Optimisation techniques for low bit rate speech coding

Shum, Ellen

Optimisation Techniques for Low Bit Rate Speech Coding

Ellen Shum

Thesis for qualification for degree of Master of Science

University of Durham

School of Engineering and Computer Science

Volume 1 of 1

September 1998

Declaration

I declare that this work is completely my own, that it comes from no other outside source and that no portion of this work has been previously submitted for a degree in this or any other university.

Statement of Copyright

Abstract

This thesis extends the background theory of speech and major speech coding schemes used in existing networks to an implementation of GSM full-rate speech compression on a RISC DSP and a multirate application for speech coding.

Speech coding is the field concerned with obtaining compact digital representations of speech signals for the purpose of efficient transmission. In this thesis, the background of speech compression, characteristics of speech signals and the DSP algorithms used have been examined. The current speech coding schemes and requirements have been studied.

The Global System for Mobile communication (GSM) is a digital mobile radio system which is extensively used throughout Europe, and also in many other parts of the world. The algorithm is standardised by the European Telecommunications Standardisation Institute (ETSI). The full-rate and half-rate speech compression of GSM have been analysed.

A real time implementation of the full-rate algorithm has been carried out on a RISC processor GEPARD by Austria Mikro Systeme International (AMS). The GEPARD code has been tested with all of the test sequences provided by ETSI and the results are bit-exact. The transcoding delay is lower than the ETSI requirement.

A comparison of the half-rate and full-rate compression algorithms is discussed. Both algorithms offer near toll speech quality comparable or better than analogue cellular networks. The half-rate compression requires more computationally intensive operations and therefore a more powerful processor will be needed due to the complexity of the code. Hence the cost of the implementation of half-rate codec will be considerably higher than full-rate.

A description of multirate signal processing and its application on speech (SBC) and speech/audio (MPEG) has been given. An investigation into the possibility of combining multirate filtering and GSM full-rate speech algorithm. The results showed that multirate signal processing cannot be directly applied GSM full-rate speech compression since this method requires more processing power, causing longer coding delay but did not appreciably improve the bit rate. In order to achieve a lower bit rate, the GSM full-rate mathematical algorithm can be used instead of the standardised ETSI recommendation. Some changes including the number of quantisation bits has to be made before the application of multirate signal processing and a new standard will be required.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Introduction to Speech Coding

Speech coders are algorithms that compress digital representation of speech signals to minimise the numbers of bits required to represent those signals. They achieve this by taking advantage, to varying degrees, of redundancies in the speech signals, and the digital storage of speech signals. In the first category are such applications as digital communications systems, mobile radio, cellular telephony, and secure voice systems. In this category, channel conditions, delay and data rate are important issues. In the second category are such applications as digital answering machines and voice response systems. In this category, speech quality requirements are generally the overriding concerns.

The most important issues in selecting a speech coder are those of perceived quality, coder complexity, and bit rate or data rate; the optimal solution will involve some balance between these features [KON95]. In general, to achieve high speech quality at low bit-rates will require a coder of high complexity; a less complex coder, which may be implemented at lower processor costs, would mandate a higher bit-rate or a reduction in speech quality.

In telecommunication systems, the design and subjective test of speech coders has been extremely difficult. A speech coder is evaluated based on the amount of compression achievable; the sound quality of the reconstructed speech; the complexity of the algorithm, signifying how powerful, and therefore expensive, a computer processor is required upon which to run it; and the delay introduced, that is, the time delay between when the speaker utters a word and the listener hears it. The importance of each factor is dependent upon the application. For example, in storage systems such as answering machines, the delay is inconsequential since the message will not be listened for a while. However, in a telephone system, large delays will make conversation particularly difficult for the users. To make speech coding practical, implementations must consume little power and provide tolerable, if not excellent speech quality. The goal of all speech coding systems is to transmit speech with the highest possible quality using the least possible channel capacity. This has to be accomplished while maintaining certain required levels of complexity of implementation and communication delay. In general, there is a positive correlation between coder bit rate efficiency and the algorithmic complexity required to achieve it. The more complex an algorithm is, the more its processing delay and cost of implementation. A balance needs to be struck between these conflicting factors, and the aim of all speech processing

developments is to shift the balance towards lower bit rates while maintaining acceptable speech quality.

1.2 Low Bit Rate Speech Coding for Telecommunications

Low speech rate speech coding attempts to provide toll-quality speech at a minimum bit rate for digital transmission or storage. In the 1960s, A-Law coding was selected for the first 24-channel pulse code modulation (PCM) systems and the 64 kbps PCM was one of the earliest speech coding scheme. Research into more complex lower bit rate coding schemes was initially inhibited by practical implementation considerations imposed by the semiconductor technology of the day. As a consequence, research into sophisticated low bit rate algorithms did not gather the momentum until the 1970s. It was not until the 1980s that the first world-wide lower bit rate coding standard was achieved [RAP96]. This standard was the International Telegraph and Telephone Consultative Committee (CCITT) G.721 Recommendation for 32 kbps Adaptive Differential PCM (ADPCM).

Since that time, the major advances made in microelectronics and digital signal processor technology have spurred research into increasingly complex speech coding schemes. Sophisticated speech coding techniques are now commercially viable, and can be implemented at low cost for more affordable mobile, computer and audio-visual terminals. Organisations such as CCITT, International Telecommunications Union (ITU), European Telecommunications Standards Institute (ETSI), Universal Mobile Telecommunication System (UMTS), International Standard Organisation (ISO) and International Maritime Satellite Organisation (Inmarsat) have achieved notable successes with in-house speech coding technology.

There was a period in the early 1980s when the need for further speech coding development was brought into question. With rapid migration to optical fibre transmission that was occurring in world networks and the potentially vast bandwidth offered by full exploitation of this technology, there is still a need for higher speech compression. However, the limited available radio bandwidth makes speech coding essential for the provision of viable services in radio communication systems. Furthermore, although the introduction of optical fibre transmission has proceeded apace, bandwidth constraints remain a significant issue in areas of the fixed network. This is particularly true for international links, whether they are satellite or cable, where speech compression is increasingly employed [CAR86]. Competition from other network operators has also been an important driver for speech coding technology, since more efficient use of the transmission medium permits lower tariffs, leading to increased market share. Another area where speech coding plays a vital role is the new media services. In these applications, the reduced data rate

required for the transmission of the audio component maximises the bandwidth available to the visual element.

The position today is that a number of low rate speech coding techniques have already been adopted as international standards for various network applications. For many new network developments such as global virtual private networks, third generation mobile, cellular satellite and even the asynchronous transfer mode (ATM) networks, it is no longer a question of whether speech compression should be used, but which speech coding technology provides the target speech quality at lowest cost.

1.3 Objective of Thesis

This thesis extends the background theory of speech and major speech coding schemes used in existing telecommunication systems to an implementation of GSM full-rate speech compression on a RISC DSP and a multirate application for speech coding. Chapter 2 describes the production and characteristics of human speech. The sampling theorem, quantisation schemes and modelling tools used in speech coding are discussed. This is followed by a description of the procedure of short-term and long-term prediction of speech.

Chapter 3 discusses the requirements speech compression algorithm and describes the current speech coding schemes under three categories: waveform coding, vocoding and hybrid coding, giving their complexities and qualities at various bit rates.

Chapter 4 presents an overview of GSM full-rate speech algorithm which is one of the existing low bit rate digital speech coding standards. An investigation into real time implementation of the algorithm on the new generation processor for parameterised DSP called GEPARD by Austria Mikro Systeme International (AMS) is discussed.

Chapter 5 gives a brief review of the new GSM half-rate speech algorithm which uses Vector Sum Linear Excited Prediction (VSELP). A comparison to the full-rate algorithm on speech quality, complexity and cost of implementation is made.

Chapter 6 describes multirate digital signal processing and its applications on speech. A brief explanation of Subband Coding and the three layers of MPEG audio compression which use the multirate technique will be given. An experiment of the investigation of combining multirate filtering and GSM full-rate compression is described. The results is discussed and compared with conventional GSM full-rate.

2 Speech Signal Analysis

2.1 Introduction

The frequency of speech waves is measured in Hertz (Hz), or number of cycles per second. The human ear can typically perceive frequencies between 20 Hz to 20 kHz, although many people have rather limited hearing above 15 kHz [TIP91]. Human voice can produce frequencies between 40 Hz and 4 kHz. These limits are important factors as most systems are designed to produce speech quality which encompasses a much smaller range of frequencies than the range perceptible to humans.

The speech signal has been studied for various applications by many researchers for many years. Some studies break down the speech signal into its smallest portions, called phonemes. However, in this thesis speech signal will be described in terms of its general characteristics. The traditional vocoders which have been in use for many years classify the input speech signal either as voiced or unvoiced. A voiced speech segment is known by its relatively high energy content, but more importantly it contains periodicity which is called the pitch of voiced speech. The unvoiced part of speech, on the other hand, looks more like random noise with no periodicity. However, there are some parts of speech that are neither voiced nor unvoiced, but a mixture of the two. These are usually called the transition regions, where there is a change either from the voiced to unvoiced or unvoiced to voiced [ROB97].

2.2 Speech Production

When a voiced speech is pronounced, air is pushed out from the lungs, opening a gap between the two vocal folds, which is the glottis. Tension in the vocal tracts increases until, pulled by the muscles and a Bernoulli force from the stream of air, they close. After the folds have closed, air from the lungs again forces the glottis open, and the cycle repeats, between 50 to 500 times per second, depending on the physical construction of the larynx and how strong you pull on the vocal tracts.

For voiceless consonants, air is blown past some obstacle in the mouth. During transitions, and for some mixed phonemes, the same air stream is used twice, first to make a low-frequency hum with the vocal tracts, then to make a high-frequency, noisy hiss in the mouth. Before vibrations from a person's glottis reach the ear, those vibrations pass through

the throat, over the tongue, against the roof of the mouth, and out through the teeth and lips [MCE95].

The space that a sound wave passes through changes it. Parts of one wave are reflected and mixed with the next oncoming wave, changing the frequency spectrum of the sound. Every vowel has three to five typical frequencies, or formants, that distinguish it from others. By changing the interior shape of the mouth, reflections that amplify the formant frequencies of the phoneme being spoken can be created.

2.3 Characteristic of Speech Signals

Speech waveforms have a number of useful properties that can be exploited when designing efficient coders. Some of the properties that are most often utilised in coder design include the non-uniform probability distribution of speech samples, the non-flat nature of the speech spectra, the existence of voiced and unvoiced segments in speech, and the quasi-periodicity of voiced speech signals, produced by opening and closing the glottis as the air passes through the vocal tract [TEM96]. The most basic property of speech waveforms that is exploited by all speech coders is that they are bandlimited by passing through a low pass filter. A finite bandwidth means that it can be sampled at a finite rate and reconstructed completely from its input signal, provided that the sampling frequency is greater than twice the highest frequency component in the filtered signal. The characteristics of speech can be quantified in a number of ways as shown in the following sections.

2.3.1 Probability Density Function (PDF)

The non-uniform probability density function of speech amplitudes is perhaps the next most exploited property of speech. The PDF [RAP96] of a speech signal is in general characterised by a very high probability of near-zero amplitudes, a significant probability of very high amplitudes, and a monotonically decreasing function of amplitudes between these extremes. The exact distribution, however, depends on the input bandwidth and recording conditions. The two-sided exponential function given in (2.1) provides a good approximation to the long-term PDF of telephone quality speech signals.

$$p(x)=\frac{1}{\sqrt{2\sigma_x}}\exp\left(-\sqrt{2}|x|/\sigma_x\right) \tag{2.1}$$

This PDF shows a distinct peak at zero which is due to the existence of frequent pauses and low level speech segments. Short-term PDFs of speech segments are also single-peaked functions and are usually approximated as a Gaussian distribution. A plot of the PDF of speech is shown in Figure 2.1.

Non-uniform quantisers, including the vector quantisers, attempt to match the distribution of quantisation levels to that of the PDF of the input speech signal by allocating more

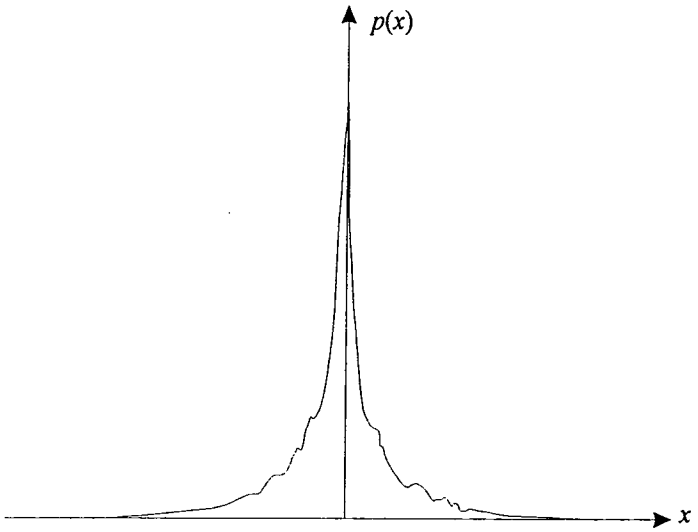


Figure 2.1 Probability density function of speech.

quantisation levels in regions of high probability and fewer levels in regions where the probability is low.

2.3.2 Autocorrelation Function (ACF)

The autocorrelation function [TEM96] [ROB97] is another very useful property of speech signals as there exists considerable correlation between adjacent samples of a segment of speech. This implies that in every sample of speech, there is a large component that is easily predicted from the value of the previous samples with a small random error. All differential and predictive coding schemes are based on exploiting this property.

The ACF gives a quantitative measure of the closeness or similarity between segments of samples of a speech signal as a function of their time separation. This function is mathematically defined as

$$C(k)=\frac{1}{N}\sum_{n=0}^{N-|k|-1}x(n)x(n+|k|) \tag{2.2}$$

where $x(k)$ represents the kth speech sample. The autocorrelation function is often normalised to the variance of the speech signal and hence is constrained to have values in the range $\{-1,1\}$ with $C(0)=1$. Typical signals have an adjacent sample correlation, $C(1)$, as high as 0.85 to 0.9.

2.3.3 Power Spectral Density Function (PSD)

The non-flat characteristic of the power spectral density [KON95] [RAP96] of speech makes it possible to obtain significant compression by coding speech in the frequency domain. The non-flat nature of the PSD is a frequency domain manifestation of the non-zero

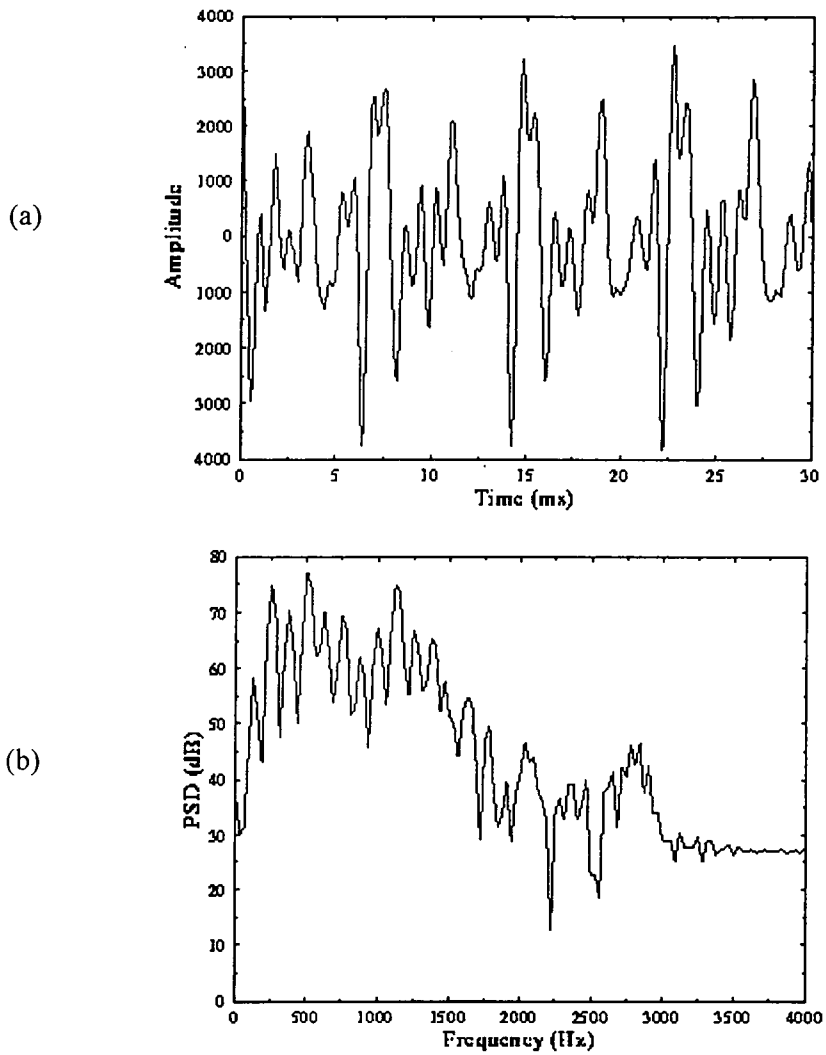


Figure 2.2 30 ms duration of speech. (a) A typical segment. (b) Power spectral density for the segment.

autocorrelation property. Figure 2.2 shows a typical segment of speech and its power spectral density. Typical long-term averaged PSD's of speech show that high frequency components contribute very little to the total speech energy. This indicates that coding speech separately in different frequency bands can lead to significant coding gain. However, the high frequency components cannot be ignored and need to be adequately represented in the coding system.

