation G.114, one-way transmission of under 150 ms are tolerable for the majority of applications, while transmissions of 150 to 400 ms are only acceptable if administrators are aware of the transmission time impact on the quality of the applications. One-way transmissions of over 400 ms, are regarded as unacceptable. Echo and talker overlap are some of the difficulties which may occur as a result of excessive delays in voice networks. While echo, where the voice of a speaker is reflected back, generally starts to be disturbing with round-trip delays in excess of 50 ms, talker overlap, where the talkers' voices overlap each other, worsens with one-way delays greater than 250 ms. Echo control, for instance, is implemented through echo cancelation [21].

4.5.3 The trade-off between bit-rate and voice quality

The majority of VoIP applications are based on an Internet connectivity where most of the users have a connection of at least 28.8 kbp/s.

There is extensive research in developing new speech coders. In particular much research effort is put into designing speech coders with extremely low bandwidths. However, some developers prioritize high-quality speech instead of achieving very low bit-rates. Thus, when designing a speech coder, a trade-off between bit-rate and voice quality is unavoidable. The G.729 speech coder studied in this thesis may be seen as an excellent example of a speech coder which has both, high-speech quality (MOS 4.1) and low bit-rate (8 kbit/s).

With the exception of low bit-rate and high speech quality, a speech coder should also have the ability to tolerate packet loss as well as miss-ordered packets. The restriction of one-way latency to one-quarter of one second and the maintenance of a buffer which restrains jitters, echo and talker overlap are also essential characteristics of a speech vocoder [21].

4.6 Frames per packet

Packet-switched technology requires an efficient mechanism for voice encoding and decoding. A voice coder converts analog speech waveforms into digital data. The coding and decoding process results in an algorithmic delay which equals frame length plus look-ahead size. However, whereas voice coders with small frame length are characterized by shorter delays and vice versa, shorter delays result in turn in a larger overhead. However, since most implementations consist of multiple frames per packet, the real frame length is determined by adding up all the frames contained in an IP packet. This means, the smaller a single frame, the greater the number of frames in a single IP packet and therefore also the smaller the resulting latency. Figure 19 depicts an IP package divided into overhead and useful data [22]. For the smallest packets, well over half of the bandwidth used, is taken up by the packet headers which is clearly an undesirable case.

If the G.729 speech coder will be a part of Ericsson's experimental ToIP system, there will be large overhead since a G.729A frame only consists of 80 bits. In a multi-channel environment, several speech frames could be put into one packet. However, then the packets should also have the same destination.

5 Overview of the Flexible ASIC Concept

5.1 Introduction

Flexible ASIC, a System-on-Chip architecture, enables the implementation of digital signal processing functions and is a concept developed for the design of small, cost efficient and high performance solutions whose essential characteristic is a high degree of reprogramming flexibility. The integration of DSP cores on a single ASIC chip forms hereby the basis of the flexible ASIC.

In order to guarantee an even higher degree of flexibility, a library of processor core building blocks, from which a customized processor core can be assembled, exists. A set of DSP software development tools, which enable the programming of the processor, are supported by the Flexible ASIC concept. The concept, which is promoted by the Ericsson Technology Board, has been used in real projects since 1996 [9].

5.2 Background

Modern digital communication systems place high demands on digital signal processing. Today, a system's manufacturing cost and power dissipation largely stem from different Digital Signal Processors (DSPs), which explains why optimizing both, the software and hardware implementation of digital signal processing, has become increasingly important. Additional aspects which have to be considered in the decision of a suitable DSP are, for instance, the DSP instruction set, clock frequency, chip size, the memory capacity of data and program memories, the respective software development tools as well as the efficiency of the C-compiler, in the case algorithms are intended to be written in C.

However, even if a DSP is clocked at a high frequency, it does not necessarily perform a particular algorithm faster compared to a DSP with a lower clock frequency. In some cases, the instruction set in the slower DSP might suit the algorithm, which is to be implemented, better. Since an increase in the performance is related with increases in the number of instructions which can be executed within a clock cycle, the parallelism of the DSP represents an essential aspect as well.

The most essential algorithms require the direct implementation in dedicated logic, as a consequence of speed requirements, in a Field Programmable Gate Array (FPGA) or an Application Specific Integrated Circuit (ASIC). Since the FPGA offers the possibility of being reprogrammed in case the functionality of the algorithm has to be changed, it guarantees a high degree of flexibility. However, the large costs and size, essentially eliminate the use of FPGAs. In contrast to the FPGA, ASIC delivers generally very good performance as well as a small circuit area. An ASIC offers however no flexibility since its functionality cannot be changed once it is manufactured [19].

5.3 ASIC and DSP

Today, the majority of projects requires the possibility to change and update the functionality of a signal processing system. However, if DSPs would be used only, performance requirements could not be met. One solution would be to combine the advantages of an ASIC with the flexibility of the DSP. Figure 20 depicts a comparison of different technolo-

gies regarding Flexibility versus Cost, Printed Circuit Board (PCB) area as well as Power [20].

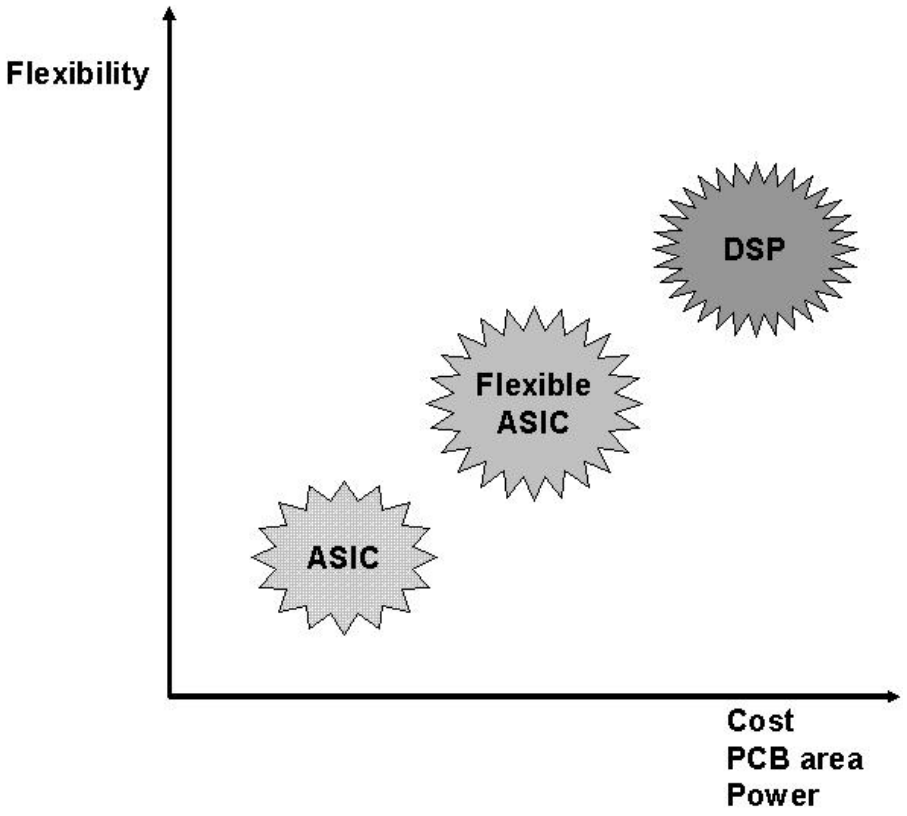

Figure 20: ASIC versus DSP

However, simply combining a DSP with an ASIC gives rise to a number of disadvantages. Firstly, there is the difficulty of building interfaces between the two units which have the width and speed required to benefit from the full potential of the ASIC as well as the DSP. Secondly, in comparison to the integrated configuration, the two unit configuration is characterized by a greater power dissipation. In some cases a greater efficiency can be achieved by using several DSPs, with only modest performance levels, in parallel instead of using only a single high performance DSP.

The Flexible ASIC concept involves the combination of a small number of DSP cores with application specific hardware on a single chip, thereby combining the speed and size of an ASIC implementation with the flexibility and programmability of a DSP. The layout of a Flexible ASIC chip is depicted in Figure 21 [19].