The qualitative measure of the theoretical maximum coding gain that can be obtained by exploiting the non-flat characteristics of the speech spectra is given by the spectral flatness measure (SFM). The SFM is defined as the ratio of the arithmetic to geometric mean of the samples of the PSD taken at uniform intervals in frequency. Mathematically,

$$SFM = \frac{\left[\frac{1}{N}\sum_{k=1}^{N} S_k^2\right]}{\left[\prod_{k=1}^{N} S_k^2\right]^{\frac{1}{N}}} \tag{2.3}$$

where, S_k is the kth frequency sample of the PSD of a speech signal. Typically, speech signals have a long-term SFM value of 8 and a short-time SFM value varying widely between 2 and 500.

2.4 Sampling Theory

An analogue speech signal is continuous in time. Before it can be processed by digital hardware it must be converted to a signal that is discrete in time. Sampling [MARV93] [PAN93] is a process that converts a continuous time signal by measuring the signal at

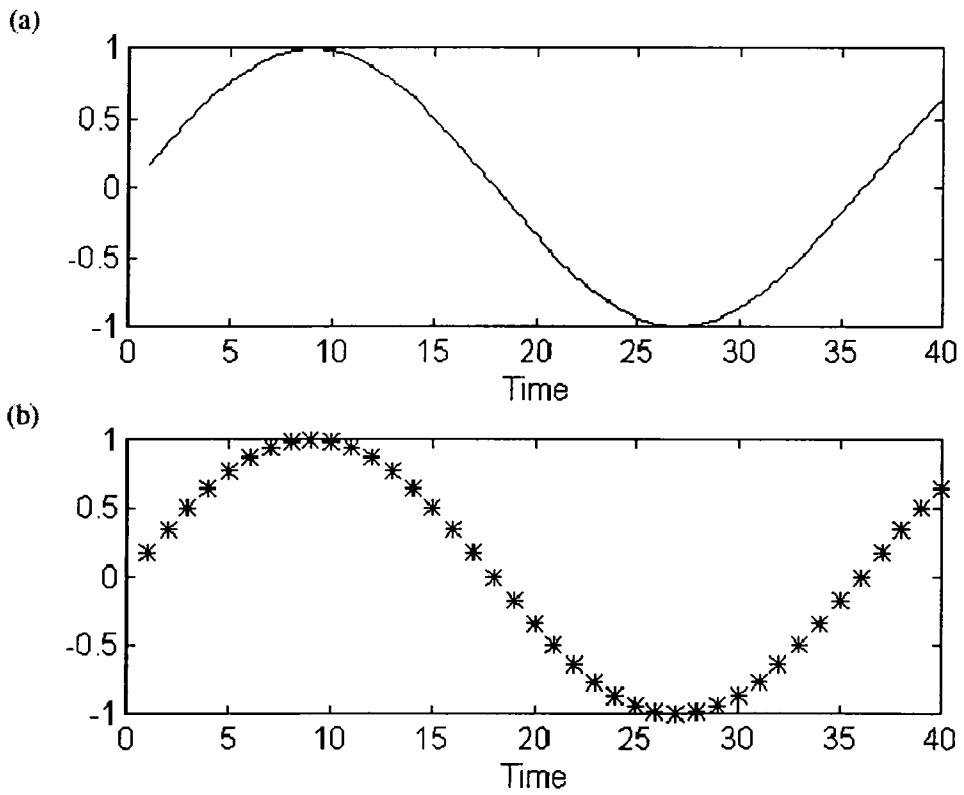


Figure 2.3 A sinusoidal signal. (a) the continuous signal and (b) the sampled signal.

periodic instants in time. Figure 2.3 shows the effect of sampling on a sinusoidal signal. It is clear that as the sampling rate increases the sampled signal approximates the continuous signal more closely. The sampled signal can be represented by

$$s(n) = s_a(nT) \tag{2.4}$$

where $-\infty < n < \infty$, n is the integer, s_a is the analogue signal, and T is the sampling time or the time difference between any two adjacent samples, which is determined by the highest frequency in the input signal.

The sampling theorem states that if a signal $s_a(t)$ has a band limited Fourier transform $S_a(j\omega)$, given by

$$S_a(j\omega) = \int_{-\infty}^{\infty} s_a(t) e^{-j\omega t} dt \tag{2.5}$$

such that $S_a(j\omega) = 0$ for $\omega \geq 2\pi f_N$, then the signal can be reconstructed from its sampled version if $T \leq 1/2f_N$. f_N is called the Nyquist frequency.

The effect of sampling is shown is Figure 2.4. As can be seen from Figures 2.4(b) and 2.4(c), the band limited Fourier transform of the analogue signal shown in Figure 2.4(a) is

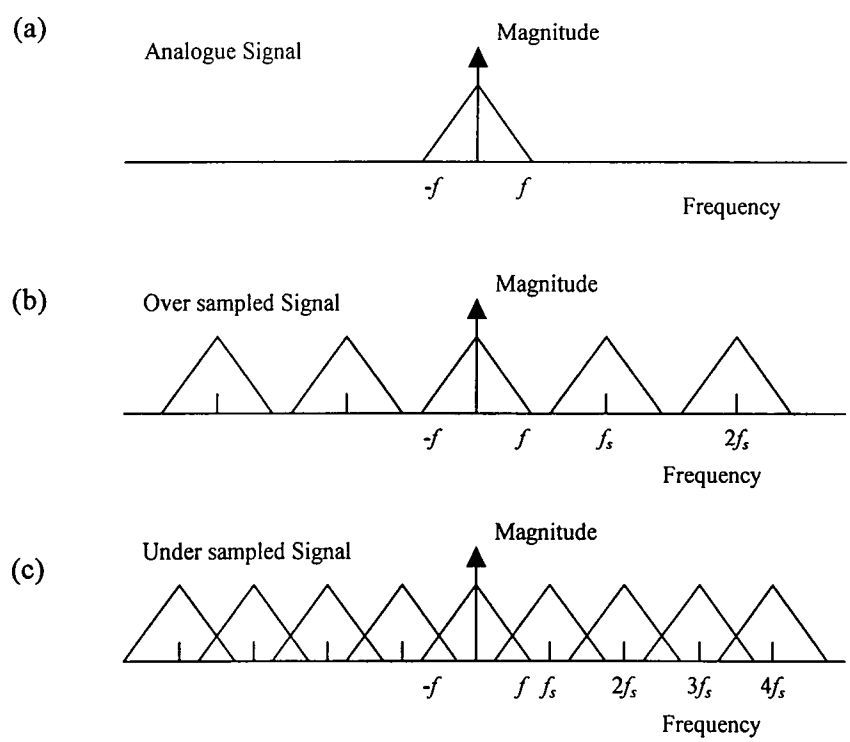


Figure 2.4 The effects of sampling. (a) original signal spectrum, (b) over sampled, and (c) under sampled signal spectra.

duplicated at every multiple of the sampling frequency. This is because the Fourier transform of the sampled signal is evaluated at multiples of the sampling frequency, which forms the relationship

$$S\left(e^{j\omega T}\right)=\frac{1}{T}\sum S_a\left(j\omega + j2\pi n/T\right). \tag{2.6}$$

Therefore, if the sampling frequency is less than twice the Nyquist frequency, the spectra of two adjacent multiples of the sampling frequencies will overlap. The distortion caused by high frequencies overlapping low frequencies is called aliasing. To avoid aliasing distortion, either the input analogue signal has to be band limited to a maximum of half the sampling frequency, or the sampling frequency has to be increased to at least twice the highest frequency in the analogue signal.

The Nyquist sampling theorem states that a signal may be reconstructed if the sampling rate is greater than or equal to twice the frequency of the highest frequency component of the signal. In telecommunication networks, the analogue speech signal is band limited to 300 - 3400 Hz and sampled at 8000 Hz [SPA94].

2.5 Short-Term Spectral Analysis

In some speech coding schemes the frequency domain representation of the speech signal is necessary. For this purpose, the short-term Fourier transform [KON95] [ROB97] is very useful. The short-term spectral transform is also important for looking at a segment of the speech signal and to determine features that are not obvious from the time domain representation.

The short-term Fourier transform plays a fundamental role in the frequency domain analysis of the speech signal. It is used to represent the time-varying properties of the speech waveform in the frequency domain. A useful definition of the time-dependent Fourier transform is

$$S_k\left(e^{j\omega}\right)=\sum_{n=-\infty}^{\infty} w(k-n)s(n)e^{-j\omega n} \tag{2.7}$$

where $w(k - n)$ is a real window sequence used to isolate the portion of the input signal that will be analysed at a particular time index, k. During the analysis of speech signals, the shape and length of the window can affect the frequency representation of speech. Various types of windows have been studied by various researchers producing window shapes and characteristics suitable for various applications.

2.5.1 Role of Windows

The window, $w(n)$, determines the portion of the speech signal that is to be processed by zeroing out the signal outside the region of interest. The ideal window frequency response

has a very narrow main lobe which increases the resolution, and no side lobes (or frequency leakage). Since such a window is not possible in practice, a compromise is usually selected for each specific application. There are many possible windows (e.g. Rectangular, Hanning, Hamming, Blackman, Kaiser, etc.).

The rectangular window has the highest frequency resolution due to the narrowest main lobe, but has the largest frequency leakage. On the other hand, the Blackman window has the lowest resolution and the smallest frequency leakage [STR89]. The effect of these windows on the time-dependent Fourier representation of speech is discussed by comparing the rectangular window and the Hamming window.

The effects using Hamming and rectangular windows for speech spectral analysis are shown in Figure 2.5, where in each figure, plot (a) show the windowed signal $s(n)w(k - n)$ using the rectangular window and (b) shows the log magnitude of its Fourier transform,

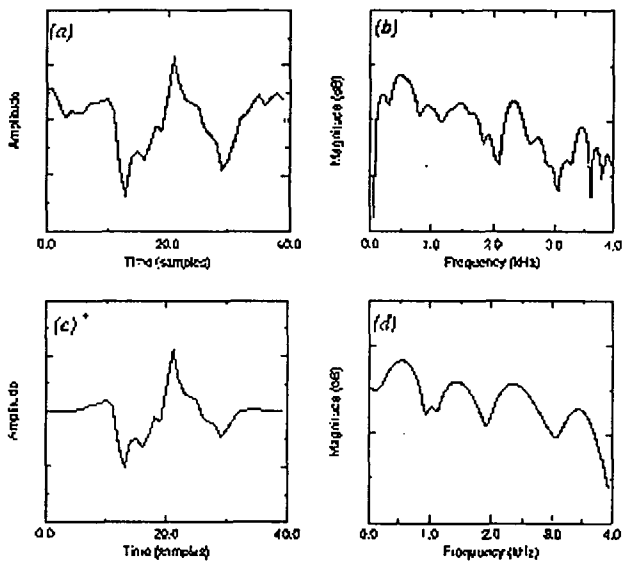


Figure 2.5 Effects of window types on voiced speech with 40 samples window length. (a) and (b) are time and frequency plots of speech using a rectangular window, (c) and (d) are time and frequency plots of speech using a Hamming window.

$S_k(\omega)$. Similarly, plots (c) and (d) show the windowed signal using the Hamming window, and the corresponding log magnitude spectrum. Figure 2.5 shows the results for a window duration of 40 samples (5 ms for 8 kHz sample rate) for a section of voiced speech. When compared, the periodicity of the signal is clearly seen Figure 2.5(b) as well as in Figure 2.5(d). However, the harmonic peaks at multiples of the fundamental frequency are narrower and sharper in rectangular windowed speech. The formant structure which consists of a strong first peak at about 500 Hz, and three broader peaks at about 1350 Hz, 2300 Hz

and 3400 Hz, as well as showing a tendency to fall off at higher frequencies due to the low-pass nature of the glottal pulse spectrum is also noticeable in Figures 2.5(b) and 2.5(d).

Although both Figure 2.5(b) and 2.5(d) show considerable overall similarity in terms of the pitch harmonics, formant structure and gross spectral shape, the pitch harmonics of Figure 2.5(b) are sharper, due to a greater frequency resolution of the rectangular window relative to that produced by the Hamming window in Figure 2.5(d). However, due to the high frequency leakage of the rectangular window produced by its larger side lobes, the windowed speech looks more noisy. This undesirable high frequency leakage between adjacent harmonics tends to offset the benefits of the flat time domain response (greater frequency resolution) of the rectangular window. As a result, rectangular windows are not usually used in speech spectral analysis.

2.6 Quantisation Techniques

Quantisation is the conversion of a discrete-time continuous-valued signal into a discrete-time, discrete-valued signal. The value of each signal sample is represented by a value selected from a finite set of possible values. The difference between the unquantised input and the quantised output is called the quantisation error (or noise) and it is desirable to minimise the perceived magnitude of this error. In order to achieve this objective several quantisation techniques can be used. All the quantisation schemes can be made to adapt to the input waveform's statistics so that the quantiser will always by locally optimum and provide the highest possible quality with the lowest possible bit rate.

2.6.1 Linear Quantisation

Linear or uniform quantisers [SPA94] are those in which the distances between all the reconstruction levels are the same. They make no assumptions about the nature of the signal being quantised. For this reason they normally do not give the best perceived performance. However, they are usually the simplest and the cheapest to implement. To quantise telephone speech a 13-bit uniform quantiser is necessary to provide toll quality speech.

2.6.2 Logarithmic Quantisation

Speech signals can have a dynamic range in excess of 60 dB so a large number of reconstruction levels are necessary for the uniform quantiser to attain high quality speech. However quantiser resolution is more important for the low amplitude parts of the signal than for the large amplitude signals. Therefore it is obvious that the uniform quantiser is wasteful of reconstruction levels and, hence, bandwidth. The situation could be improved if

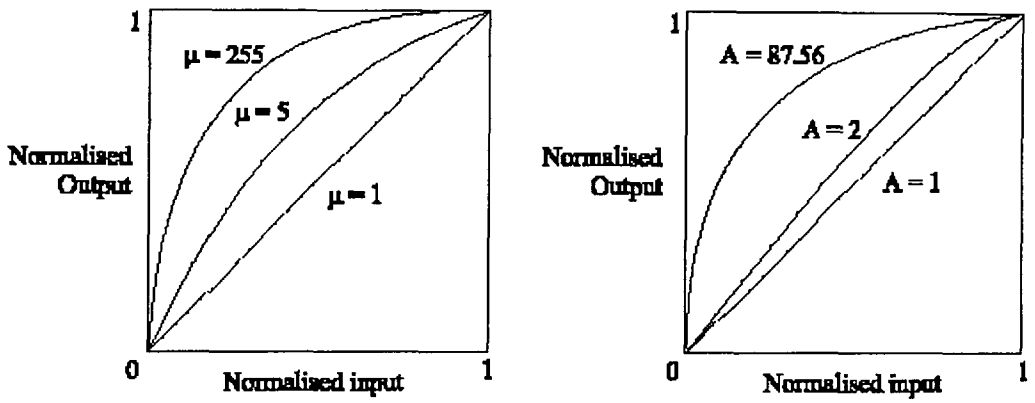


Figure 2.6 Companding characteristics. (a) μ-law and (b) A-law.

the distance between the reconstruction levels increased as the amplitude of the signal increased. This processing is called *companding* [RAP96].

Two very popular companding characteristics are A-law and μ-law companding. They are very similar and their transfer characteristics are shown in Figure 2.6. The A-Law compression is defined by

$$AL(x) = \begin{cases} \dfrac{Ax}{1+\log_{10}(A)} & \text{for } 0 \le x \le \dfrac{1}{A} \\ \dfrac{1+\log_{10}(Ax)}{1+\log_{10}(A)} & \text{for } \dfrac{1}{A} \le x \le 1 \end{cases} \tag{2.8}$$

where A is the compression parameter with typical values of 86 for 7 bit (North America PCM) and 87.56 for 8 bit (European PCM) speech quantisers.

The μ-Law compression, on the other hand, is defined by

$$\mu L(x) = \text{sign}(x)\frac{V_0 \log_{10}\left[1+\dfrac{\mu|x|}{V_0}\right]}{\log_{10}[1+\mu]} \tag{2.9}$$

where V_0 is given by $V_0 = L\sigma_x$, in which L is the loading factor and σ_x is the rms value of the input speech signal.

A typical value of compression factor μ is 255. The above expression show that the A-Law is a combination of logarithmic curve used for large amplitudes, while the small amplitudes the curve becomes linear. The μ-Law, on the other hand, is not exactly linear or logarithmic in any range, but it is approximately linear for small and logarithmic for large amplitudes [KON95].

2.6.3 Non-Uniform Quantisation

The problem with uniform quantisation is that as the signal amplitude decreases the signal to noise ratio also decreases. This problem is partially solved by the logarithmic quantiser. However is the probability distribution function (PDF) of the input is known then the reconstruction levels can be matched to the PDFs so that the mean squared quantisation error is minimised [RAP96]. This means that most of the reconstruction levels occur in the vicinity of the most likely inputs and has the effect of minimising the perceived quantisation error.

In practice an estimate of the PDF can be used to design the quantisers. This can be obtained from a large library of the data to be quantised. Iterative techniques can then be used to determine the reconstruction levels using this library.

2.6.4 Vector Quantisation

In the previous methods each sample is quantised independently from its neighbouring samples. Shannon [SHA48] proposed that the information rate of a continuous source could be measured in terms of some specified distortion criterion. Appropriate digital source coding then allows efficient information transmission over a channel whose capacity equals the rate in question. This proposal has subsequently been developed into the field known as rate-distortion theory which states that there exists a mapping from a source waveform to output code words such that for a given distortion D, $R(D)$ bits per sample are sufficient to reconstruct the waveform with an average distortion arbitrarily close to D. Therefore, the actual rate R has to be greater than $R(D)$. The rate-distortion function $R(D)$ represents a fundamental limit on the achievable rate for a given distortion. Scalar quantisers do not achieve performance close to this information theoretical limit. The theorem predicts that better performance can be achieved by coding many samples at a time instead of one sample.

Vector quantisation [SPA94] [KON95] is a delayed-decision coding technique which maps a group of input samples, called a vector, to a codebook index. A codebook is set up consisting of a finite set of vectors covering the entire anticipated range of values. In each quantising interval, the codebook is searched and the index of the entry that gives the best match to the input frame is selected. Vector quantisers can yield better performance even when the samples are independent of one another. Performance is greatly enhanced is there is a strong correlation between the samples in the group. A block diagram of a simple vector quantiser is shown in Figure 2.7.

Let $\mathbf{x} = [x_1, x_2, ..., x_N]^T$ be an N dimensional vector with real valued, continuous-amplitude random varying components x_k, $1 \leq k \leq N$ (the superscript T denotes transpose), in a vector

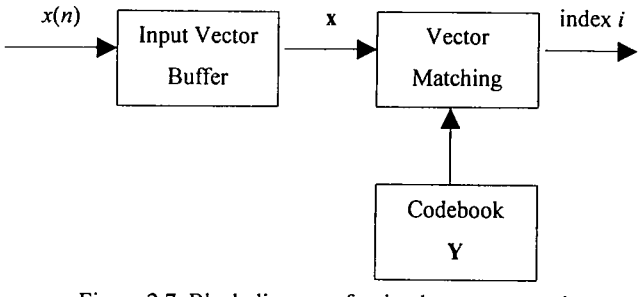


Figure 2.7 Block diagram of a simple vector quantiser.

quantisation this vector is matched with another real-valued, discrete-amplitude, N dimensional vector **y**. Thus, **x** is quantised as **y**, hence **y** is used to represent **x**. Usually, **y** is chosen from a finite set of values $\mathbf{Y} = \mathbf{y}_i$, $1 \le i \le L$, where $\mathbf{y}_i = [y_1, y_2, ..., y_N]^T$. The set **Y** is called a codebook, L is the size of the codebook and $\mathbf{y}_i$ are the set of codebook vectors.

The rate, R, of the vector quantiser is defined as

$$R = \frac{\log_2 n}{N} \text{ bits per sample.} \tag{2.10}$$

Where n is the size of the vector quantisation codebook. R may also take fractional values. All the quantisation principles used in scalar quantisation apply to vector quantisation as a straightforward extension. Instead of quantisation levels, quantisation vectors are used, and distortion is measured as the squared Euclidean distance between the quantisation vector and the input vector.

Vector quantisation is very efficient at low bit rates, where R = 0.5 bits/sample or less. This is because when R is small, a large vector dimension N can be used and then vector quantisation codebook can still have a reasonable size, 2^{RN}. Use of large dimensions brings out the inherent capability of vector quantisation to exploit the redundancies in the components of the vector being quantised. Vector quantisation is a computationally intensive operation and hence is not often used to code speech signals directly. It is also more sensitive to transmission error compared with scalar quantisation. However, it is used in many speech coding systems to quantise the speech analysis parameters like the linear prediction coefficients and the spectral coefficients.

2.7 Linear Predictive Modelling of Speech

One of the most powerful speech analysis methods is that of Linear Predictive Coding, or LPC analysis [MAK95] as it is commonly referred to. In LPC analysis the short-term correlations between speech samples are modelled and removed by a very efficient short

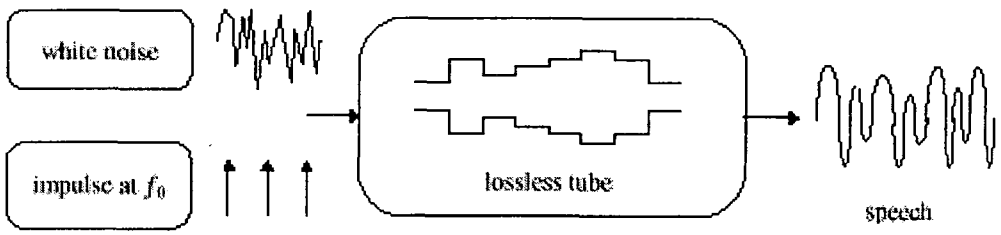


Figure 2.8 The lossless tube model of speech production.

order filter. Another equally powerful and related method is pitch prediction. In pitch prediction, the long-term correlation of speech samples is modelled.

LPC analysis assumes that the vocal tract is a lossless tube which can be described by an all pole infinite impulse response (IIR) filter with a transfer function described by

$$H(z)=\frac{G}{1+\sum_{k=1}^{M} b_k z^{-k}} \tag{2.11}$$

where G is a gain of the filter and z^{-1} represents a unit delay operation. This lossless tube model of speech production is shown in Figure 2.8. In other words, each speech sample is

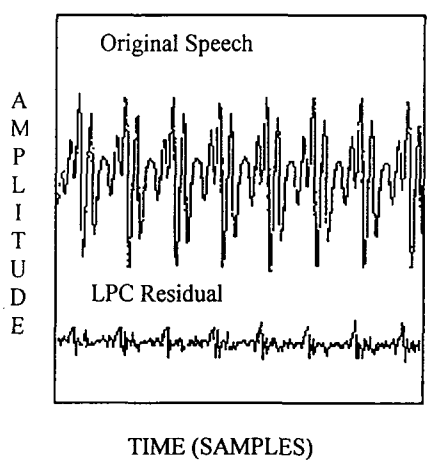


Figure 2.9 Waveform plots of original and LPC inverse filtered speech.

assumed to be a linear combination of the previous samples. The excitation to the linear predictive filter is either a pulse at the pitch frequency or random white noise depending on whether the speech segment is voiced or unvoiced. The coefficients of this filter are calculated to minimise the error between the prediction and the actual sample [TEM96].

In the LPC filter a block of about 20 ms of speech is stored and analysed to determined the predictor coefficients. The method is described in the next section. These coefficients are then quantised and transmitted to the receiver. This speech is then passed through the

inverse of the vocal tract filter to obtain the prediction error or residual. The effect of the predictor is to remove the correlation between adjacent samples. This makes determining the pitch period of the speech much easier by making the long term correlation more visible.

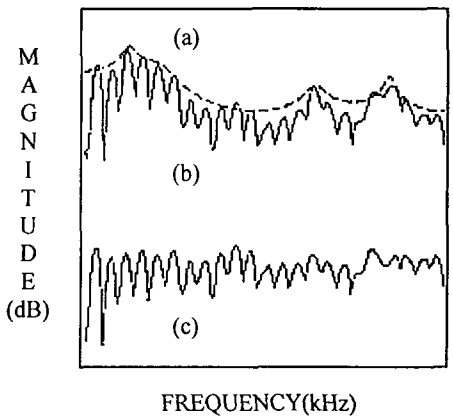


Figure 2.10 Spectra of original and LPC inverse filtered speech. (a) original speech envelope, (b) original speech spectrum, and (c) LPC residual spectrum.

Hence a more reliable voiced/unvoiced decision can be made using this residual. The plots of the waveform and spectrum of the original and LPC inverse filtered speech are shown in Figure 2.9 and 2.10.

LPC analysis is very popular because the all pole model of the vocal tract works very well. It can be used to achieve highly intelligible speech at bit rate as low as 2.4 kbps [SPA94]. The algorithm is explained in the next section.

2.7.1 Determination of Predictor Coefficients

The linear predictive coder uses a weighted sum of p past samples to estimate the present sample,

$$s_n = \sum_{k=1}^{p} a_k s_{n-k} + e_n \tag{2.12}$$

where p is typically in the range of 8 - 14. Using this technique, the current sample s_n can be written as a linear sum of the immediately preceding samples s_{n-k}

where, e_n is the prediction error (residual). The predictor coefficients are calculated to minimise the average energy E in the error signal that represents the difference between the predicted and actual speech amplitude.

$$E = \sum_{n=1}^{N} e_n^2 = \sum_{n=1}^{N} \left(\sum_{k=0}^{p} a_k s_{n-k} \right)^2 \tag{2.13}$$

where a_0 = -1. Typically the error is computed for a time window of 20 ms which corresponds to a value of N = 160. To minimise E with respect to a_m, it is required to set the partial derivatives equal to zero.

$$\frac{\partial E}{\partial a_m} = \sum_{n=1}^{N} 2 s_{n-m} \sum_{k=0}^{p} a_k s_{n-k} = 0 \tag{2.14a}$$

$$= \sum_{k=0}^{p} \sum_{n=1}^{N} s_{n-m} s_{n-k} a_k = 0 \tag{2.14b}$$

The inner summation can be recognised as the correlation coefficient C_{rm} and hence the above equation can be rewritten as

$$\sum_{k=0}^{p} C_{mk} a_k = 0 \tag{2.15}$$

After determining the correlation the correlation coefficients C_{rm}, (2.15) can be used to determine the predictor coefficients [ROB97]. (2.15) is often expressed in matrix notation and the prediction coefficients calculated using matrix inversion. A number of algorithms have been developed to speed up the calculation of predictor coefficients, e.g. the autocorrelation method and the covariance method. Normally the predictor coefficients are not coded directly, as they would require 8 bits to 10 its per coefficients for accurate representation. The accuracy requirements are lessened by transmitting the closely related reflection coefficients which have a smaller dynamic range. These reflection coefficients can be adequately represented by 6 bits per coefficients. Thus, for a 10th order predictor, the total number of bits assigned to the model parameters per frame is 72, which includes 5 bits for a gain parameter and 6 bits for the pitch period. If the parameters are estimated every 15 ms to 30 ms, the resulting bit rate is in the range of 2400 bps to 4800 bps. The coding of the reflection coefficient can be further improved by performing a non-linear transformation of the coefficients prior to coding. This non-linear transformation reduces the sensitivity of the reflection coefficients to quantisation errors. This is normally done through a log-area ratio (LAR) transform which performs an inverse hyperbolic mapping of the reflection coefficients, $R_n(k)$

$$LAR_n(k) = \tanh^{-1}(R_n(k)) \log\left[\frac{1 + R_n(k)}{1 - R_n(k)}\right] \tag{2.16}$$

2.8 Pitch Prediction

There are still considerable variations in the spectrum after the removal of the spectral envelope in the signal spectrum in LPC analysis. The long-term correlations, especially

during voiced regions, still exist between samples in the residual signal. In order to removal the periodic structure of the residual or excitation signal, a second stage of prediction is required [CHA96]. The objective of this second stage is to spectrally flatten the signal, i.e. to remove the fine structure. Unlike the LPC analysis, it exploits correlation between the speech samples that are one pitch or multiple pitch period away. For this reason, the pitch prediction filter is usually called the long-term prediction (LTP) and the filter delay is called the lag.

2.8.1 Pitch Predictor Formulation

The aim of pitch prediction is to model the long-term correlation left in the speech residual after LPC inverse filtering such that when the model parameters are used in a filter, it will

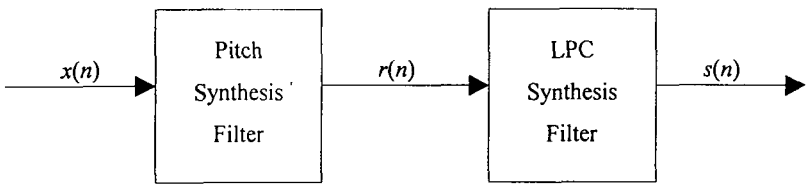


Figure 2.11 A typical pitch-LPC formulation model.

remove the long-term correlation as much as possible. The Long-Term Predictor (LTP) can be interpreted as

$$P(z)=\frac{1}{1-\sum_{j=-I}^{I} b_j z^{-(j+T)}} \tag{2.17}$$

where T is the pitch period, and b_j are the pitch gain coefficients which reflect the amount of correlation between the distant samples. Referring to Figure 2.11, the combined analysis model can be represented by a time domain difference equation

$$s(n)=Gx(n)+\sum_{j=-1}^{I} b_j r(n-T-j)+\sum_{j=1}^{p} a_j s(n-j) \tag{2.18}$$

where $r(n)$ is the past excitation signal. Following a similar procedure to that of the LPC analysis, the aim is to determine estimates (β_j, τ_j, α_j) of the model parameters (b_j, T, a_j). Then the prediction error is given by

$$e(n)=s(n)-\sum_{j=-1}^{I} \beta_j r(n-\tau-j)-\sum_{j=1}^{p} \alpha_j s(n-j) \tag{2.19}$$

2.9 Summary

This chapter describes background information characteristics of human speech. The DSP algorithms used in speech coding for sampling, quantisation and modelling are discussed and formulated. The short-term and long-term prediction of speech is described. The application of the algorithms and description of current speech coding schemes can be found in the next chapter.

3 Speech Coding Strategies and Algorithms

3.1 Introduction

Digital encoding of voiceband speech has been a topic of research for many years, and as a result of this intense activity, many strategies and approaches have been developed for speech coding. As these strategies and techniques matured, standardisation followed with specific application targets. In this chapter, a brief review of speech coding techniques will be presented. The requirements of the current generation of encoding standards will also be discussed. The motivation behind the review is to highlight the different advantages and disadvantages of various techniques employed in the past and present. The success of the different coding techniques is revealed in the description of the many coding standards currently, and soon to be, in active operation, ranging from 64 kbps down to 2 kbps.

3.2 Algorithms Objectives and Requirements

Table 3.1 Applications and networking requirments for speech coding. (Async. = asynchronous PCM transcoding, Sync. = synchronous PCM transcoding, DCM = digital circuit multiplication)

Application	Codec Delay	PCM Transcoding	Voiceband Data	Other Non-Voice	Minimum Speech Quality
Land DMR + Portables	70 ms	2 Async.	--	Tones	Analogue (900 MHz)
Maritime Satellite Systems	60 - 80 ms	2 Async.	Up to 2.4 kbps	Tones	Companded FM (6-bit PCM)
DCM Equipment	40 - 80 ms	2 Async.	Yes	Tones	6 - 7 bit PCM
ISDN	--	4 Sync.	No	No	6 - 7 bit PCM
Digital Leased Lines	70 ms	--	--	Tones	7 bit PCM
Voice Store Forward Systems	--	--	No	No	6 - 7 bit PCM
Voice Messages for Announcement	--	--	No	No	Speech Intel.

The design and coding capacity of a particular algorithm is often dictated by the target application. Therefore, in the design of an algorithm the relative weighting of the influencing factors requires careful consideration in order to obtain a balanced compromise between the often conflicting objectives. Some of the factors which influence the choice of algorithm for foreseeable network applications are shown in Table 3.1.

3.2.1 Quality and Capacity

Speech quality and bit rate are two factors that directly conflict with each other. The lower the bit rate of the speech coder, i.e. higher signal compression, the more the quality inevitably suffers (vocoders). For systems that connect to the Public Switched Telephone Network (PSTN) and associated systems, the quality requirements are strict and must conform to constraints and guidelines imposed by the relevant regulatory bodies, e.g. ETSI, CCITT. Such systems demand a very high quality of encoding, the usual requirement being toll or landline quality (waveform coders). However, for closed systems such as private commercial network and military systems, the quality factor may be reduced to lower the capacity requirements. Although absolute quality is often specified, it may be compromised for a lower standard if other factors are more important [WON96]. For instance, in a mobile radio system it is the overall average quality that usually takes into account both good and bad transmission conditions, therefore is often the deciding factor.