5.4 DSP cores

The adaptation of DSP cores for their respective applications results in a greater suitability of the interfaces and gives the implementation more computational power by stripping away unused logic. Also routing overhead can be minimized since a number of DSPs find place on the same chip. Higher efficiency with regard to the silicon area and a more economic

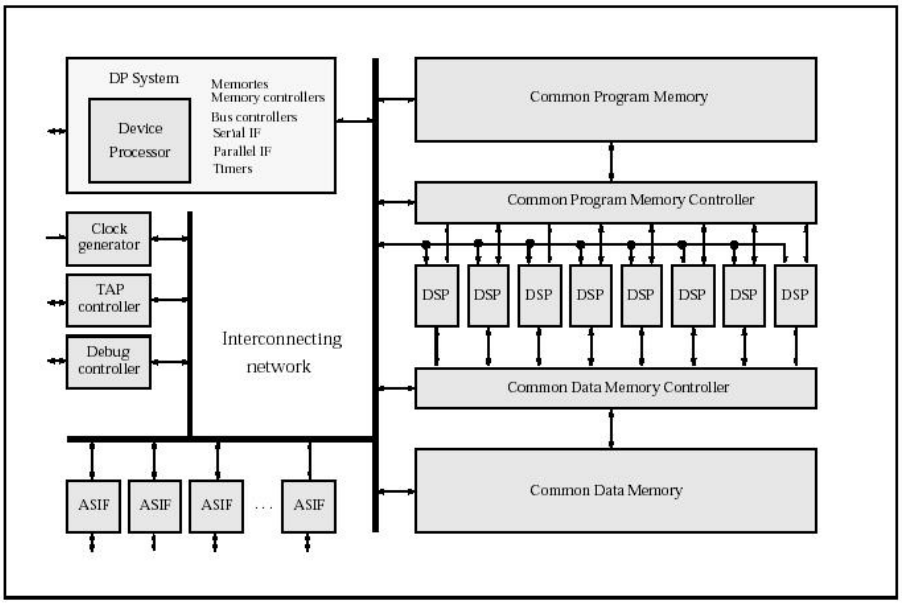

Figure 21: Flexible ASIC layout

ASIC are guaranteed through the use of a small internal Random Access Memory (RAM) in each DSP core as well as a common external RAM for several DSP cores as a group. Interfaces of the specific applications are custom made and implemented with the other ASIC architecture.

A common external memory utilizes the issue that programs often consist of split loops. A higher level of efficiency regarding memory usage as well as a minimized silicon area used for memory, is achieved if the common program memory contains the main program and the internal program memory of an assigned DSP code, contains a copy of a loop code.

For the case that every DSP core contains a copy of the program, a larger internal memory would be required and therefore a larger total memory. In addition, since memory generally requires a large amount of silicon area compared to other logic contained in a DSP core, the silicon area would increase as well.

As part of the Flexible ASIC project, different DSP-cores have been developed. Figure 22 depicts the general layout of a Flexible ASIC DSP core [19]. The PCU (Program Control Unit), the basic block, dispatches program instructions, received from the LPM (Local Program Memory), to the appropriate CU (Computational Unit). The LDM (Local Data Memory) is connected through the DAAU (Data Address Arithmetic Unit) and provides an address interface to the LDM which in turn is divided into memory banks of equal size (1024x16 bit). Also, the architecture enables simultaneous access from two different memory banks. The CDMIF (Common Data Memory Interface) provides a communication interface to the CDM (Common Data Memory) which is shared between several cores. Additionally, the cores are characterized by IRQ (Interrupt Requests) and timer functionality.

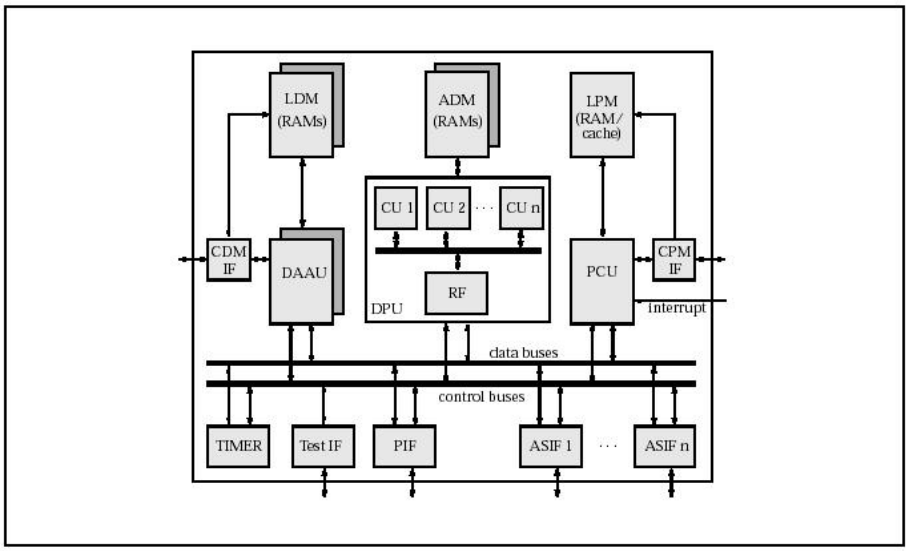

Figure 22: DSP core layout

5.5 Instruction Set

The DSP-cores allow the parallel execution of a maximum of four 16 bits of instructions in each clock cycle. Physical limitations, such as the number of CU:s or the number of busses, of the hardware, establish rules for how the instructions can be placed in parallel. Since there exist, for instance, two separate data and address busses in the DSP cores, a parallel performance of two memory move instructions is possible.

6 The Capella ASIC

6.1 Introduction

This section specifies the main characteristics of the Capella ASIC, the hardware platform on which the G.729 speech coding algorithm will be implemented.

The Capella ASIC consists of multiple DSPs and is designed specifically for speech-coding and echo-canceling applications. In Capella efficient memory usage is achieved since many DSPs share common memories. However, each DSP separately gets instructions from the CPM.

6.2 Block Diagram

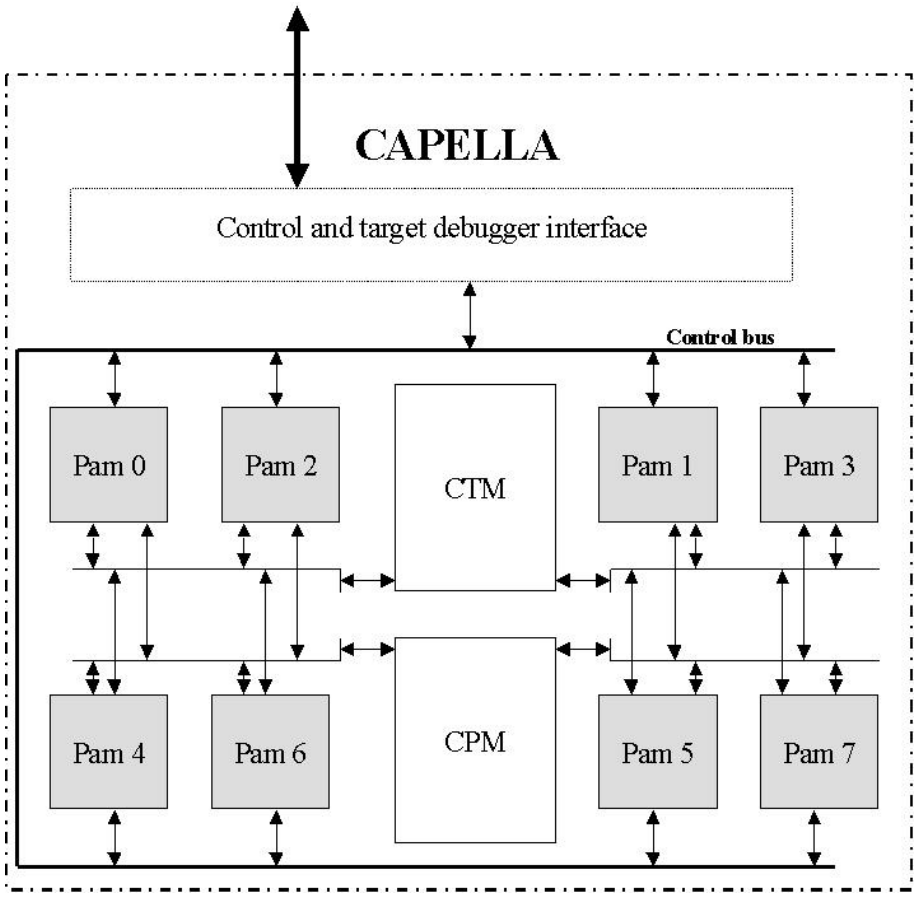

Figure 23: Block diagram of the Capella ASIC

Figure 23 depicts the block diagram for the Capella ASIC. As can be seen from the figure, Capella consists of eight DSPs. The blocks in the figure are:

- **Pam**
  Pam is a DSP core which is optimized for speech-coding and echo-cancelation algo-

rithms. Each Pam consists of two Computation Units (CU), one Accumulator register File and a Local Data Memory.