It is plausible to assume that by combining vocoders and waveform coders, a high quality speech coder operating at low bit rates (less than 8 kbps) could be produced. This is called the hybrid coder and it attempts to preserve the perceptually important parts of the input speech.

3.2.2 Coding Delay

The coding delay of a speech transmission system is a factor closely related to the quality requirements. Coding delay includes algorithmic (the buffering of speech for analysis), computational (time taken to process the stored speech samples) and transmission factors. Only the first two concern the speech coding subsystem, although very often the coding scheme is tailored such that transmission can be initiated even before the algorithm has completed processing all the information in the analysis frame. For example, in the Pan-European mobile system the encoder starts transmission of the spectral parameters as soon as they are available. For PSTN applications, low delay is essential if the major problem of echo is to be minimised. For mobile system applications and satellite communication systems, echo cancellation is already employed as substantial propagation delays already

exist. However, in the case of PSTN, where delay is very small, extra echo cancellers will be required if coders with long delays are introduced. This increases the overall cost of the system. The other problem of encoder/decoder delay is purely the subjective annoyance factor. Most low rate algorithms introduce a substantial coding delay compared with the standard 64 kbps PCM system [KON95]. For instance, the Pan-European DMR system's initial upper limit was 65 ms for a back-to-back configuration, whereas for the 16 kbps CCITT specification it was a maximum of 5 ms with an objective of 2 ms.

3.2.3 Complexity and Cost

As ever more sophisticated algorithms are devised, the computational complexity is increased. The advent of the digital signal processor (DSP) chips and custom application specific integrated circuits (ASIC) chips has enabled the cost of processing to be considerably lowered. However, complexity/power consumption, and hence cost, is still a major problem, especially in applications where hardware portability is a prime factor. One technique to overcome power consumption whilst also improving channel efficiency is digital speech interpolation (DSI). DSI exploits the fact that only around half of speech conversation is actually active, thus during inactive periods the channel can be used for other important purposes, and it can be used to limit the transmitter activity, hence saving power. An important subsystem of DSI is the voice activity detector (VAD) which must operate efficiently and reliably to ensure that real speech is not mistaken for silence, and vice versa [KON95]. Obviously, a silence mistaken for voice is tolerable, but the opposite can be very annoying.

3.3 Speech Coding Algorithms

Speech coders [SPA94] are broadly classified based on the means by which they achieve compression into two categories – waveform coders and vocoders. The hierarchy of speech coders is shown is Figure 3.1.

Waveform coders essentially strive to reproduce the time waveform of the speech signal as closely as possible. They are designed to be source independent and can hence code equally well a variety of signals. However, this class of coders have only moderate economy in transmission bit rate. Vocoders on the other hand achieve very high economy in transmission bit rate but are in general more complex. They are based on using a priori knowledge about the signal to be coded, and for this reason, they are signal specific.

The general function of these speech coders is to analyse the signal, remove the redundancies, and efficiently code the non-redundant parts of the signal in a perceptually acceptable manner. As the coding capacity is reduced the strategies for redundancy removal

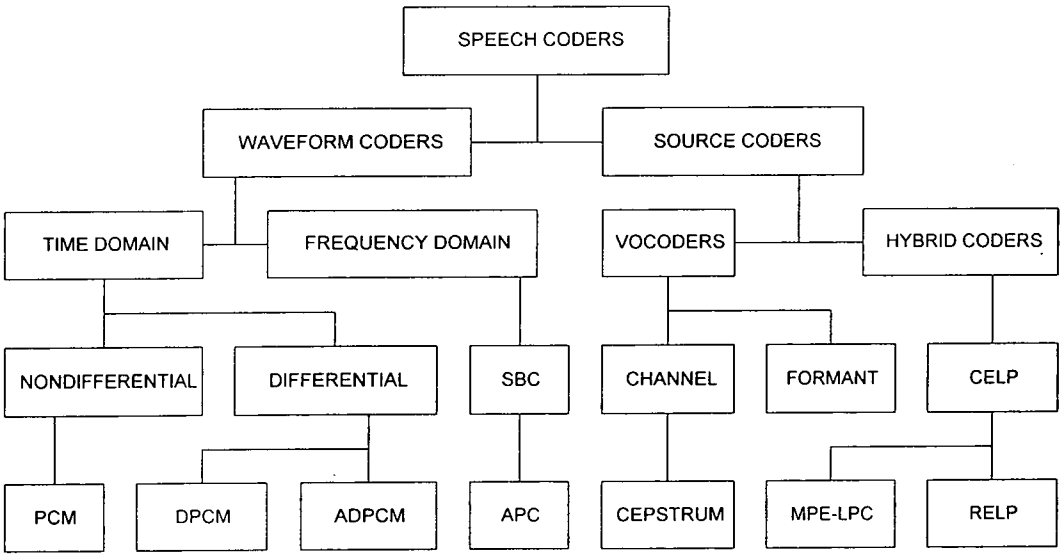


Figure 3.1 Hierarchy of speech coders.

and bit allocation need to be ever more sophisticated [RAP96]. The quality versus bit rate for the three main coding strategies, waveform coding, vocoding and hybrid coding, are shown in Figure 3.2. In this figure, the quality is represented by mean opinion scores (MOS)

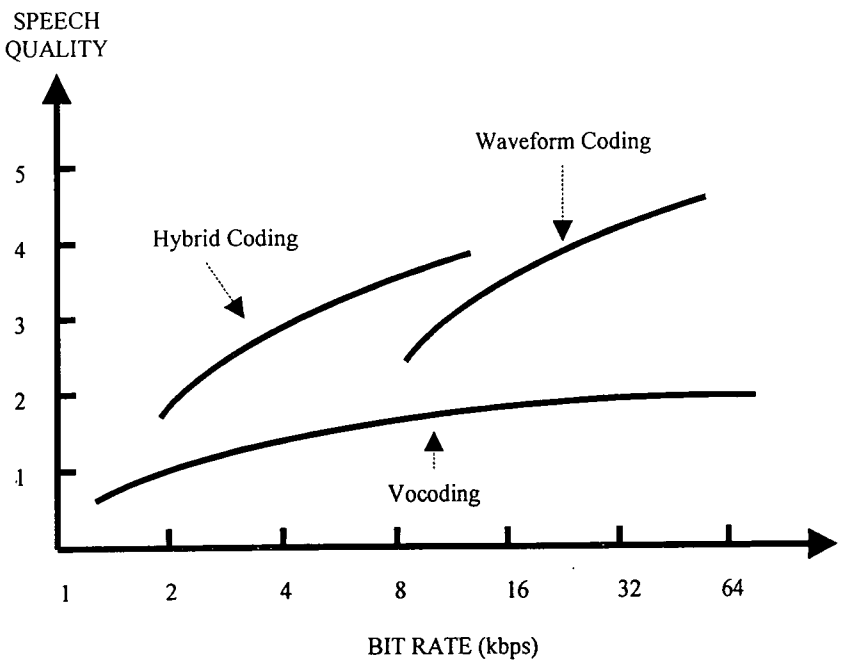


Figure 3.2 Quality comparison of speech coders.

ranging from 1 to 5, which corresponds to 1 = bad, 2 = poor, 3 = fair, 4 = good and 5 = excellent. A summary of the application standards currently in operation and those in development is shown in Table 3.2.

Table 3.2 Summary of available speech coders.

Coder	Coding Technique	Bit Rate (kbps)
G.711	PCM	32
G.722	SB-ADPCM	48/56/64
G.726	ADPCM	16/24/32/40
G.728	LD-CELP	16
G.729	CS-ACELP	8
GSM	Full-Rate RPE-LTP	13
GSM	Half-Rate VSELP	5.6
GSM	Enhanced Full-Rate CELP	13
IS-54	VSELP	7.95

3.4 Waveform Coding

Waveform coders attempt to reproduce the input signal's waveform. They are generally designed to be signal independent so they can be used to code a wide variety of signals. They also exhibit a graceful degradation in the presence of noise and transmission errors. However, to be effective they can only be used for medium bit rates. Waveform coding can be carried out in either the time domain or the frequency domain.

3.4.1 Time Domain Coding

Time domain coding defines a range of input voltages, splits this range of input into bits, and assigns a binary code to represent a number, in which the lowest number corresponds to the lowest input voltage, and the largest number corresponds to the highest input voltage. The resulting quality of this technique is approximately linearly proportional to the bits allocated per sample.

3.4.1.1 Pulse Code Modulation (PCM)

The PCM waveform coding algorithm [CAR86] [WON96] may be used to convert any band-limited analogue signal to a digital coded stream. Narrow-band speech is sampled 8

kHz (see 2.4), and then each speech sample must be quantised. If linear quantisation is used then about 12 bits per sample are need, giving a bit rate of about 96 kbps. However this can be easily reduced by using a non-linear quantisation. For coding speech it has been found that with non-linear quantisation 8 bits per sample is sufficient for speech quality which is almost indistinguishable from the original. This gives a bit rate of 64 kbps, and two such non-linear PCM coding algorithms were standardised in 1960s. In America μ-law is the standard, while in Europe the slightly different A-law compression is used. Because of their simplicity, excellent quality and low delay both of these techniques are still widely used today.

3.4.1.2 Differential Pulse Code Modulation (DPCM)

PCM makes no assumptions about the nature of the waveform to be coded, hence it works very well for non-speech signals. However, when coding speech there is a very high correlation between adjacent samples. This correlation could be used to reduce the resulting bit rate. One simple method of doing this is to transmit only the differences between each sample. This difference signal will have a much lower dynamic range than the original speech, so it can be effectively quantised using a quantiser with fewer reconstruction levels. In the above method the previous sample is being used to predict the value of the present sample. The prediction could be improved if a much larger block of speech is used to make the prediction. This technique is known as differential pulse code modulation (DPCM) [SPA94] [KON95].

3.4.1.3 Adaptive Delta Pulse Code Modulation (ADPCM)

The ADPCM waveform coding algorithm [SPA4] [RAP96] quantises the difference between the speech signal instead of quantising the speech signal directly and a prediction that has been made of the speech signal. If the prediction is accurate then the difference between the real and predicted speech samples will have a lower variance than the real speech samples, and will be accurately quantised with fewer bits than would be needed to quantise the original speech samples. At the decoder the quantised difference signal is added to the predicted signal to give the reconstructed speech signal. The performance is aided by using adaptive prediction and quantisation, so that the predictor and difference quantiser adapt to the changing characteristics of the speech being coded.

In the mid 1980s the CCITT standardised a 32 kbps ADPCM, known as G.721, which gave reconstructed speech almost as good as the 64 kbps PCM coding algorithm. Later in recommendations G.726 and G.727 operating at 40, 32, 24 and 16 kbps were standardised.

3.4.2 Frequency Domain Coding

Frequency domain waveform coders split the signal into a number of separate frequency components and encode these independently. The number of bits used to code each frequency component can be varied dynamically.

3.4.2.1 Subband Coding (SBC)

This is the simplest of the frequency domain techniques. In the subband coder [PRO92] [CRO93] [MARV93], the signal is passed through a bank of bandpass filters. Each sub-band is then lowpass translated and the sampling rates are reduced to the Nyquist rate for each band. The subbands are then coded using one of the time domain techniques. The number of bits assigned to each band can be varied according to the band's perceptual importance. At the receiver the sampling rates are increased and the bands are modulated back to their original positions. They are then summed to produce the output speech.

The main advantage of subband coding is that the quantisation noise produced in one band is confined to that band. This means that separate quantiser set-sizes can be used for each band. Therefore bands with lower energy can have lower step-sizes and hence are preserved in the reconstructed signal. The confinement of the quantisation noise also allows the perceptually weighted distribution of bits.

Subband coding has found widespread use in wide bandwidth, high quality commentary channels for teleconferencing. These systems use a coder described in the CCITT's G.722 standard. Subband coding using multirate techniques is described in chapter 6.

3.4.1.3 Adaptive Transform Coding (ATC)

This is a more complex technique and involves a block of transformation of a windowed segment of the input signal. The idea is that the signal is transformed into the frequency domain. Coding is then accomplished by assigning more bits to more important transform coefficients. At the receiver the decoder carries out the inverse transform to obtain the reconstructed signal.

The most commonly used transform is the Discrete Cosine Transform (DCT) [IFE93]. It is used because it is significantly less computationally intense than other transforms and its properties are almost the same. Most of the practical transform coding schemes vary the bit allocation among different coefficients adaptively from frame to frame while keeping the total number of bits constant. This dynamic bit allocation is controlled by time-varying statistics which have to be transmitted as side information. This constitutes an overhead of

about 2 kbps. The frame of N samples to be transformed or inverse-transformed is accumulated in the buffer in the transmitter and receiver respectively. The side information is also used to determine the step size of the various coefficient quantisers [RAP96].

3.5 Vocoding

Waveform coders make no assumptions about the nature of the signal to be coded. However, if the signal is always a speech signal then it would be more efficient to take the method of producing the signal into account. Vocoders [SPA94] [ROB97] assume an explicit model of speech production, shown in Figure 3.3. This model assumes that speech is produced by exciting a linear system, the vocal tract, by a series of periodic pulses if the sound is voiced or noise if it is unvoiced.

If the speech is voiced the excitation consists of a periodic series of impulses, the distance between these pulses equals the pitch period. If the speech is unvoiced the excitation is a random noise sequence, corresponding to the hiss production by air blowing through a constriction in the vocal tract.

The linear system models the vocal tract and its parameters can be determined using several techniques. It is the methods of obtaining this model that distinguishes the many different kinds of vocoders.

Vocoders attempt to produce a signal that sounds like the original speech, whether or not the time waveform resembles the original. At the transmitter the speech is analysed to determine the model parameters and the excitation. This information is then transmitted to

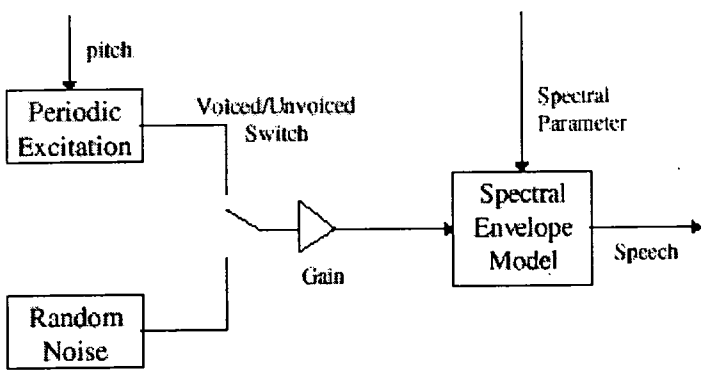


Figure 3.3 The speech production model used by vocoders.

the receiver where the speech is synthesised. The result of this is that they can produce intelligible speech at very low bit rates. However the synthesised speech sounds unnatural, so vocoders are normally used where bit rate is of utmost importance.

The poor quality of the vocoder output is attributable to the very simple nature of its speech production model. Especially the assumption that speech is either voiced or unvoiced, allowing no intermediate states. The ear is very sensitive to pitch information so for voiced speech the pitch must be accurately determined, a problem that has never been satisfactorily solved. They also suffer from a sensitivity to errors in vocal tract model, the error occurring in either calculating the model's parameters or transmitting the data to the receiver [MCCR95].

3.5.1 Channel Vocoders

The channel vocoder [SPA94] [RAP96] is the earliest of the vocoders. This coder takes advantage of the ear's insensitivity to short-time phase. For speech segments of about 20 ms only the magnitude of the spectrum needs to be considered. The spectrum is estimated using a filter bank. The more filters in the bank the better the results, but the higher the resulting bit rate. The output of each of these filters is then rectified and lowpass filtered to find the envelope of the signal. It is then sampled and transmitted to the receiver. The receiver does the exact opposite to the transmitter.

The bandwidths of the filters used in the filter bank tend to increase with frequency. This allows for the fact that the human ear responds logarithmically.

The channel vocoder can be implemented using either digital or analogue hardware and is capable of providing highly intelligible speech at bit rates in the region of 2.4 kbps.

3.5.2 Cepstrum Vocoders

The cepstrum vocoder [RAP96] separates the excitation and vocal tract spectrum by inverse Fourier transforming of the log magnitude spectrum to produce the cepstrum of the signal. The low frequency coefficients in the cepstrum correspond to the vocal tract envelope, with the high frequency excitation coefficients forming a periodic pulse train at multiples of the sampling period. Linear filtering is performed to separate the vocal tract cepstral coefficients from the excitation coefficients. In the receiver, the vocal tract cepstral coefficients are Fourier transformed to produce the vocal tract impulse response. By convolving this impulse response with a synthetic excitation signal, the original speech is reconstructed.

3.5.3 Formant Vocoders

The vast majority of the information in a speech signal is contained in the positions and bandwidths of the vocal tract's formants [ZOL96]. If these formats could be accurately determined then it would be possible to obtain a very low bit rate. In fact, with this technique it is possible to achieve less than 1 kbps. However, the formants are very difficult to determine accurately. For this reason the formant vocoder has never been very popular.

3.6 Hybrid Coding

To overcome the deficiencies of pure waveform and vocoding schemes, hybrid coding methods have been developed which incorporate the advantages offered by each of the pure schemes.

3.6.1 Multi-Pulse Excited LPC (MPE-LPC)

The main problem with the linear predictive vocoder is the excitation for the vocal tract model. The vocoder categorises the speech as either voiced or unvoiced. This has the effect of producing a very synthetic sounding output. Multi-pulse excitation tries to rectify this problem.

When a speech signal is passed through the linear predictor the correlation between adjacent samples is removed. However, for voiced speech the pitch of the speech introduces a long term correlation into the speech, resulting in the quasi-periodicity mentioned earlier. This periodicity is not removed by the linear predictor and produces large spikes in the residual.

This long term correlation can be removed by passing the residual through a second predictor. This second predictor, the pitch predictor, is designed not to remove the correlation from adjacent samples but to remove the correlation from adjacent periods of the residual. This is achieved by inserting a delay corresponding to the pitch period into the predictor. The output of this predictor will approximate Gaussian noise. The multi-pulse coder then excites the cascade of the two linear predictors with a series of impulses. Generally about four to six impulses are used as the excitation. The position and amplitudes of these impulses are determined using an analysis-by-synthesis procedure [CUC96]. The locations of the impulses are determined sequentially.

The multi-pulse coder is very effective at producing high quality speech at bit rate around 9.6 kbps and lower. A variation of the multi-pulse coder [KON95] is the regular pulse excitation coder (RPE). This coder uses regularly spaced pulse patterns instead of the lone

impulses of the multi-pulse coder. The GSM standard uses an RPE coder operating at 13 kbps.

3.6.2 Codebook Excited LPC (CELP)

As described above the main problem with vocoders is the simplistic model of the excitation used. Codebook Excited Linear Prediction is another way of circumventing this problem.

In the CELP coder [SPA94] [PAI96] [ROB97] the speech is passed through the cascade of the vocal tract predictor and the pitch predictor. The output of this predictor is a good approximation to Gaussian noise. This noise sequence has to be quantised and transmitted

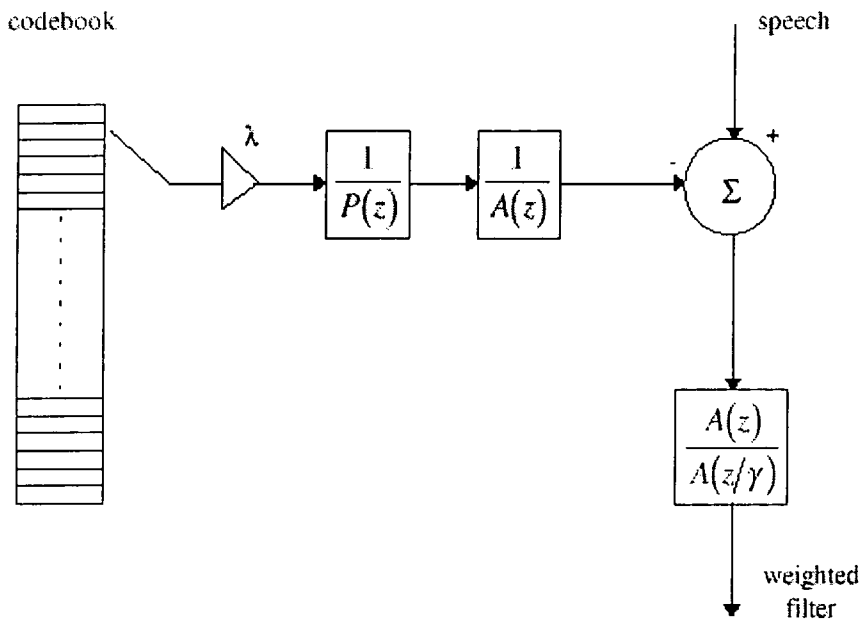


Figure 3.4 The analysis-by-synthesis codebook search of a CELP coder.

to the receiver. Multi-pulse coders quantise it using a series of weighted impulses. CELP coders use vector quantisation. The index of the codeword that produces the best quality speech is transmitted along with a gain term for it.

The codebook search is carried out using an analysis-by-synthesis technique, see Figure 3.4. The speech is synthesised for every entry in the codebook. The codeword that produces the lowest error is chosen as the excitation. The error measure used is perceptually weighted so the chosen codeword produces the speech that sounds the best [BLA96].

The codebook search is very computationally intensive but fast algorithms have been developed so that a CELP coder can now be implemented in real-time using modern digital signal processing microprocessors. This technique is currently one of the most effective method of obtaining high quality speech at very low bit rates.

3.6.3 Residual Excited LPC (RELP)

The rationale behind the residual excited LPC (RELP) [KON95] [RAP96] is related to that of the DPCM technique in waveform coding. In this class of LPC coders, after estimating the model parameters (LP coefficients or related parameters) and excitation parameters (voiced/unvoiced decision, pitch, gain) from a speech frame, the speech is synthesised at the transmitter and subtracted from the original speech signal to form a residual. The residual signal is quantised, coded, and transmitted to the receiver along with the LPC model parameters. At the receiver the residual error signal is added to the signal generated using the model parameters to synthesis an approximation of the original speech signal. The quality of the synthesised speech is improved due to the addition of the residual error. More information about RELP coders can be found in the Chapter 4.

3.7 Summary

This chapter discusses the current speech coding strategies and algorithms. The objectives and requirements of speech compression which includes the quality and capacity, coding delay, complexity and cost have been described. The current schemes have been outlined in the following categories: the high quality waveform coders, the low bit rate vocoders and the hybrid coders that attempts to fill the gap between waveform coders and vocoders. In the next chapter, the procedure of a real time implementation of the GSM full-rate speech compression which uses RELP enhanced by a long term predictor (LTP) will be described.

4 Real Time Implementation of GSM Full-Rate Codec on RISC DSP

4.1 Introduction

Raw digitised speech can be too large to store and requires more bandwidth than typically available on ordinary telephone lines. The Global System for Mobile communication (GSM) is a digital mobile radio system which is extensively used throughout Europe, and also in many other parts of the world. The GSM full-rate speech transcoder operates at 13 kbps and uses RPE-LTP (regular pulse excitation long-term predictor). The specification requires bit-exactness which imposes strict constraints on an implementation using any fixed-point digital signal processor (DSP).

A real time process is a task which is performed at a rate which can keep up with incoming data within a specified time limit. The time limit may vary from 200 ns to 30 ms, hence the delay can be detected if it is more than 30 ms [GAB47]. In digital speech coding, assuming 8 kHz sampling rate, the real time processing needs to be performed within 125 μs for coders involving the sample-by-sample coding process.

In this chapter, an investigation into a real time implementation of GSM full-rate speech compression algorithms was carried out on the new generation RISC processor for parameterised DSP called GEPARD which is by Austria Mikro Systeme International (AMS) is described and analysed.

4.2 Overview of GSM Full-Rate Speech Transcoding (GSM 06.10)

The GSM full-rate speech compression algorithm is a lossy technique which is based on a residually excited linear predictive coder (RELP) and this is further enhanced by using a long term predictor (LTP). This improves speech quality by removing the structure from the vowel sounds prior to coding the residual data. It compresses frames of 160 13-bit signed samples to 260-bit compressed frames. The specification of the coder/decoder (codec) is fully defined in [ETS94]. The architecture of GSM is shown in Figure 4.1.

The simple block diagram of the speech encoder is shown in Figure 4.2. The encoder is comprised of four major processing blocks. The input speech frame consisted of uniform 13-bit PCM signed samples converted from 8-bit A-law companded format. The speech sequence is first pre-emphasised to produce an offset-free signal, ordered into segments of

20 ms duration (160 samples), and then Hamming-windowed. This is followed by short-term prediction (STP) filtering analysis where the samples obtained are analysed to determine the reflection coefficients (8 in total). These parameters are then used for the filtering of the same 160 samples. The results is 160 samples of the short term residual

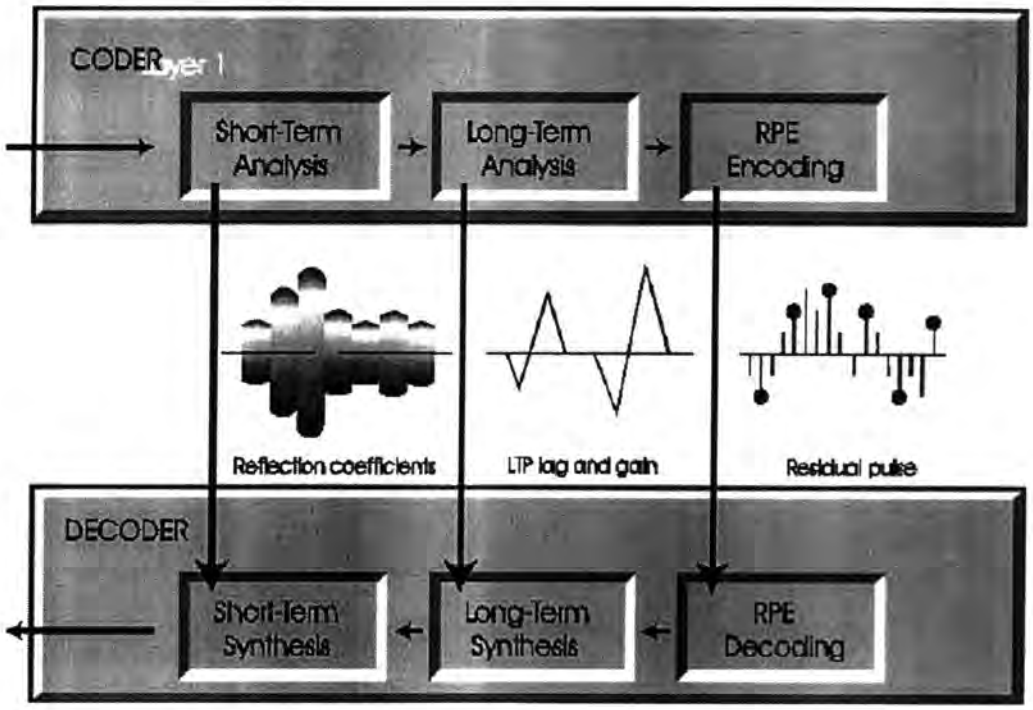


Figure 4.1 GSM architecture.

signal. The reflection coefficients are transformed to logarithmic area ratios, LAR, before transmission. The 8 LAR parameters have different dynamic ranges and probability distribution functions, and hence not all of them are encoded with the same number of bits. The LAR parameters are also decoded by the LPC inverse filter so as to minimise the error .

For the LTP analysis which involves finding the pitch period and gain factor, the speech frame is divided into four subframes with 40 samples of the short term residual signal in each. Each subframe is processed blockwise by the subsequent functional elements. To minimise the LTP residual, pitch extraction is done by the LTP by determining that value of delay which maximises the crosscorrelation between the current STP error sample a stored sequence of the 120 previous reconstructed short term residual samples. The extracted pitch and gain factor are transmitted and encoded at a rate of 3.6 kbps.

A block of 40 long term residual signal samples is obtained by subtracting 40 estimates of the short term residual signal from the short term residual signal itself. The resulting block of 40 long term residual samples is fed to the Regular Pulse Excitation, RPE, analysis which performs the basic compression function of the algorithm.

As a results of the RPE analysis, the block of 40 input long term residual samples are represented by four candidate excitation sequences of 13 pulses each. The energies of these sequences are identified, and the one with the highest energy is selected to represent the LTP

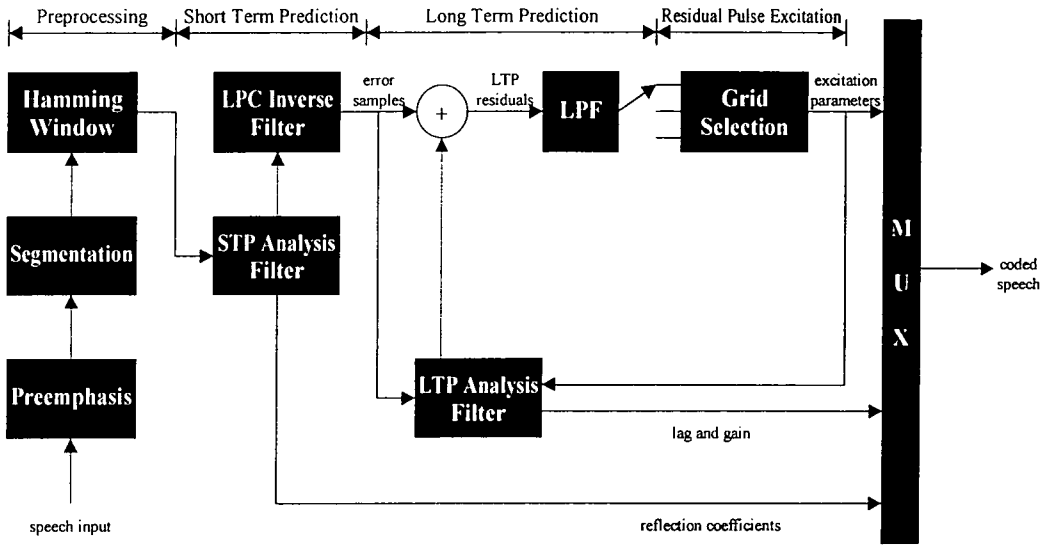


Figure 4.2 Simple block diagram of GSM speech encoder.

residual. The subsequence selected is encoded using Adaptive Pulse Code Modualtion, APCM, with estimation of the subblock amplitude which is transmitted at a rate of 9.6 kbps [SCO95].

The RPE parameters are also fed to a local RPE decoding and reconstruction module which produces a block of 40 samples of the quantised version of the long term residual signal. By adding these 40 quantised samples of the long term residual to the previous block of short term residual signal estimates, a reconstructed version of the current short term residual signal is obtained.

The block of reconstructed short term residual signal samples is then fed to the long term analysis filter which produces the new block of 40 short term residual signal estimates to be used for the next subframe thereby completing the feedback loop.

A detailed block diagram of the encoder can be found in Appendix B. Table I.1 shows the input and output parameters of the encoder.

Figure 4.3 shows a simple block diagram of the GSM decoder. It consists of four blocks which perform operations complementary to those of the encoder. The received excitation parameters are RPE encoded and passed to the LTP synthesis filter which uses the pitch and gain parameter to synthesise the long-term signal. Short-term synthesis is carried out using the received reflection coefficients to recreate the original speech signal. The input and

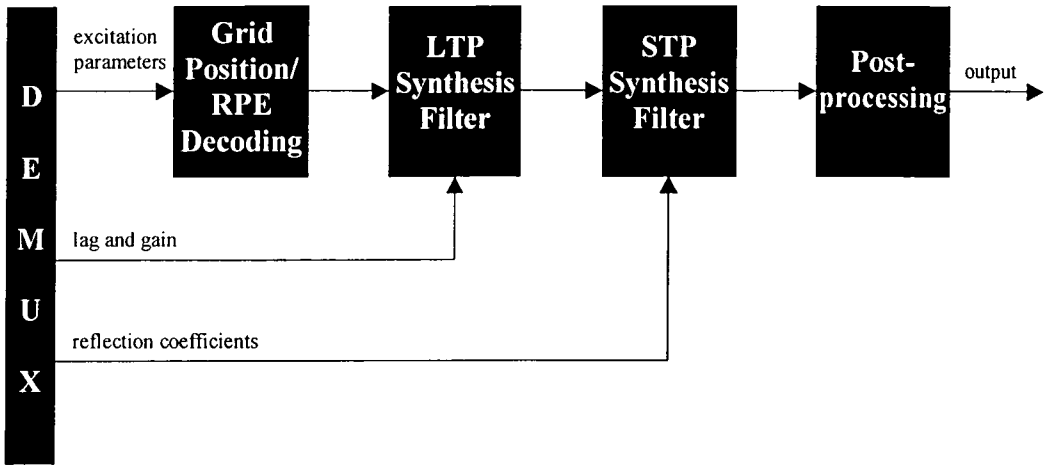


Figure 4.3 Simple block diagram of GSM speech decoder.

output parameters are shown in Table I.2. A detailed block diagram of the RPE-LTP decoder is shown in Appendix B.