In normal operation, one instruction is fetched from program memory and one operand from data memory every instruction cycle.

- **CPM**
  CPM is the Common Program Memory. The CPM contains programs which can be run simultaneously in Capella.
- **CTM**
  CTM is the common table memory. In CTM ingoing and outgoing speech samples are saved and data shared between Pams.
- **LDM**
  LDM stands for Local Data Memory, which is the memory that each Pam uses for calculations. The memory block can hold 32768 words of 16 bits each.

6.3 Performance Specifics

- 120 MHz
- Four parallel instructions per clockcycle
- 16-bit fixed point

7 Flexible ASIC Development Tools

The Flexible ASIC tools represent a set of software tools and programs, particularly developed for the Flexible ASIC concept, which are used to support the development of DSP application programs for different core architectures. Through these tools a development environment for application software is established even before the hardware is available. This high degree of portability guarantees that following the production of a new hardware ASIC or DSP, in order to support the new hardware, the development tools can be altered without much effort. The software application is hereby constructed through the use of assembly or C language or a combination of both.

An availability of all the tools necessary for accurate software simulation of processors and external units as well as benchmarking and profiling is hereby guaranteed, as is also the case for multi DSP simulation and debugging. The following section focuses on the Flexible ASIC tools used in the G.729A implementation process.

7.1 The Flexible ASIC Assembler (Flasm)

Flasm is the general assembler in the set of FlexASIC development tools, which uses Instruction Set Descriptions (ISD) as well as run-time link libraries for the configuration of a specific target architecture (instruction set). A list of statements forms the basis of an assembler source file, whereby each statement can be an instruction row, a label definition or an assembler directive. Directives and labels are hereby characterized by the same syntax for all instruction sets. Although each instruction is individually described by an instruction set definition, the basic syntax rules apply to all instructions. An instruction row is defined as a pipe character separated list consisting of individual instructions which require parallel execution. The following basic syntax is used by all instructions:

<mnemonic><operand>,<operand>,...

Flasm uses a rule checker to identify illegal relations between instructions in the code and issues error and warning messages. The rule checker performs checks on the parallel execution of instructions, the access to computational units, repeat and block repeat instructions, stack manipulation as well as on read/write sequences of memories.

In this project, flasm was used to assemble the files written to check the overflow, see Implementation Process section.

7.2 ANSI-C Compiler with DSP-C Extension

Essentially, a compiler represents a program which translates a high-level programming language, such as, for instance, Ada, Pascal, Fortran or C, to the lowest code level known by the computer, machine code. While some compilers first translate the high-level code to assembly code and then pass this to a separate assembler or linker program, other compilers perform the entire process.

The Flexible ASIC C compiler, also known as flacc, represents an ANSI-C compiler with the DSP-C extension to the ISO C standard to provide support for the specific hardware features of DSPs. Several optimization engines are provided by flacc, which enables the performance of constant evaluation and expression simplification. In addition, flacc performs:

- expression canonization and the merger of basic blocks - optimization at the basic block or statement level, such as the elimination of tail recursion and jump-to-jump optimization
- loop analysis, optimization and hardware loops generation
- the propagation of copies of local variables and constants as well as the removal or change of useless code
- invariant code motion on the inner loops, which is optimized by rearranging address calculations in such a way that the invariant part is maximized - read-only data optimization, which sets all globals with static linkage to read-only if they have no direct assignments; this enables a constant evaluation to inline the initialized values

The DSP-C extension provides support for the specific hardware features of Digital Signal Processors. The most essential language elements, which are added, are hereby the fixed point data types, memory spaces, all of which allows for several memory qualifiers and circular buffer support.

Flacc represents the interface to the Flexible ASIC C compilation system which comprises an ANSI-C/DSP-C compiler with code generator, an assembler (flasm) and a link editor (flink). The supplied options will be processed by flacc which will then execute the different components with the respective arguments, whereby several types of files are accepted as arguments. While files with .c suffix are taken to be C source files, an object file with .obj suffix is produced in the case when the compilation process runs through the assembler. Assembly source files are files with .s or .asm suffix and may be assembled and link edited. Files with .o or .obj suffix are passed to the link editor which then produces an executable with an .out suffix.

As will be seen in the Implementation Process section, a series of problems with the compiler, flacc, occurred during this project.

7.3 Flexible ASIC Link Editor (Flink)

The Flexible ASIC Link Editor, also known as flink, represents the linker for the Flexible ASIC concept which creates executable modules through the combination of the object files created by the assembler (flasm).

The necessary information on the building of an executable system is derived by the linker through the use of a link control file which consists of two elements, the load blocks and the run blocks. The latter elements include the code description as well as initialized data in CPM, the description of uninitialized data, the description of run blocks and of data loaded in LDM. The linker which can be invoked with a set of switches controlling the resulting output, is responsible for the following tasks:

1. The relocation of symbols
2. To resolve references between input files
3. The combination of object file sections to load blocks
4. The allocation of load blocks into specific areas of common program memory (CPM)

5. The combination of load blocks to run blocks
6. The relocation of run blocks into specific areas of program memory
7. The provision of support for data load from the common data memory into the local data memory (LDM)

One important note about the linker is that a library file needs to be included in the linking process. This will be explained later in this report.

7.4 Fladb/flunk

The output derived from flasm is an object file which can be loaded into fladb/flunk. Whereas fladb is the simulator/debugger for FlexASIC, flunk is the graphical user interface version. Fladb is an instruction set simulator with debugging attributes which is used for the simulation and debugging of a Flexible ASIC DSP core program. The debugger kernel and the DSP model are the two elements of the debugger. The DSP model, in turn, comprises a number of DSP core block models which are loaded by the debugger under the control of a configuration file, which enables the configuration of the debugger to any Flexible ASIC DSP architecture without using a separate debugger for each.

Fladb enables the user to place breakpoints in the code and to observe CPU registers and LDM. In addition, fladb offers the possibility of a connection between a CPU register and a file on disk, which can be used for reading input data into fladb as well as writing results back to the disk.

Figure 24 shows the working environment with flunk. The upper left window is the source window, in the figure assembly mode is on. In the source window it is also possible to display the source code in C-mode. The upper right window is the register monitor and the lower right window displays the LDM. The lower left window is the command window.

7.5 Flism

Flism provides support to the development of Flexible ASIC programs, including application and verification programs. Hereby a high-level language, C, is used for simulations to load, run and debug programs written in flexible ASIC assembly language. It is possible to initialize fladb, the command line simulator, or flunk, both described above, and run them from a start to an end address through the call of a set of functions, the flism functions. Almost all of the registers and the entire memory can be read and written from the C program.