The ETSI standard for full rate must be bit-exact. The specification gives no allowance for even minor deviations since in telecommunications different equipment from different suppliers must be compatible. The algorithm is also explained in more detail in Appendix A.

4.2.1 Encoded Parameters

The three different groups of data produced by the encoder are:

- the short term filter parameters
- the long term prediction (LTP) parameters
- the RPE parameters

The encoder produces this information in a unique sequence and format, and the decoder must receive the same information in the same way. In Table C.1, the sequence of the output bits b1 to b260 and the bit allocation for each parameter is shown.

4.3 Hardware Requirements

Real time implementation of a speech coding algorithm is very important from a cost point of view. Any speech coding algorithm can be implemented using available digital signal processor (DSP) chip technology, but the cost of that implementation will increase rapidly with the increase in the number of DSP chip used. The other important consideration in real

time implementation is the power consumption of the final product, especially in applications where hand held mobile telephones are used.

The selection of a DSP for the implementation of a specific speech coding algorithm is also affected by other factors. The two most important factors that need careful consideration are the instruction cycle time of the DSP and the suitability of its instruction set for the main processing in the speech coding algorithm [KUN85]. The other useful features of a DSP that should be taken into account are

- the amount of on-chip RAM/ROM
- number of MIPS (million instructions per second)
- program and data bus structure
- off-chip memory capacity
- boot memory controller
- on-chip peripherals
- fixed and floating-point arithmetic capability
- on-chip cache
- addressing modes
- on-chip direct memory access (DMA)
- internal and external interrupts
- hardware/software programmable wait states
- on-chip emulation ports; power-down capability
- power dissipation

4.3.1 DSP Chips

The programmable DSPs [IFE93] are categorised by precision and arithmetic types

- fixed point
- floating point

The fixed-point DSPs tend to be faster and cheaper, but they are more difficult to program and provide less precision. Fixed-point DSPs exist with a data precision of up to a 32-bit word length. The overall number of gates in a DSP system is a function of the bits in its numeric representation. The gate count increases with the available range and precision of

the numeric representation used. Therefore one trade-off that must be made is mathematical performance versus circuit and system complexity.

There are a lot of advantages of floating-point implementation over fixed-point. These advantages often lead to great savings in the development effort for a product or program. The precision of a floating-point number remains constant throughout a program because of automatic normalisation of the mantissa by the processor, whereas the precision of fixed-point data varies with the size of the stored data. Rounding or truncation leads to much smaller overall errors than for a fixed-point implementation [KON95]. This constant precision provided by the floating-point DSPs, coupled with the ability to represent very large or very small numbers allows a for more precise placement of poles and zeros, eliminating most of the implementation problems of filters.

4.3.2 RISC Machines

RISC stands for *Reduced Instruction Set Computer* [FUR96]. In the mid-1970's advances in semiconductors technology began to reduce the difference in speed between main memory and processor chips. As memory speed increased, and high-level language, computer designers began to look at ways computer performance could be optimised beyond just making faster hardware. One of the key realisation was that a sequence of simple instructions produces the same results as a sequence of complex instructions, but can be implemented with a simpler and faster hardware design, assuming that memory can keep up. RISC machines were the result.

In a RISC machine, the instruction set contains simple, basic instructions, from which more complex instructions can be composed. Each instruction is the same length, so that it may be fetched in a single operation. Most instructions complete in one machine cycle, which allows the processor to handle several instructions at the same time. This pipelining is used to speed up RISC machines.

RISC machines offer the advantage of a smaller die size. They require fewer transistors and less silicon area. A whole Central Processing Unit (CPU) can fit on a chip at an earlier stage in process technology development. A RISC CPU leaves more die area free for performance-enhancing features such as cache memory and memory management functions. RISC machines also take less design effort and therefore have a lower design cost compared to more conventional Complex Instruction Set Computer (CISC) [FUR96] architecture.

4.4 GEPARD

The complexity of the software required to implement the GSM full-rate codec algorithm on different DSPs strongly depend on the architecture of the processor (e.g. saturation logic). The codec uses basic mathematical operations and requires bit-exact shift operations as well as the multiply-accumulate (MAC) instruction. A bit-exact implementation is done using GEPARD by AMS. One of the major objectives of this work was to use the GSM algorithm to investigate optimisation technique for an Application Specific Integrated Circuit (ASIC) [SCH92] processor core GEPARD, which is produced by AMS. The implementation of GSM full-rate algorithm on the GEPARD processor will be discussed in the next section.

GEPARD is an embedded software programmable DSP core for telecommunication, consumer and industrial applications. The GEPARD core is a RISC DSP as the instruction set is reduced and simplified to achieve minimum silicon cost for good performance. Its main features are as follows

- fixed point DSP with parameterised word length
- fully parallel multiplier with multiply-accumulate facility which has the extended precision needed to overcome the growth of word length present in many DSP algorithms
- four accumulators for efficient complex number calculations
- complete arithmetic/logic functions for implementation of high level language constructs such as for-loops and if-then decisions
- software stack is implemented for interrupt support which has a latency of two cycles, also used for function/procedure calls
- A macro-assembler and a simulator are available for a debugging environment.

More information about GEPARD programming can be found in [AMS96], [AMS97a] and [AMS97b].

4.5 Implementation Strategies

The real time implementation is important in many communications systems, where as well as requiring high quality speech, the equipment cost and power consumption must be very low. Therefore it is very important to implement the algorithm in an optimised manner. The real time implementation of the GSM full-rate speech coding algorithm on GEPARD was approached in several stages

- Preparation
- Development of bit-exact C code
- Real time implementation on GEPARD
- Optimisation of GEPARD assembly code
- Testing and debugging of GEPARD assembly code

The different stages are described in the following sections.

4.5.1 Preparation

The GSM full-rate algorithm was analysed. A floating point version of the algorithm was obtained from [DEG94] and tested with real speech signals. The waveform of the input file is shown in Figure 4.4. Some mathematical algorithms used the in LPC analysis described in 4.2, e.g. autocorrelation and linear prediction, were studied. The plots of the first nine autocorrelation values and eight linear predictive coefficients of the first frame of the input file are shown in Figure 4.5. The first frames of the original and decoded GSM compressed signals are shown in Figure 4.6. As the GSM full-rate algorithm is a lossy technique, it can be seen that some level of information of the original speech signal has been lost during the process. A fast Fourier transform of each signal was performed and the results are shown in Figure 4.7. The plots show that the algorithm work better at lower frequencies as some high frequency components are missing in the decoded GSM compressed frame.

A GSM 06.10 RPE-LTP coder and decoder in C, called *toast*, developed by Jutta Degener and Carsten Bormann [DEG94] were examined and tested with various speech files. The C code is available at ftp://ftp.cs.tu-berlin.de/pub/local/kbs/tubmik/gsm/ddj/gsm-1010.zip.

Before the algorithm was implemented on GEPARD, some test programs were written in GEPARD assembler in order to become familiar with the software. The programs were designed to carry out simple tasks including reading and writing a set of values and performing arithmetic operations. A key feature of GEPARD was the parallelisation of instructions. Several operations could be combined into a single instruction which will be executed in a single cycle. However, the combinations of instructions were restricted and required careful planning. For example, only one Arithmetic/Logic Unit (ALU) instruction (e.g. `ADD`, `ABS`) can be used per cycle. Therefore, an `ADD` instruction can be combined with `MUL` which uses the multiplier, or a parallel memory load, or both in the same cycle.

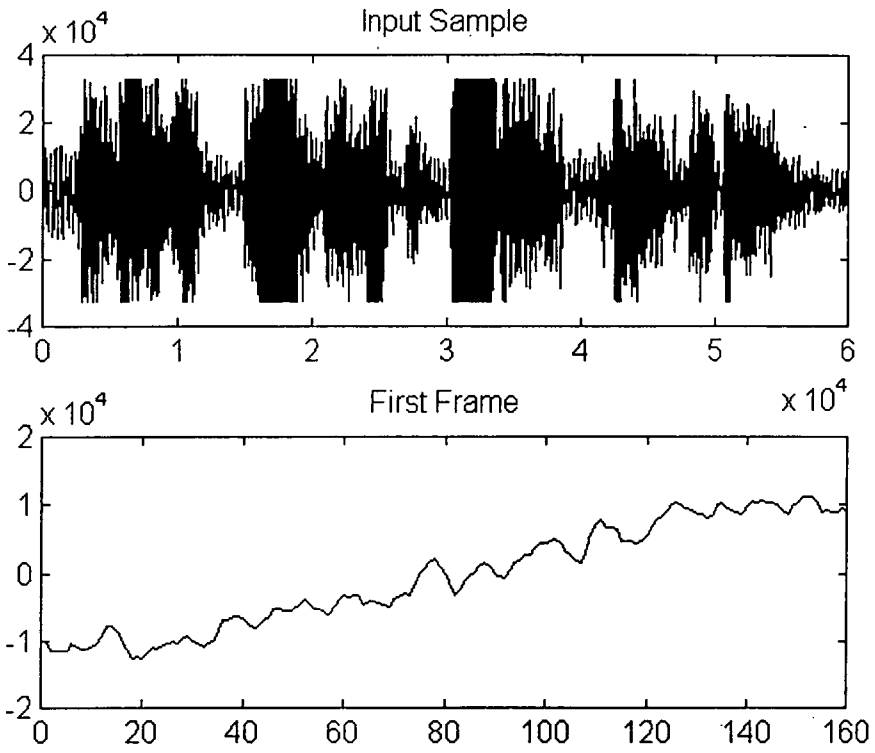


Figure 4.4 The waveform of "no wonder we're dangling at the bottom of the food chain".

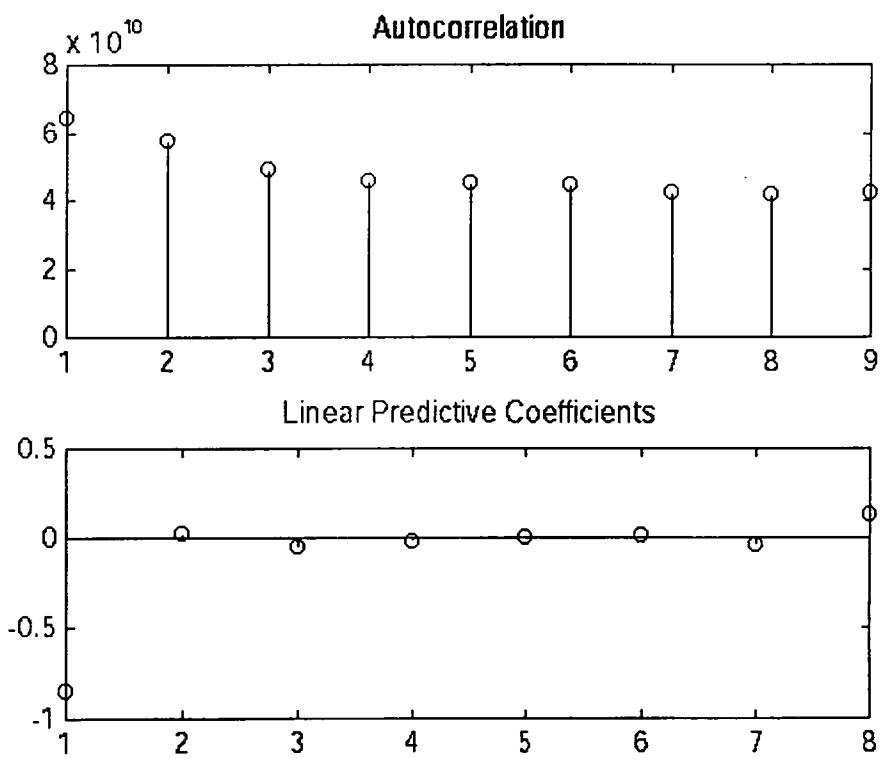


Figure 4.5 The autocorrelation values and linear predictive coefficients of the first frame. Calculated using MATLAB commands `X = xcorr(wavedata)` and `A = lpc(ac(1:9),8)` respectively.

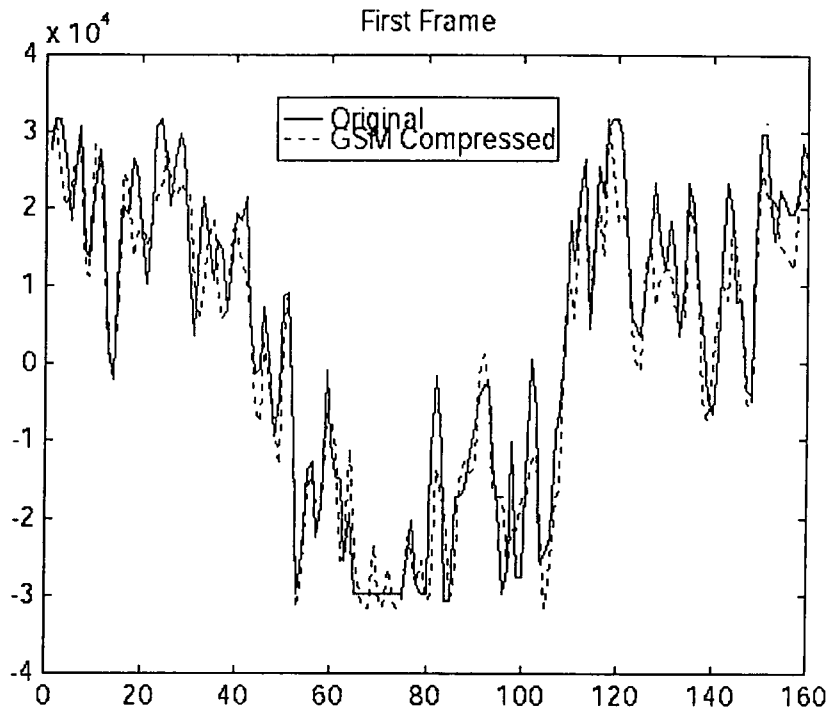


Figure 4.6 Comparison of the first frame of original and reconstructed speech signal.

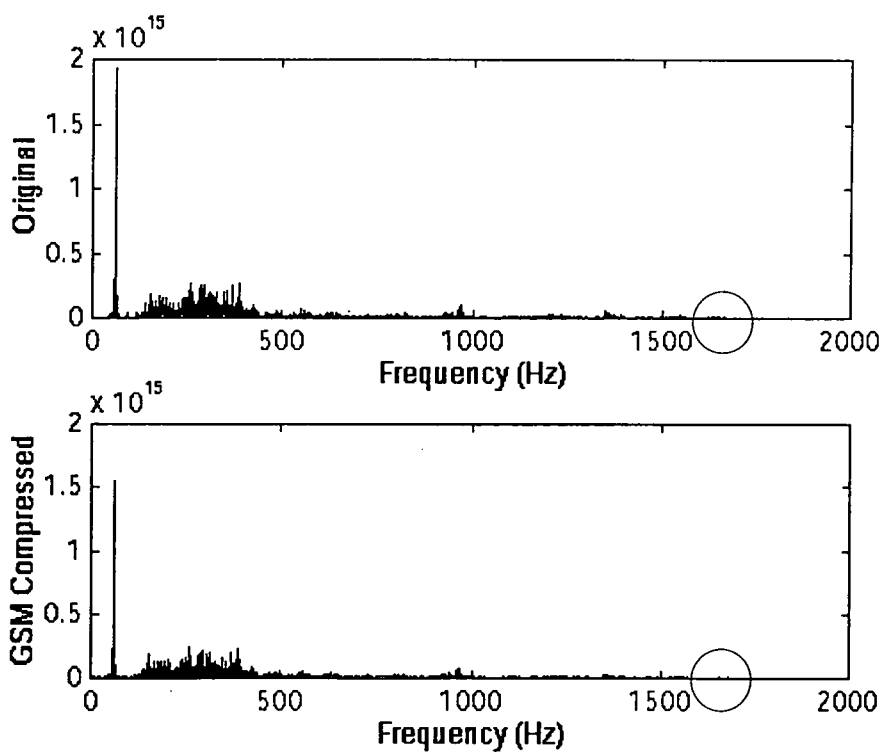


Figure 4.7 Fast Fourier transform of the original and reconstructed speech signal. Calculated using MATLAB command `fft(wavedata)`. The circled areas show the difference between the original and GSM compressed frames at high frequencies.

4.5.2 Development of Bit-Exact C Code

The ETSI specification was analysed and both coding and decoding parts were divided into four sections as illustrated in Figure 4.1 and 4.2. As the decoder consisted fewer subblocks and used some of the routines of the encoder, this was chosen to be implemented first. Both the long term and short term synthesis parts were the reverse of the analysis. The extra routines of the encoder included the autocorrelation and linear prediction subblocks, where the algorithms had been studied from previous experiments described in 4.5.1.