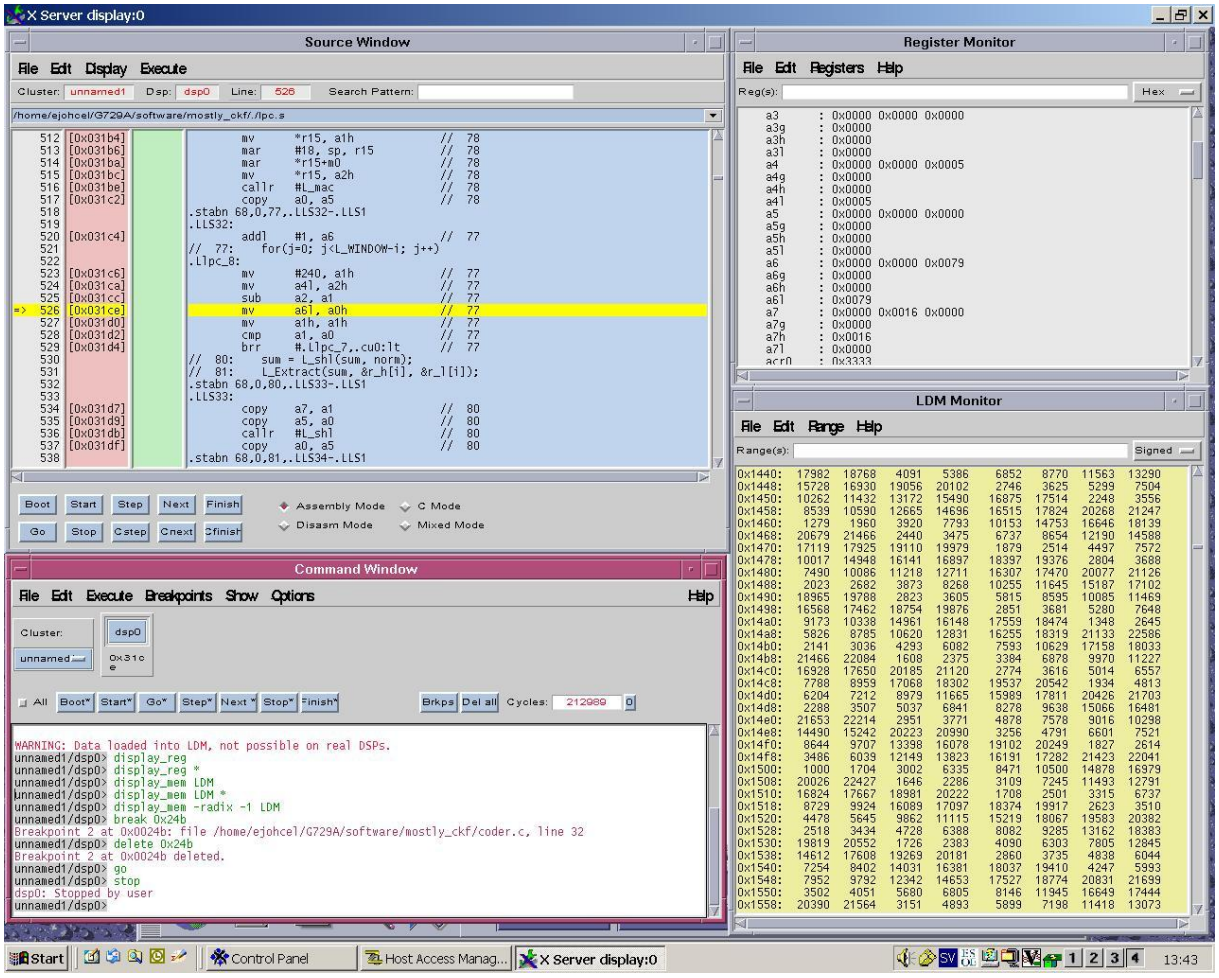

Figure 24: Working environment with flunk

8 Implementation Process

8.1 Introduction

This section documents the actual work done in implementing the G.729 to the Capella ASIC. The original plan was to significantly optimize the given C-source code for the Pam DSPs. The thought was that an optimized code would give the opportunity to run multiple codecs on the Capella ASIC. Unfortunately, due to a series of problems with the Flexible ASIC tools, the original optimization plan had to be modified and the ambitions be lowered. Most of the work efforts were put into error-seeking. However, with time, the error factors were found and eliminated, resulting in a working algorithm.

One limiting factor during this project has been the lack of information about previous speech-coder implementations with the Flexible ASIC tools. Apparently, the GSM speech coder has previously been implemented with the Flexible tools on an precursor of the Capella ASIC. This work was however performed at an Ericsson unit in Germany which does no longer exist.

8.1.1 Computers Used for Simulation

The Flexible ASIC tools are designed to operate in UNIX. Therefore, at first, a Sun Ultra 10 computer was used as workstation. To shorten simulation run-times the Ultra 10 was, later on, replaced by a SunBlade 150. Microsoft Visual C was run on an HP PC.

8.2 ITU-T G.729A Source Code

The source code from ITU-T consists of 32 C-source code files, five header files and a makefile, each for the coder and encoder. All the files are listed in Table 6. ITU-T also provides a set of testvectors which are listed in Table 7. When implementing the G.729A, these vectors can be used to verify bit-compatibility with the standard.

8.3 Test Run in Microsoft Visual C

As a first step in this thesis work, the G.729A speech source code was test-run in Microsoft Visual C. All test vectors were coded/encoded with success. Knowing that the code worked accurately in Visual C, Visual C could then be used to test and verify bit-accuracy after algorithmic changes.

8.4 Initial Modifications

8.4.1 Flextools Installation

At first, the flextools were installed on the workstation. The tools included:

- *flacc*, version 4_40, C-compiler for PAM.
- *flink* linker
- *fladb* simulator debugger

Table 6: G.729A source-code files

Encoder specific	Decoder specific	Common files
acelp_ca.c	decoder.c	basic_op.c
cod_ld8a.c	de_acelp.c	bits.c
coder.c	dec_gain.c	dspfunc.c
cor_func.c	dec_lag3.c	filter.c
lpc.c	dec_ld8a.c	gainpred.c
oper_32b.c	lspdec.c	lpcfunc.c
pitch_a.c	oper_32b.c	lspgetq.c
pre_proc.c	post_pro.c	pred_lt3.c
qua_gain.c	postfilt.c	p_parity.c
qua_lsp.c	tab_ld8a.c	util.c
tab_ld8a.c		basic_op.h
taming.c		ld8.h
		oper_32b.h
		tab_ld8a.h
		typedef.h

Table 7: G.729A test-vector files

Encoder Inputs	Encoder Outputs and Decoder Inputs	Decoder outputs
algthm.in	algthm.bit	algthm.pst
fixed.in	fixed.bit	fixed.pst
lsp.in	lsp.bit	lsp.pst
pitch.in	pitch.bit	pitch.pst
speech.in	speech.bit	speech.pst
tame.in	tame.bit	tame.pst
	erasure.bit	erasure.pst
	overflow.bit	overflow.pst
	parity.bit	parity.pst

- *flunk* graphical simulator debugger
- *flism* linkable simulator.
- *flasm* assembler

In the installation, care was taken of a common setup problem with the Command Pre-Processor (CPP) according to [12] section 5.1 solution 1.

8.4.2 Makefile Modifications

The makefile provided with the G.729A package was modified to comply with the flacc C-compiler. The ending of the object files were changed from ".obj" to ".o". For flacc the following flags were chosen:

- g for generating debug information and enabling simulation in C-mode in the graphical simulator.
- c to suppress the link editing face and creating an object file for each source file. The linking was performed with flink.
- 00 for no optimization for the initial test run.
- wall for displaying most warnings.

In the makefile all of the source files were referred to in small letters, whereas the actual source files were named with capital letters. Because UNIX is case sensitive all source files were renamed with small letters to correspond with the makefile.

In the linking phase, using flink, the necessary library file "rtlib.obj" was included. The way the file was included turned out to be slightly incorrect, see Error-Seeking Section.

8.4.3 File Reading

The main file for the encoder, "encoder.c", and the main decoder file, "decoder.c", reads in the speech samples from a file and also the output is written to file. This is not possible in a DSP. At this initial trial stage, all fread and fwrite instructions were removed from the two main files. Instead, input speech samples for one frame were saved in a vector in the encoder file. Later on, the temporary trial fix was replaced with a C-program which conducts input reading and output writing from file and linking into the simulator via flism.

8.5 Initial Simulation Problem

With modifications the G.729A code went through the compilation and linking stage. However, when the executable encoder file was loaded into flunk, flunk crashed. After correspondence with the Flexible ASIC support, it was verified that the crash was due to a bug in the Flex tools and an error report was filed.

In order to avoid flunk crashing, a temporary work-around was to compile without the -g flag which generates debug info for flunk. Then, is was possible to load the encoder executable program file into flunk. However, now C-mode could not be shown in flunk, thus, the debugging had to be done in assembler-mode.

During the time of the -g flag problem a new C-compiler (version 4.40) for Pam was released. Unfortunately, flunk crashed also after using this new compiler.

After some weeks, as a response to the filed error report, a beta version of flacc v.4.42 was released. In the beta version of the compiler the bug concerning the -g flag had been removed. The reason for the crash was that the register-mapping for the Pam core was wrong in the generation of stabs information for the registers. The G.729A code happened to utilize a rare kind of register optimization which lead to the mapping to a register that did not exist and therefore flunk crashed.

8.6 Flism Simulation Program

To enable simulation of the G.729A codec, with input data read from a file and output data written to a file, a C-program utilizing flism was developed. With flism, memories and registers can be read or written to and the simulator, either fladb or flunk, can be ran through this program. With all links set according to [12], this simulation program is compiled with gcc including a flism library file

gcc <c-files> -o <output> -I$(FLISA_TOOLS)/compat/include -I$(FLISA_TOOLS)/include -lflismx

When compiling the simulation program one has to choose if the graphical simulator, flunk, shall be used or the non-graphical, fladb. For flunk, -lflismx should be chosen, while for fladb only -lflism [11].

Two simulation programs were written, one for simulating the encoder and one for simulating the decoder. In the encoder simulation program, 80 speech samples corresponding to one frame, at a time are read from file and written into CTM starting from address 0x8000 and forward. These input samples are then coded by the encoder and the output bitstream is written into CTM, where the simulation program reads the samples and writes them to the output bitstream file. This is repeated until the end of the input data file.

The decoder uses the same principle. As input to the decoder then, of course, the output bitstream file from the encoder is used.

The simulation program for the encoder is named ”Sim_G729A_Enc.out” and the decoder simulation program is named ”Sim_G729A_Dec.out”. To run an encoder simulation

Sim_G729A_Enc.out < speech input file> <output bitstream file>

and likewise for the decoder

Sim_G729A_Dec.out < input bitstream file> <synthesized speech file>

8.7 Coder and Decoder Main Files Modification

In order to comply with the simulation programs, the main files of the encoder and decoder needed to be changed. As mentioned, the input speech samples are placed into CTM by the simulation program. The encoder file was rewritten to copy the input data from CTM to LDM. Then, these samples are processed and the output bitstream is first put into LDM, then copied to CTM from where the simulation program reads the output samples. This principle is the same for encoder and decoder.

Originally, both, the main files of the encoder and decoder, ran initialization functions before starting the actual coding frame by frame. In the initialization functions, pointers are set which are used in precursive calculations. However, the simulation program is designed so it reads one frame at a time, runs the coder and reads the output data. Thus, without modifications, the simulation of the encoder and decoder ran the initialization process for each frame and pointers were misplaced.

To deal with this, the simulation program was designed to set a certain bit in the CTM indicating the first frame. This bit is checked by the encoder main file, which then runs the initialization functions only for the first frame when the control bit is set. In the simulations, the control bit was chosen to the address 0x9000 in CTM.

8.8 Error-Seeking and Algorithmic Changes

Once it was possible to simulate the encoder with the flism program and the given testvectors, the correct functioning of the coder could easily be checked. Unfortunately, the coder did not function correctly. The output bitstream from the encoder for a given input testvector did not match the corresponding bitstream testvector.

8.8.1 Big and Little Endian

The first natural step in finding the error in the encoding was to check if the input samples in DSP's LDM were correct. When looking at the LDM in flunk, it was obvious that the 80 input speech samples were there, although, the values were not the same as in the input codevector. The flism simulation program was then checked and rewritten several times but the fault was not found. However, when looking closer at the values in the LDM and comparing them to the original testvector a pattern emerged. For example:

Original values 8, 16, -16
Values in simulation 2048, 4096, -3841

With an experienced eye, Tore André noted that the problem was due to the fact that the input bytes were read in wrong order. Each input speech sample consists of 16 bits, which yields 2 bytes. Table 8 depicts the byte-swapping for the decimal numbers 8/2048. The reason for the byte-swapping is that the Sun Sparc processor stores the most significant byte first. This is called *Big Endian*. Whereas, Intel processors use *Little Endian* where the least significant byte is stored first. Thus, the testvectors were stored in little endian but read in big endian so the two bytes for each speech sample were reversed and the number did not make sense.

Table 8: Big and little Endian Ilustration

Decimal 8 in Little Endian															
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
yields decimal 2048 with Big Endian															
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

To deal with the endianess a program was written to reverse the bytes in the testvectors. The executable program file is called ”EndianConv.out”

8.8.2 Known limitations to the C-compiler

With the endianess figured out and with correct input samples the encoder still did not give a correct output bitstream. Even though the flacc C-compiler is said to be ANSI-C compatible, certain aspects were not implemented according to the standard specifications. These limitations are documented in [10]. From the document, two stated limitations that possibly could be part of the G.729A code were identified. These limitations were:

1. **Relation expression for loops**
   In a relation expression, for *for* and *while* statements, comparing the loop control variable to a long data type (32-bits representation) is not implemented.

```
int long_loop(int *a, long b)
{
int i;
for (i=0; i<b; i++)
{
*a+=1;
}}

or

int long_loop(int *a)
{
int i;
for (i=0; i<0x1ffff; i++)
{
*a+=1;
}}
```

2. **Function pointers as parameter**
   Passing function pointers as parameter is not implemented. An illustration of the case is given in the following code.

```
int perform(int a, int b, int (*f) ())
{
return (*f) (a,b);
}
```

These two cases will not be compiled or a wrong code will be generated [10]. The G.729A source code was searched for occurrences of code resembling to the one given in the examples above.

8.8.3 Rewriting 32bit Comparison Loops

To search the G.729A source code for the limitation given in case 1, all *for* and *while* loops in the code had to be controlled. This was done by

```
grep -i ”while” *.c
or
grep -i ”for” *.c
```

and then manually looking at the declaration of the comparison variables for each loop.

One occurrence matching the search criteria was found in the file ”basic_op.c” in the function Word16 norm_l(Word32 L_var1)

```
for(var_out = 0;L_var1 < (Word32)0x40000000L;var_out++)
{ L_var1 <<= 1;
}
```

which was then rewritten to

```
int a=0; int b=0;
...
if(L_var1 < (Word32)0x40000000) {a=1;}
for(var_out = 0;a!=b;var_out++)
{
L_var1 <<= 1;
if(L_var1 >= (Word32)0x40000000) a=b;
```

However, after the change, the encoder did still not function correctly so the next step was to look for function pointers.

8.8.4 Rewriting Function Pointer

To search for code that resembled the second known limitation, as previously stated, was a more difficult task than the 32-bit comparison loop. With no ingenious idea, function pointers were searched with

```
grep ”(” *.c
```

which then gave all occurrences of left parantheses in the entire code.

One function pointer was found in the file ”lpc.c” in the Levinson function. It was declared as

```
Word16 (*pChebps)(Word16 x, Word16 f[], Word n)
```

and the function pointer was used at two times within the function. However, it was possible to remove this function pointer and instead rewrite the code, were the function pointer was used, with extra if and else statements.

Even with the function pointer removed, and the G.729A source code carefully searched for other occurrences of code matching the compiler known limitations, the encoder still had some bits wrong in the output bitstream.

8.8.5 Debug Method

Knowing that the encoder did not function correctly, and that there was a problem somewhere in the complex code, a good debugging method had to be used to find the code causing an error.

The graphical simulator/debugger flunk has a command cprint with which it is possible to print the value of a varible in C-mode. Initially, the debugging was performed by inserting breakpoints and looking at variables with cprint and comparing them to variable values taken from the original codec ran in Microsoft Visual C. This soon turned out to be a hopeless method. Many times, the cprint command printed a variable value as zero even if in fact it had a value that differed from zero. This could be confirmed by looking at the DSPs memory. Thus, using the cprint command was not reliable.

Fortunately, flacc allows the printf command. It is also possible to use printf in flunk, then the printables are displayed in the shell window. Thus variable values could be checked with printf. To simplify the control of the values displayed with printf, the original G.729A C source code was also compiled with gcc. Thus, printf commands could be added to the code and then compiled with both, flacc and gcc. The output displayed in the shell window could be saved in a file for both the original code and for the flunk simulated code. These two files could then be compared. With this debugging method many printf commands could be placed in the code and the values of several variables could be checked at once.

8.8.6 Wrong Type Definitions

In the debugging process, an error was found in the function L_mult in the file ”basic_op.h”, which was, at first, thought to be due to a compilation error. However, the fault was not in the compiler. Instead, the wrong type definitions had been used in the file ”typedef.h” in the initial modifications of the code. In flex an *int* is 16 bits and *long* is 32 bits. A silly mistake.

With the right type-definitions the function L_mult worked correctly. The encoder, however, did still give errors in the output bitstream.