A bit-exact version of GSM full-rate speech codec was written in C language and is listed and Appendix F. The code was kept as similar to the specification as possible due the requirement of bit-exactness and tested by using the test vectors provided by ETSI [ETS94]. Details of the test files are listed in 4.5.7. The C code was later used for the debugging of the GEPARD assembly code.

4.5.3 Implementation on GEPARD

As GEPARD did not have a DSP cross compiler (i.e. from C-language to DSP machine language), the C code written previously could not be translated directly into assembly language. The algorithm was hand coded using the GEPARD instruction set. The C code was converted to assembly language and some modification was made due to the special characteristics of GEPARD including parallelisation. For example, for the scaling subblock in the preprocessing section which downscaled in input samples by a factor of two, the C code was written as shown below,

```
void scaling (word sop[160], word so[160])
{
  int k;

  for (k = 0; k < 160; k++) {
     so[k] = sop[k] >> 3;          /* shift right 3x, LSBs lost */
     so[k] = so[k]  << 2;          /* shift left 2x, 0s fed into LSB bits */
  }
}
```

The function of the code is explained in A.1.1. Five cycles would be required for the inner loop. However, by using the instructions available in GEPARD the assembly code could be written as

```
AND    sop[k], [-8], so[k];        /* -8₂ = 1111111111111000 */
RSHIFT so[k], so[k];               /* so[k] only right shifted once */
```

From the above code only two cycles were needed. Therefore the code is reduced by a factor of two and a half.

The algorithm required bit-exactness. It was decided that 16-bit data word length was used in this implementation to save silicon costs and therefore the 32-bit operations specified by ETSI had to be modified. For example, for a long multiplication `L_MULT(VAR1,VAR2)` that produced a 32-bit result, two 16-bit registers were used to represent a 32-bit number. The condition specified by ETSI was

```
L_MULT(VAR1,VAR2)=(VAR1 x VAR2) << 1
```

with the exception of

```
L_MULT(-32768,-32628)
```

which did not occur in the algorithm [ETS94]. The assembly code was

```
MUL     c, x;          /* c x x (the multiplier input is always c) */
MOV     ph1, z;        /* move hiword << 1 to z */
MOV     pl, y;         /* move loword to y */
ADD     y, y, y;       /* y << 1 (shifts explained below) */
```

The numbers were kept as 16 bit at all times in the program and a 32-bit number was stored as two 16-bit numbers `y` and `z`. The frequently used shift operations were performed using the multiply instruction, since

$$x << 1 = x \times 2^1$$

$$x << 15 = x \times 2^{15}$$

$$x >> 15 = x / 2^{15}$$

An example of a shift operation, `x << SHIFT`, can be written as

```
LOAD    [SHIFT], c;    /* load no of shifts from memory to c */
MUL     c, x;          /* x x 2^SHIFT */
```

For example, if `c = x = 3`, the answer would be $3 \times 2^3 = 24$, which is equivalent to $3 << 3$ shifting the value 3 in binary three places left). This method was generally more efficient if the number of shifts was more than three or the operation was done within a loop.

One of the main features of GEPARD was the fully parallel multiplier with multiply-accumulate (`MAC`) facility [AMS97a] which had the extended precision needed to overcome increasing of word length present in many DSP algorithms. The multiply-accumulate operation required by ETSI involves a 32-bit multiplication and a 32-bit addition. This facility reduced the number of cycles required for this specific operation and simplified the code without losing accuracy. The algorithm

```
L_TEMP = L_MULT(s[i], s[i - k]);          /* L_TEMP = s[i] x s[i - k] */
L_ACF[k] = L_ADD(L_ACF[k], L_TEMP);       /* L_ACF[k] = L_ACF[k] + L_TEMP */
```

was simplified by using the built-in MAC instruction which combines the above operations

```
MAC     s[i], s[i - k];
```

At the end of the operation, the results were moved to the accumulators. Since it is a 32-bit operation, the final result will be split into two parts and had to be stored in two separate accumulators. The MAC instruction can be used in parallel with either a memory load or an ALU instruction. This is very useful as the values for the next operation can be loaded from memory within the same cycle.

4.5.3.1 Specification of Arithmetic Operations

The arithmetic operations used in the GSM full-rate codec are defined by ETSI to ensure the bit-exactness of the code. The variables are represented in two's complement integer format [SHO87] (see Table 4.1). In this representation, the most significant bit (MSB) is the sign bit. For a 16-bit wordlength, each number lies in the range from -32768 to 32767.

Table 4.1 Two's complement number system for a 16-bit wordlength.

Number	2's Complement (Binary)	2's Complement (Hexadecimal)
32767	0111111111111111	0x7fff
16384	0100000000000000	0x4000
8192	0010000000000000	0x2000
4096	0001000000000000	0x1000
0	0000000000000000	0x0000
-4096	1111000000000000	0xf000
-8192	1110000000000000	0xe000
-16384	1100000000000000	0xc000
-32768	1000000000000000	0x8000

Two's complement arithmetic is used in the operations specified by [ETS94]. The operations performed include saturation controls which checks if the final result is within the 16-bit number range. The definitions of the arithmetic operations in the ETSI recommendation could be directly translated into C code. However, it was more complicated in assembly language. Since GEPARD does not provide the saturation requirements specified, a set of macros for the operations with overflow control and

saturation was written. For example, for a simple 16-bit operation like the ADD instruction, in order to make sure that the result is between 32767 and -32768, the signs of the two operands are checked and then one of them is compared with the result. The overflow (or underflow) check is shown in Table 4.2.

Table 4.2 Saturation check for `ADD(word VAR1,word VAR2)`.

VAR1	VAR2	RES	
+	+	+	no overflow
+	+	-	overflow (return 32767)
+	-	x	no overflow
-	+	x	no overflow
-	-	-	no overflow
-	-	+	overflow (return -32768)

The GEPARD assembly code for a 16-bit ADD instruction with saturation check was written as

```
XOR     VAR1, VAR2, TEMP;
JN      TEMP, END;          /* jump if sign bits of operand different */
ADD     VAR1, VAR2, RES;    /* RES = VAR1 + VAR2 */
NOT     VAR1, VAR1;         /* NOT(VAR1) */
XOR     RES, VAR1, TEMP;
JN      TEMP, END;          /* jump if sign bits of operands & result same */
ADD     NULL, ONES, TEMP;   /* ONES = 1111111111111111 */
JN      VAR1, END;          /* jump if NOT(VAR1) negative */
RSHIFT TEMP, RES;           /* set RES = 32767 */
NOT     RES, RES;           /* set RES = -32768 */
END:
```

The GEPARD macros for other arithmetic operations can be found in Appendix G.

4.5.3.2 Program Planning

The structure of the GEPARD code was similar to the C code. Both the encoder and the decoder were divided into 4 sections. The subblocks were kept the same as the ETSI specification. Some macros were written for the frequently used routines, e.g. shift routines and loops and can be found in Appendix G. The constants used in the codec, including those in the C program `gsmtab.c` were organised in tables and data memory was allocated. This can be seen in the GEPARD code in Appendix H.

4.5.4 Real Time Implementation

The implementation on GEPARD was done in the same way as the C code, with the decoder written first. The parameters of the encoder and decoder are listed in Appendix I. Each subblock was tested separately before put together. The assembly code of GSM full-rate speech codec for GEPARD is shown in Appendix H.

4.5.5 Optimisation of GEPARD Assembly Code

As the GEPARD assembly code was hand coded as mentioned in 4.5.3, an efficient way to optimise the assembly code was the look for the critical loops which are executed repeatedly. It was very important to reduce the number of cycles inside the loops, even if this added extra instructions outside. The approximate number of cycles of each subblock can be found in Appendix I. A typical example is the autocorrelation loop.

```
/* Compute the autocorrelation
 *           ,--,
 *   ac(l) = >  x(i) * x(i-l)   for all i
 *           `--'
 * for lags l between 0 and lag-1, and x(i) == 0 for i < 0 or i >= n
 */
void autocorrelation(
    int   n, double const * x,  /*  in: [0...n-1] samples x          */
    int lag, double       * ac) /* out: [0...lag-1] autocorrelation */
{
    double d; int i;
    while (lag--) {
        for (i = lag, d = 0; i < n; i++) d += x[i] * x[i-lag];
        ac[lag] = d;
    }
}
```

In this example, it can be seen that the *for* loop will be executed from 152 to 160 times, as lag is 9. The while loop will be executed 9 times. Therefore this whole subblock will require at least 1440 cycles in C code and possibly more in GEPARD code. Parallelisation is very useful in this case and the number of cycles required this routine was reduced by a factor of 4.

The techniques used for optimisation of the assembly code can be described using the example code that reads 160 input values and stores them in memory.

```
/* code */
LOAD   #DATA, [i0];         /* load address of DATA to register */
LOAD   #0x4800, [i2];       /* load input file */
```

```
LOAD   #160, x;          /* load constant 160 to accumulator x */
LOAD   #-1, a;           /* load constant -1 to accumulator a */
READ_DATA:               /* READ_DATA loop */
ADD    x, a, x;          /* decrement x */
LOAD   [i2]+1, y;        /* load input from memory to accumulator y */
NOT    x, z;             /* NOT(x) */
STORE  y, [i0]+1;        /* store y in memory */
JN     z, READ_DATA;     /* check if all 160 values are read, if not go
                            back to READ_DATA */
NOP                      /* delay slot, no operation */
END:                     /* exit loop when all values are read */
```

The code can be optimised as shown below. The changes are printed in bold and the methods used are numbered and explained.

```
/* data */
TABLE: -160, 1;          (1)
/* code */
LOAD   #DATA, [i0];      /* all constant load instructions,      */
LOAD   #0x4800, [i2];    /* cannot be parallelised               */
LOAD   #TABLE, [i4];
LOAD   [i4]+1, x;        (2)
LOAD   [i4], a;          (2)
READ_DATA:
ADD    x, a, x;          LOAD   [i2]+1, y;    (3)
/* NOT   x, z; */        (2)
JN     z, READ_DATA;
STORE  y, [i0]+1;        (4)
/* NOP */                (4)
END:
```

The optimisation techniques used for the GEPARD assembly code are

1. The frequently used constants were arranged in tables and loaded from memory. In this example, the constants -160 and 1 are both stored in the same data memory to prevent a constant load instruction being used twice since it cannot be parallelised with other instructions. An extra cycle is actually needed to load the address of TABLE. However, the memory load instruction can be combined with an ALU instruction or a multiply (not shown here) and this is particularly useful if a lot of constants are used.

2. For the repetitive loops, the jumps were optimised by initialising the loop count register to the negative of the number of iterations. The constant -160 is used in the example for 160 repetitions as it does not have to be made negative by using NOT (used in the unoptimised example). The loop count is incremented by adding 1 until it becomes positive and the loop is exited.

3. The instructions were rearranged to make use of parallelisation. A memory load can be combined with the ALU ADD.

4. The number of nop was minimised, and substituted with another instruction if possible especially after a jump instruction. The delay slot is filled with a memory store in the optimised example.

A list of other optimisation example code is listed in Appendix G.

Since the GEPARD core is a RISC DSP, the instruction set was reduced to minimum and the program memory was sometimes sacrificed to cut down the execution time. Using the previous example code, the total time for this code is $3 \times 160 + 5 = 485$ cycles. By modifying the READ_DATA loop,

```
/* data */
TABLE: -160, 1;
/* code */
LOAD   #DATA, [i0];
LOAD   #0x4800, [i2];
LOAD   #TABLE, [i4];
LOAD   [i4]+1, x;
LOAD   [i4], a;
READ_DATA:
ADD    x, a, x;            LOAD   [i2]+1, y;
STORE  y, [i0]+1;
ADD    x, a, x;            LOAD   [i2]+1, y;
JN     z, READ_DATA;
STORE  y, [i0]+1;
END:
```

The total time is now $5 \times 80 + 5 = 405$ cycles. The jump that instruction occupies one cycle, is shared by two iterations of the original loop.

Global data structure planning was also important as this avoided loading constants (as explained before). The two most expensive operations in GEPARD instruction set are load constant and jumps as they are full moves and any other instructions could not be run in the same cycle. It was often found to be effective to replace these with other instructions.

4.5.6 Testing and Debugging of GEPARD Assembly Code

Test sequences provided by ETSI were used for the testing of the GSM full-rate GEPARD assembly code. Since the test sequence files are written in binary using 16 bit words, they are first converted into text format using the program dataconv.c written in C language which can be found in Appendix F.

The files provided are

- Files for input of the encoder (SEQxx.INP)
- Files for the input of the decoder or the comparison with the encoder output (SEQxx.COD)
- Files for comparison with the decoder output (SEQxx.OUT)

Table 4.3 gives the contents of the files, the size in bytes, the number of frames and test area for each test sequence file.

Table 4.3 Contents and size of test sequence files.

File	No of Frames	Size	Test Area
SEQ01.INP	584	186880	Over and underflow
SEQ01.COD		88768	
SEQ01.OUT		186880	
SEQ02.INP	947	303040	Jumps
SEQ02.COD		143944	
SEQ02.OUT		303040	
SEQ03.INP	673	215360	General
SEQ03.COD		102296	
SEQ03.OUT		215360	
SEQ04.INP	520	166400	Zeros
SEQ04.COD		79040	
SEQ04.OUT		166400	
SEQ05.COD	64	9728	Decoder only
SEQ05.OUT		20480	

The GEPARD assembly code was debugged with the bit-exact C code. In GSM full-rate algorithm, each new frame requires the results produced by the previous frame, hence the test sequences could not be broken up into segments and tested separately as the information from the previous frame would be lost. The C code was modified and used to generate the variables required for each frame. The input frames (160 values) were isolated and the output of each subblock was compared with the output of the C program. A section of assembler which read in the memory values which should be in memory at the start of each frame generated from the C program for testing purposes. Simulating a single frame was a

much quicker using this method and there was also a possibility of simulating the rest of the sequence after the frame to see if or where the next problem occurred.

Most of the bugs occur at shifts, e.g.,

```
MUL    c, [0x0008];
```

The result is equivalent to shifting x three times to the left. However if the number has to be shifted 15 times to the left, it is multiplied by 0x8000 (which is recognised as a negative number by GEPARD) and the sign is changed, e.g.,

```
MUL    c, [0x8000];
```

The sign of the result is reversed in this case. Some other bugs are found in conditional jumps, e.g. index registers point to different addresses after jumps. An example is given below.

```
LOAD   [i2], a0;     /* load the value in [i2] to a0 */
JN     a0, NEXT;     /* jump to NEXT if a0 is negative */
NOP
STORE  a0, [i2]+1;   /* store a0 in [i2] and increment [i2] */
NEXT:
LOAD   [i2], a0;     /* address of [i2] will be different */
```

There were also some bugs caused by saturation, i.e., if the result was greater than 32767 or less than -32768, the sign of that result was changed automatically. Since GSM full-rate speech codec depended mainly on previous results, if something went wrong in one frame, the results of the following frames would be affected. This made testing and debugging a very slow process.

The GEPARD code was tested with all of the test sequences and was bit-exact. The performance of the code will be discussed in the next section.

4.6 Performance of GEPARD Assembly Code

Both compression and decompression times are proportional to the number of samples and the processor clock rate. Timings and number of cycles required for a 20MHz GEPARD processor per frame (160 samples) and per 8000 samples (one second of speech data at the standard GSM sampling rate of 8 kHz) are shown in Table 4.4.

Table 4.4 Performance of the GSM full-rate GEPARD assembly code.

	Average Number of Cycles		Average Time (ms)	
Performance	Per Frame	Per 8000 Samples	Per Frame	Per Samples
Compression	105000	5250000	5.25	1050
Decompression	55000	2750000	2.75	550
Total	160000	8000000	8.0	400

The processing time and the frame size of an algorithm determine the transcoding delay of the communication. The transcoding delay is the time interval between the instant speech frame of 160 samples has been received at the encoder input and the instant the corresponding 160 reconstructed speech samples have been output by the speech decoder at an 8 kHz sample rate. The transcoding delay required by [ETS94] is less than 30 ms.

From Table 4.4 it can be seen that the transcoding delay is lower than the requirement. The total time taken for sampling and processing each frame is around 28 ms. The assembly code has been optimised by reducing the number of cycles of the critical loops which are executed repeatedly. Tables I.3 and I.4 show that the majority of the cycles are taken up the short term analysis and synthesis filtering subblocks, `i_filter` and `s_filter` (see Appendix I). Other critical loops occurs in `offset`, `autocorr`, `lpc_calc` and `w_filter`. The total time required to process these subblocks have been minimised.