8.8.7 Increment Problem

The last cause of error required a substantial amount of work to find. To track down the cause of error the output bitstream was analyzed for incorrect bits. In the ”read.me” file, included with the source code, each position in the bitstream is matched with its corresponding coded parameter, according to Table 9. Thus, the debugging method now included: encoding a frame and then analyzing the output bitstream, identifying which positions in the bitstream were incorrect and then finding where in the encoder these parameters are coded.

One source of error was tracked down to the main encoder file ”cod_ld8a.c” and the function cod_ld8a. When looking at the addresses of pointers, it was obvious that something was incorrect.

In the DSPs memory, the right parameter values were stored, but the pointers could point at a memory address to the left or right of the address with the wanted value.

The first identified line of code that caused an error was

```
*p0=mult(*p0++,psign[i1]);
```

Where mult is a function, p0 a pointer and psign a vector. The error in this line laid in the increment of the pointer p0.

If this code was rewritten as

```
*p0=mult(*p0,psign[i1]);*p0++;
```

it functioned desirably.

In this thesis work, the same code had previously been compiled with three different compilers (Visul C, gcc, TI) without this problem occurring. Now, certain that a compiler error was found the FlexASIC support was contacted again . The following states their response.

In the first case, with the original code:

The function call is a sequence point (section 3.3.2.2 in the C-standard), the increment must be done before the function call is done (defined from section 2.1.2.3 in the standard).

It is undefined whether the left hand side or the right hand side of an assignment is evaluated first, so ’p0’ may be read before the argument to the function is evaluated. It may also be read after its evaluation, but before the function call, or after the function call (supposed ’p0’ is a global object, ’mult’ may again change its value).

Second case:

The function call and the first assignment are both done (end of statement is a sequence point), the increment must be done after the first assignment is completed.

Whether it is a bug:

No, it is in the area of undefined and unspecified issues in the C-standard. I think all these compilers are OK. The programmer should not use such constructions.

Interesting to note is that the ITU-T G.729A C source code is written with a code that is not defined in the ANSI-C standard.

Since this was not the only line in the code using this kind of increment by pointers, all similar occurrences had to be found and rewritten according to case 2. Naturally, the same issue concerned decrement of pointers in the same context.

Thus, the next step was to replace all faulty lines of code which was done by searching for ++ and – in all .c files. The search resulted in an extensive response . These type of function calls, with pointer increment or decrement were used in several files which were all changed.

The changed files using this kind of function calls are: ”acelp_ca.c”, ”cod_ld8a.c”, ”dec_ld8a.c”, ”postfilt.c”, ”bits.c” and ”filter.c”.

With these changes both, the encoder and decoder, were successfully tested with the testvectors speech.(in/bit/pst) and algorithm.(in/bit/pst) with success. Also some self-created shorter testvectors, were tested with success. The reason why all provided testvectors have not yet been tested is due to the long simulation time, which is handled in the next section.

Table 9: Bitstream bit allocation

Word	Parmeter	Description
01	LPC1-	MA predictor switch
02	LPC1-	1st codebook 7 bit
03	LPC1-	
04	LPC1-	
05	LPC1-	
06	LPC1-	
07	LPC1-	
08	LPC1-	
09	LPC2-	2nd codebook low 5 bit
10	LPC2-	
11	LPC2-	
12	LPC2-	
13	LPC2-	
14	LPC2-	2nd codebook high 5 bit
15	LPC2-	
16	LPC2-	
17	LPC2-	
18	LPC2-	
19	M_1	pitch period 8 bit
20	M_1	
21	M_1	
22	M_1	
23	M_1	
24	M_1	
25	M_1	
26	M_1	
27		parity check on 1st period 1 bit
28	CB_1	codebook pulse positions 13 bit
29	CB_1	
30	CB_1	
31	CB_1	
32	CB_1	
33	CB_1	
34	CB_1	
35	CB_1	

Continued on next page

Table 9: *continued*

Word	Parmeter	Description
36	CB_1	
37	CB_1	
38	CB_1	
39	CB_1	
40	CB_1	
41	S_1	codebook pulse signs 4 bit
42	S_1	
43	S_1	
44	S_1	
45	G_1	pitch and codebook gains 3 bit stage 1
46	G_1	
47	G_1	
48	G_1	pitch and codebook gains 4 bit stage 2
49	G_1	
50	G_1	
51	G_1	
52	M_2	pitch period (relative) 5 bit
53	M_2	
54	M_2	
55	M_2	
56	M_2	
57	CB_2	codebook pulse positions 13 bit
58	CB_2	
59	CB_2	
60	CB_2	
61	CB_2	
62	CB_2	
63	CB_2	
64	CB_2	
65	CB_2	
66	CB_2	
67	CB_2	
68	CB_2	
69	CB_2	
70	S_2	codebook pulse signs 4 bit
71	S_2	
72	S_2	
73	S_2	
74	G_2	pitch and codebook gains 3 bit stage 1
75	G_2	
76	G_2	
77	G_2	pitch and codebook gains 4 bit stage 2
78	G_2	

Continued on next page

Table 9: *continued*

Word	Parmeter	Description
79	G_2	
80	G_2	

8.9 Simulation Results and Aspects

8.9.1 Simulation Results

Table 10: G.729A encoder simulation results

	Cycles	Instructions	MIPS Approx.
Frame 1	2,652,804	1,295,785	130
Preceding frames	2,614,383	1,266,476	130

Table 11: G.729A decoder simulation results

	Cycles	Instructions	MIPS Approx.
Frame 1	545,340	268,474	27
Preceding frames	534,557	261,024	26

8.9.2 Simulation Time

Simulation times proved to be an important, and negative factor, when simulating the G.729A codec in flunk. To simulate the fully-optimized encoder takes approximately 45 seconds per frame of 10 ms. For the decoder it takes approximately 10 seconds per frame. For example, the testvector ”speech.in” consists of 3750 10 ms frames. To completely simulate the encoder and decoder for this file then takes about 60 hours.

Therefore it is strongly recommended to use shortened testvectors when testing the codec after minor changes.

8.10 Future Optimization

For the intended ToIP application, a further optimization of the G.729A codec is desirable. However, due to limited time, further optimization of the codec is not possible as part of this thesis work, although some efforts were put into using the Compiler Known Functions, discussed below.

8.10.1 Compiler Known Functions

The Compiler-Known functions (CKF) are compiler-recognized functions which are directly mapped into a set of assembly instructions by the compiler. The instructions generated by CKFs are scheduled with other instructions and all arguments and return values are allocated to registers, which results in a very efficient code.

The CKF include a ETSI library of functions. These CKF functions functions correspond to functions in the ETSI GSM-HS and ETSI GSM-EFR codecs.

All of the functions in the files ”basic_op.c” and ”oper_32b.c” in the G.729A codec can be found in the CKF library. However, even if the functions provided by the CKF correspond to the ones in the G.729A ”basic_op.c” in the definitions, there is one crucial difference. The functions in the G.729A ”basic_op.c” return an overflow flag. This flag is set if a variable value is out of range for a 32-bit signed integer.

Thus, it is possible to use the CKFs, but then there is the dilemma of how to check the occurrence of an overflow. One possible solution to this is to check the status register of the DSP. The solution which seems to be used in the AMR speech coder, also implemented on Capella, is to clear the the overflow register with

```
mv #0, custat1
```

and test for overflow with

```
brr #AmrDec_Decoder_amr_core_dec_amr.L250,.cu0:lov32
brr #AmrDec_Decoder_amr_core_dec_amr.L250,.cu1:lov32
```

Where custat0 and custat1 are the computational unit status registers and LOV32 is the control bit for Latched Overflow at bit 32 flag. This solution was proposed by Francesco Agnoni (RM/ERI).

The G.729A codec was implemented with the CKFs and assembly code was written to check the overflow according to the proposed solution. This did not function desirably. However, with more working effort, a solution to check for overflow when using the CKFs should be found relatively easy.

8.10.2 Complexity Profile

To simplify future optimization of the G.729A codec, a complexity analysis was performed. This analysis yields a complexity profile which is displayed in Appendix A. In the profile, one can see how much of the encoders or decoders execution time is allocated to each function. Thus, from the complexity profile one can see which functions should be prioritized when optimizing the G.729A encoder and decoder.

8.10.3 Optimizing Techniques

To further optimize the G.729A codec, several techniques may be applied. To mention a few:

- **Multichannel Modifications**
  In the ToIP application, the G.729A codec should work in a multi-channel environment. Therefore, global variables and constants do not need to be saved separately for each coder. Instead, these data may be shared by several codecs.

- **C Level Optimization**
  With a high performance C-compiler, a high level of optimization may be achieved solely with optimizing in C.

- **Assembly level implementation**
  Rewriting C functions directly in assembly often increases speed and decreases code size. Optimizing in assembly has the advantage that efficient code can be written even by an not-so experienced programmer, whereas for writing code in C the programmer's skill is a more crucial factor.

Since the G.729 and G.729A codecs have previously been implemented on several DSPs, many good documents exist, which may be consulted for guidance on further optimization. The reader may, for instance, consult [6], [31], [3], [28] and [30]

8.10.4 Estimation of Optimization Workload

When using the CKF, the encoder execution takes only 800,000 cycles per frame (80 MCPS) and the decoder 500,000 cycles per frame. (50 MCPS). These values are taken from a simulation when there is no overflow.

Although, even if the overflow flag is set during the encoding of a frame it does not affect execution time noticeable. This was confirmed by modifying the original code not using CKF, to disregard the overflow check. Then, the execution time could be compared between the version using overflow check and the one not checking for overflow.

Assuming that the coder with CKF corresponds to the porting phase of document [31], which is a qualified assumption since in the document the porting phase includes ”intrinsic” functions which correspond to the Flexible ASIC CKF, one could give an estimation of the expected workload of further optimization.

In document [31] the encoder was optimized from the starting point of 15.07 MCPS to 4.7 MCPS in approximately seven man-months. Using this as a basis for an estimation, and with our porting phase for the encoder and decoder summarized to 130 MCPS, a rough estimate could be to have the encoder and decoder optimized to a sum of about 40 MCPS in seven man-months. This would then enable three channels per Pam DSP core and a total of 24 channels. However, it should be possible with even more channels if extensive effort is dedicated to the optimization. The limiting factor for maximum number of channels is, however, not only the processing power available, memory usage might in fact be the issue that sets the upper limit.

Note, this is a very vague estimation and can be at most seen as a qualified guess. Notable is also that the decoder execution time is hardly affected at all by using the CKF. Therefore, one can assume that it will be more difficult to lower the execution time for the decoder than the encoder.

9 Conclusion

VoIP is an expanding field in which high levels of research and development are invested. The benefits of VoIP are cost-savings and the possibilities of innovative services. Ericsson's ToIP solution is particulary interesting since it aims to provide an exceptionally high-quality service. If this ToIP system is able to overcome to the many deficiencies of other existing VoIP solutions, it could indeed be a very successful product.

In the process of implementing the G.729A speech coder on the Capella ASIC, several problems were encountered and eliminated in the course of this thesis work. Since the coder is now functioning correctly, this work can serve as a solid basis for further optimization. The Capella ASIC, with its eight DSPs, is a powerful platform for implementing the G.729A. If substantial efforts are put into optimizing the coder, it is certainly possible to run numerous G.729A codecs on one Capella. Thus, the Capella could process multiple channels.

10 References

[1] The g.729 speech coding standard. a description of g. 729, and its low complexity and silence suppression annexes. Technical brief, nortelnetworks. http://www.nortelnetworks.com. Accessed on September 7, 2004.

[2] Speech production. http://ispl.korea.ac.kr/ wikim/research/speech.html. Accessed on August 15, 2004.

[3] Chiouguey Chen and Xiangdong Fu. G.729/a multichannel tms320c62x implementation. Application report, Texas Instruments, February 2000.

[4] W.C. Chu. *Speech Coding Algorithms: Foundation and Evolution of Standardized Coders*. Wiley-Interscience, 1st edition, March 2003.

[5] Intel Corporation. Intel internet video phone trial applet 2.2. http://www.chebucto.ns.ca/řakerman/articles/ig-h323_firewalls.html, 1997. Accessed on August 15, 2004.

[6] B. Costinescu, R. Ungureanu, M. Stoica, E. Medve, R. Preda, M. Alexiu, and C. Illas. Itu-t g.729 implementation on starcore sc140. Application note, Motorolla, February 2001.

[7] Protocol Dictionary. Voice over ip and voip protocols. http://www.javvin.com/protocolVOIP.html. Accessed on August 15, 2004.

[8] M. Engström and H. Olsson. A study of speech codec implementations with the flexible asic concept. Master's thesis, Chalmers University of Technology and Ericsson Microwave Systems AB, November 1996. Ericsson Document No. SR/HX-96:047.

[9] Flexasic. Flexasic homepage. http://flexasic.ericsson.se. Ericsson internal.

[10] Flexasic. Known limitations in the flexible asic c compiler. http://flexasic.ericsson.se, March 2000. Ericsson Document No. ERA/X/F-99:059 Uen.

[11] Flexasic. Flexasic flism - programmers guide. http://flexasic.ericsson.se, September 2002. Ericsson Document No. ERA/X/F-99:085 Uen.

[12] Flexasic. Flexasic getting started guide. http://flexasic.ericsson.se, July 2002. Ericsson Document No. RLG/T 02:.

[13] J.D. Gibson, T. Berger, T. Lookabaugh, D. Lindberg, and R.L. Baker. *Digital Compression for Multimedia: Principles and Standards*. Morgan Kaufmann Publishers, Inc., San Fransisco, CA, 1998.

[14] J. Häggström. Puheenkodaus tietoliikenteessä. http://keskus.hut.fi/opetus/s38116/1996/esitelmat/40457h/. Accessed on August 15, 2004.

[15] X. Huang, A. Acero, and H.-W. Hon. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. Prentice Hall PTR, Upper Saddle River, NJ 07458, 2001.

[16] ITU-T. Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited-linear-prediction (cs-acelp), March 1996. Recommendation G.729.

[17] ITU-T. Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited-linear-prediction (cs-acelp), November 1996. Recommendation G.729, Annex A.

[18] A. Kiviluoto. Speech coding standards. http://mia.ece.uic.edu/ papers/WWW/MultimediaStandards/chapter3.pdf . Presentation.

[19] J. Kjellsson and M. Wallberg. Evalutaion of the flexible asic c compiler for wcdma algorithms. Master's thesis, Lulea University of Technology and Ericsson Radio Systems AB, February 2000. Ericsson Document No. ERA/X/F-00:024 Uen.

[20] M. Liljeblad and D. Sandberg. Evalutaion of the flexible asic concept for wideband cdma implementation. Master's thesis, Lulea University of Technology and Ericsson Radio Systems AB, March 1999. Ericsson Document No. ERA/X/F-99:025 Uen.

[21] P. Mehta and S. Udani. Voice over ip: Sounding good on the internet. *IEEE Potentials*, 20, October-November 2001.

[22] Newport Networks. Voip bandwidth calculation. http://www.newport-networks.com/whitepapers/voip-bandwidth2.html. Accessed on August 23, 2004.

[23] Protocols.com. Voice over ip. http://www.protocols.com/pbook/VoIPFamily.htm. Accessed on August 15, 2004.

[24] L.R. Rabiner and R.W. Schafer. *Digital Processing of Speech Signals*. Prentice-Hall,Inc., Englewood Cliffs, NJ 07632, 1978.

[25] J. Samuelsson. 2e1400 speech signal processing. Lecture Notes. http://www.s3.kth.se/speech/courses/2E1400/. Accessed on August 15, 2004.

[26] P. Senesi, P. Ferrabone, G. Gritella, R. Rinaldi, and M. Siviero. Telephony over ip: theoretical modelling and lab experiments. *Universal Multiservice Networks, 2000. ECUMN 2000. 1st European Conference on , 2-4 Oct. 2000*, pages 262 – 271, October 2000. IEEE.

[27] J. Svedberg. Speech coding an overview. EAB/TVP,Multimedia Technologies, Ericsson Research, August 2004. AL/EAB-presentation.

[28] Lim Hong Swee. Implementation of g.729 on the tms320c54x. Application report, Texas Instruments, March 2000.

[29] S .Tomazic, A. Vugrinec, and P. Skraba. Wireless communication employing high altitude long endurance aeronautical platforms. *Electrotechnical Conference, 2000. MELECON 2000. 10th Mediterranean*, 1(29-31):361–364, May 2000.

[30] A. Tripathi, S. Verma, and D.D. Gajski. G.729e algorithm optimization for arm926ej-s processor. Technical report, Center for Embedded Computer Systems, University of California, Irvine, March 2003.

[31] R. Ungureanu, B. Costinescu, and C. Illas. Itu-t g.729a implementation on starcore sc140. Application note, Motorola, July 2001.

11 Appendix A

The complexity profiles of the ITU-T G.729A encoder and decoder are displayed in Table 12 and Table 13, respectively. These complexity profiles were created when encoding and decoding the testvectors speech.in/bit.

Self seconds: the number of seconds accounted for by this function alone. This provides the basis for how the lists are sorted.
Calls: the number of times this function was invoked, if this function is profiled, otherwise blank.

Table 12: G.729A encoder complexity profile

Self seconds	Calls	Function name
7.48	122070032	L_mult
5.65	101962120	L_add
4.72	94023896	L_mac
2.46	36883675	sature
1.35	40978205	extract_l
1.11	22052914	L_sub
1.09	14753230	mult
0.96	18093661	L_msu
0.96	15000	Cor_h_X
0.91	30000	Syn_filt
0.90	21368	Pred_lt_3
0.90	3750	Pitch_ol_fast
0.89	7500	Lsp_pre_select
0.85	7500	D4i40_17_fast
0.82	4395127	L_shl
0.77	15943124	sub
0.71	13822969	extract_h
0.67	3750	Autocorr
0.36	5287321	add
0.34	85118	Dot_Product
0.33	3552428	L_shr
0.33	7500	Cor_h
0.31	5812220	round
0.26	3144170	Mpy_32_16
0.24	2139170	L_Extract
0.21	7500	Residu
0.20	273149	Chebps_11
0.20	7500	Lsp_select_1
0.19	7500	Lsp_select_2
0.12	3750	Pre_Process
0.10	7500	Qua_gain

continued on next page

Table 12: *continued*

Self seconds	Calls	Function name
0.10	3750	Coder_ld8a
0.09	900000	mult_r
0.09	47615	Copy
0.09	3750	Levinson
0.08	1194250	shr
0.08	7500	Corr_xy2
0.07	89539	div_s
0.06	498750	Mpy_32
0.05	675000	L_abs
0.05	150000	norm_l
0.05	7500	G_pitch
0.04	830534	shl
0.04	15000	Lsp_expand_1_2
0.04	7500	Lsp_Az
0.04	7500	Pitch_fr3_fast
0.04	3750	Lsf_lsp2
0.04	3750	Relspwed
0.03	816578	L_deposit_l
0.03	3750	Az_lsp
0.03	3750	Lsp_lsf2
0.03	3750	Lsp_prev_compose
0.02	397500	L_deposit_h
0.02	37500	Div_32
0.02	7500	Lsp_get_tdist
0.02	7500	Lsp_prev_extract
0.01	299887	negate
0.01	15000	Get_lsp_pol
0.01	15000	Log2
0.01	11387	memcpy
0.01	7507	Set_zero
0.01	7500	ACELP_Code_A
0.01	7500	Gain_predict
0.01	7500	Gbk_presel
0.01	7500	update_exc_err
0.01	3750	Get_wegt
0.01	3750	Lag_window
0.01	3750	Lsp_get_quant
0.01	3750	Lsp_qua_cs
0.01	3750	Lsp_stability
0.01	3750	Parity_Pitch
0.01	1	main
0.00	251250	L_Comp
0.00	86250	abs_s

contiuned on next page

Table 12: *continued*

Self seconds	Calls	Function name
0.00	82500	L_shr_r
0.00	41250	int2bin
0.00	41250	norm_s
0.00	14634	L_negate
0.00	11250	Inv_sqrt
0.00	7518	ferror_unlocked
0.00	7500	Enc_lag3
0.00	7500	Gain_update
0.00	7500	Lsp_expand_1
0.00	7500	Lsp_expand_2
0.00	7500	Pow2
0.00	7500	Weight_Az
0.00	7500	test_err
0.00	3759	memchr
0.00	3750	Int_qlpc
0.00	3750	Lsp_last_select
0.00	3750	Lsp_prev_update
0.00	3750	Qua_lsp
0.00	3750	prm2bits_ld8k
0.00	1080	Chebps_10
0.00	19	mutex_lock
0.00	19	mutex_unlock
0.00	9	getiop
0.00	4	sbrk
0.00	3	atexit
0.00	3	get_mem
0.00	3	ioctl
0.00	3	isatty
0.00	2	cleanfree
0.00	2	free_mem
0.00	2	fstat64
0.00	2	lseek64
0.00	2	malloc
0.00	2	memset
0.00	2	realfree
0.00	2	strlen
0.00	1	Init_Coder_ld8a
0.00	1	Init_Pre_Process
0.00	1	Init_exc_err
0.00	1	Lsp_encw_reset
0.00	1	check_nlspath_env
0.00	1	exit
0.00	1	fflush

contiuned on next page

Table 12: *continued*

Self seconds	Calls	Function name
0.00	1	getegid
0.00	1	geteuid
0.00	1	getgid
0.00	1	getuid
0.00	1	mem_init
0.00	1	profil

Table 13: G.729A decoder complexity profile

Self seconds	Calls	Function name
1.45	26089523	L_mult
1.09	17869979	L_add
0.85	15043720	L_mac
0.73	22500	Syn_filt
0.60	2500931	L_shl
0.50	8708606	L_sub
0.47	8212083	L_msu
0.32	7500	Pred_lt_3
0.21	3589711	sature
0.18	3154562	extract_h
0.18	7500	Residu
0.17	4241794	extract_l
0.17	63755	Copy
0.16	7500	pit_pst_filt
0.15	1758821	mult
0.13	7500	agc
0.10	2293759	round
0.09	3750	Post_Process
0.08	898283	sub
0.07	1177745	shr
0.06	7500	Gain_predict
0.05	932607	add
0.04	757500	Mpy_32_16
0.04	679600	L_shr
0.04	41250	bin2int
0.04	3750	Decod_ld8a
0.04	3750	Post_Filter
0.03	457500	L_Extract
0.03	16418	div_s
0.03	3750	Lsp_prev_compose
0.02	82500	L_shr_r

contiuned on next page

Table 13: *continued*

Self seconds	Calls	Function name
0.02	7500	Lsp_Az
0.01	189785	L_deposit_l
0.01	37500	shl
0.01	7505	Set_zero
0.01	7500	Decod_ACELP
0.01	7500	preemphasis
0.01	3750	Lsp_get_quant
0.01	3750	bits2prm_ld8k
0.01	1	main
0.00	43759	norm_l
0.00	37083	L_deposit_h
0.00	15000	Get_lsp_pol
0.00	15000	Log2
0.00	15000	Weight_Az
0.00	15000	negate
0.00	11386	memcpy
0.00	7520	ferror_unlocked
0.00	7500	Dec_gain
0.00	7500	Dec_lag3
0.00	7500	Gain_update
0.00	7500	L_Comp
0.00	7500	Lsp_expand_1_2
0.00	7500	Pow2
0.00	7083	Inv_sqrt
0.00	3760	memchr
0.00	3750	Check_Parity_Pitch
0.00	3750	D_lsp
0.00	3750	Int_qlpc
0.00	3750	Lsf_lsp2
0.00	3750	Lsp_iqua_cs
0.00	3750	Lsp_prev_update
0.00	3750	Lsp_stability
0.00	19	mutex_lock
0.00	19	mutex_unlock
0.00	9	getiop
0.00	7	thr_main
0.00	4	sbrk
0.00	3	atexit
0.00	3	get_mem
0.00	3	ioctl
0.00	3	isatty
0.00	2	cleanfree
0.00	2	free_mem

continued on next page

Table 13: *continued*

Self seconds	Calls	Function name
0.00	2	fstat64
0.00	2	lseek64
0.00	2	malloc
0.00	2	memset
0.00	2	realfree
0.00	2	strlen
0.00	1	Init_Decod_ld8a
0.00	1	Init_Post_Filter
0.00	1	Init_Post_Process
0.00	1	Lsp_decw_reset
0.00	1	check_nlspath_env
0.00	1	exit
0.00	1	fflush
0.00	1	getegid
0.00	1	geteuid
0.00	1	getgid
0.00	1	getuid
0.00	1	mem_init
0.00	1	profil