Further optimisation of the assembly code has been constrained by saturation in these two subblocks where overflow control is needed. In both subblocks, there is a pair of nested loops, where the inner has more than 50 iterations, and the outer 160 iterations, resulting in the inner loop being executed over 8000 times. A lot of saturation occurs in the inner loop and therefore an overflow check was need in each `ADD` or `MULT_R` (multiply with rounding) operation. This step was done by using only ALU instructions and since they could not be written in parallel, the number of cycles needed was increased from one instruction to about seven on average. It was important that these two subblocks were as optimised as possible, however, due to the problem with the overflow, the code could not be further reduced.

The optimisation can be improved by adding new instructions to GEAPRD with overflow control and saturation. For example, an `ADD` instruction that sets the result at +32767 when overflow occurs or at -32768 when underflow occurs. The instructions than require overflow check are: `ADD`, `SUB`, `MUL` and `ABS`. The overflow is needed, however, when a 32-bit arithmetic operation is performed using 16 bit registers. The only 32-bit arithmetic operation used in GSM 6.10 is long subtraction (`L_sub`).

4.7 Summary

The GSM 06.10 full-rate speech compression algorithm is a lossy technique which is based on a residually excited linear predictive coder (RELP) and this is further enhanced by using a long term predictor (LTP). This improves speech quality by removing the structure from the vowel sounds prior to coding the residual data. It compresses frames of 160 13-bit signed samples to 260-bit compressed frames.

The GSM full-rate speech transcoder has been implemented on GEPARD. The GEPARD code has been tested with all of the test sequences provided by ETSI and the results are bit-exact. The total time taken for sampling and processing each frame is around 28 ms and is lower than the ETSI requirement for transcoding delay, which is less than 30 ms. The code has been optimised by reducing the number of cycles taken by critical loops in the algorithm. The optimisation can be further improved by adding new instructions with overflow control and saturation. A comparison between this codec and the new half-rate speech algorithm will be discussed in the next chapter.

5 Comparison of GSM Full-Rate and Half-Rate Speech Codecs

5.1 Introduction

The existing GSM full-rate channel fulfils its quality goals for the mobile phone user most of the time. As the market for digital cellular telephones expands, non-mobile applications are emerging, some of which demand high speech quality even in difficult conditions. Even since the GSM full-rate standard was finalised, advances in speech coding technology have reduced the bandwidth required for toll quality compression to less than 13 kbps GSM full-rate codec. This can be seen with the advent of the new GSM half-rate speech codec.

ETSI specified the GSM half-rate codec with a bit rate of 5.6 kbps in 1995. The algorithm is based on Motorola's Vector Sum Excited Linear Prediction (VSELP) technology similar to IS-54 full-rate [KON95]. It uses two 7-bit codebooks for unvoiced speech and one 9-bit codebook for voiced segments. The algorithm is a lot more complex than the GSM full-rate. A brief description of the codec and a comparison to the full-rate codec can be found in later sections.

There is very little literature on the GSM half-rate codec that is publicly available. Therefore this comparison is based on the ETSI specification.

5.2 The VSELP Algorithm

The VSELP algorithm [SPA94] closely resembles the Codebook Excited Linear Prediction (CELP) algorithm. For more information about CELP see 3.6.2. The difference lies in the form and structure of the codebooks. Whereas CELP uses a stochastically overlapped codebook, VSELP utilises two sets of basis vectors with a predefined structure such that a brute-force search can be avoided. The stochastic codecook search of CELP corresponds to two codebook searches in VSELP. There are seven basis vectors for each search. Each basis vector contains 40 elements. The selection of the basis vectors is fundamental to deriving fast codebook search procedures. The basis vectors with a vector **V**, the entire 128 (2^7) space, defined by the seven basis vectors, is also orthogonalised.

An open-loop LPC analysis is performed on a frame of speech to derive a set of LPC filter coefficients. These coefficients are bandwidth expanded for use in perceptual error weighting filters, $H(z)$ and $W(z)$, defined in (5.1) and (5.2). The input frame of speech is

filtered through the filter $W(z)$ to obtain a perceptually weighted frame of speech. Perceptual weighting of the input speech improves the performance of the coder. The high-energy formant regions of the speech spectrum mask noise better than lower energy portions of the spectrum. The error signal generated by each synthesiser pass is weighted appropriately to capitalise on the perceptual effect. The filter simplifies the error signal spectrum in non-formant regions of the speech spectrum and attenuates the error signal spectrum in formant regions.

The analysis by synthesis proceeds with three codebooks. First, the adaptive codebook is searched and the resulting best entry and gain are found. This entry multiplied by its gain factor is orthogonalised with the first set of seven basis vectors. Thus, the second codebook search can be performed independently of the first codebook search. The new set of basis vectors is used from the codebook for the second set of basis vectors. Finally the third codebook search is performed. The gains of each of the three codebook searches are jointly quantised and transmitted with the three codebook indices to the receiver.

The basic blocks in the VSELP coder are

- Tenth-order LPC analysis
- Long term predictor
- Adaptive codebook search
- First basis vector codebook search
- Second basis vector codebook search
- Vector quantisation of the codebook gains

The VSELP algorithm was developed by Motorola and the Electronics Industries Association [HIL97].

5.3 Overview of Half-Rate Speech Transcoding (GSM 06.20)

The GSM half-rate speech codec uses the VSELP algorithm, which is an analysis-by-synthesis coding technique and belongs to Code Excited Linear Prediction (CELP). The encoding process is performed on 20ms of speech at a time. A speech frame of the sampled speech signal is read and based on the current signal and the past history of the signal. The encoder derives 18 parameters that describes the speech in three general classes:

- energy parameters (R0 and GSP0);
- spectral parameters (LPC and INT_LPC);

- excitation parameters (LAG and CODE).

These parameters are quantised into 112 bits for transmission. Since the codec uses the analysis-by-synthesis technique, the speech decoder is primarily a subset of the speech encoder. The quantised parameters are decoded and synthetic excitation is generated using

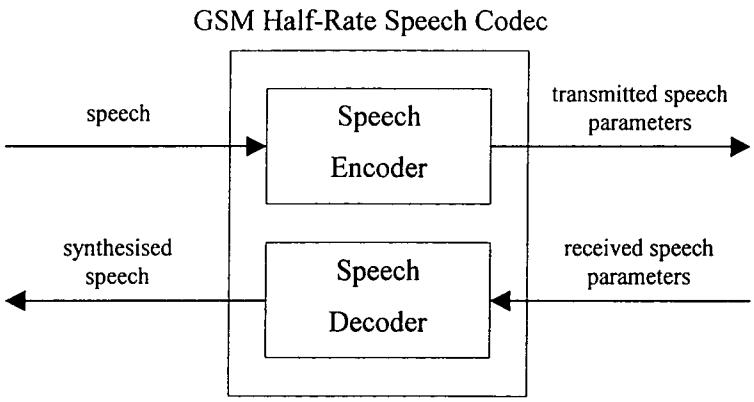


Figure 5.1 Block Diagram of the GSM half-rate speech codec.

the energy and excitation parameters. The synthetic excitation is then filtered to provide the spectral information resulting in the generation of the synthesised speech as shown in Figure 5.1. A detailed explanation of the codec can be found in [ETS95].

5.4 GSM Half-Rate Speech Encoder

A block diagram of the GSM half rate speech encoder is shown in Figure 5.2. The encoder uses an analysis-by-synthesis approach to determine the code to use to represent the excitation for each subframe. The codebook search procedure consists of trying each codevector as a possible excitation for the CELP synthesiser. The synthesised speech $s'(n)$ is compared against the input speech and a difference signal is generated. The difference signal is then filtered by a spectral weighting filter, $W(z)$, (and possibly a second weighting filter, $C(z)$ to generate a weighted error signal, $e(n)$. The power in $e(n)$ is computed. The codevector which generates the minimum weighted vectors is chosen as the codevector for that subframe. The spectral weighting filter serves to weight the error spectrum based on perceptual considerations. This weighting filter is a function of speech spectrum and can be expressed in terms of the α parameters of the short term (spectral) filter.

$$W(z)=\frac{1-\sum_{i=1}^{N_p}\alpha_i z^{-1}}{1-\sum_{i=1}^{N_p}\tilde{\alpha}_i z^{-1}} \tag{5.1}$$

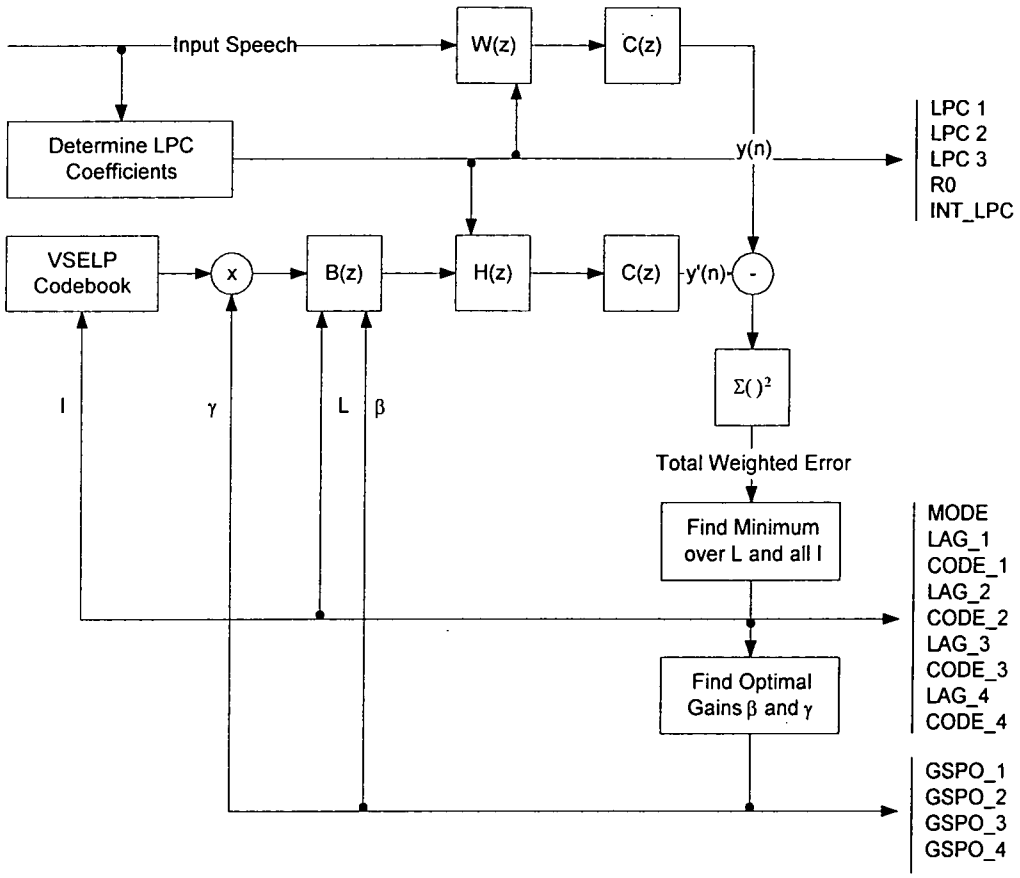


Figure 5.2 Block diagram of the GSM Half Rate Speech Encoder (MODE = 1, 2 and 3).

The second weighting filter $C(z)$, if used, is a harmonic weighting filter and is used to control the amount of error in the harmonics of the speech signal. $H(z)$ is a combination of $A(z)$, the short term (spectral) filter, and $W(z)$, the long term spectral filter.

$$H(z) = \frac{1}{\sum_{i=1}^{N_p} \tilde{\alpha}_i z^{-1}} \tag{5.2}$$

There are two approaches that can be used to calculate the gain, γ. The gain can be determined prior to codebook search based on residual energy. This gain would then be fixed for the codebook search. The other approach is to optimise gain for each codevector during the codebook search. The codevector which yields the minimum weighted error would be chosen and its corresponding optimal gain would be used for γ. This approach yields better results.

5.5 GSM Half-Rate Speech Decoder

The speech decoder is a subset of the speech encoder. The quantised parameters are decoded and a synthetic excitation is generated using the energy and excitation parameters. The synthetic excitation is then filtered to provided the spectral information resulting in the generation of the synthesised speech.

A block diagram of the GSM half rate speech decoder for MODE = 1, 2, or 3 is shown in Figure 5.3. The speech decoder creates the combined excitation signal, $ex(n)$, from the long

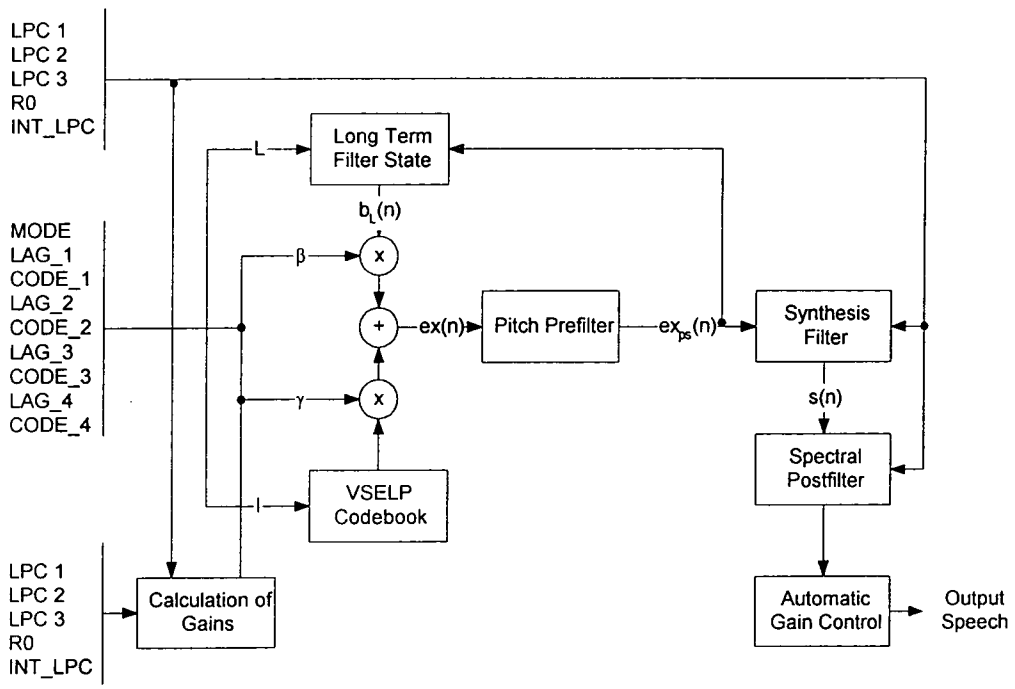


Figure 5.3 The GSM half rate speech decoder for MODE = 1, 2 and 3.

term filter state and the VSELP codevector. For MODE = 0, the long term filter state is replaced by another VSELP codebook and the pitch prefilter is not used. The combined excitation is then processed by an adaptive pitch prefilter and gain. The prefiltered excitation is applied to the LPC synthesis filter. After reconstructing the speech signal with the synthesis filter, an adaptive spectral postfilter is applied followed by an automatic gain control which is the final processing step in the speech decoder.

5.6 A Comparison to Full-Rate Codec

Both GSM full-rate and half-rate speech compression offer high speech quality and low cost. The two algorithms are now compared. The speech quality, complexity and cost of

implementation will be discussed. A brief summary of the characteristics of the two algorithms in shown in table 5.1.

Table 5.1 A comparison between GSM full-rate and half-rate codecs.

Coder	Full-Rate	Half-Rate
Sampling Rate	8 kHz	8 kHz
Frame Length	160 samples (20 ms)	160 samples (20 ms)
Subframe Length	40 samples (5 ms)	40 samples (5 ms)
Bit Rate	13 kbps	5.6 kbps
Short Term Predictor Order	8	10
Coding Technique	RPE-LTP	VSELP
Complexity	--	4 times full-rate
Speech Quality	near toll	near toll

5.6.1 Quality of Speech

Speech quality is a very important criterion for algorithm requirement as explained in chapter 3. Since all narrow band speech compression algorithms are lossy, speech quality generally degrades as the bit rate decreases. The analogue bandwidth supported by a voice coder also directly affects its speech quality, e.g. telephone bandwidth. An algorithm's speech quality is a function of its bit rate and its mathematical approach.

However, algorithm quality and bit rate are not linearly not linearly related [KLE97]. Algorithms that produces twice the bit rate do not necessary provide twice the quality. GSM full-rate codec operates at 13 kbps and half-rate at 5.6 kbps, however, they both offer near toll quality speech comparable to or better than analogue cellular networks [HIL97]. There is a demonstration of both full-rate and half-rate codecs on the World Wide Web at http://www.eas.asu.edu/~speech/table.html.

5.6.2 Complexity of Algorithm

The GSM half-rate coder is four times as complex as that of full-rate [HIL97]. Both coders derive the output parameters by reading a frame of the sampled speech waveform and using the current waveform and past history of waveform. As previously stated, the encoded parameters of the full-rate coder are

- the short term filter parameters

- the long term prediction (LTP) parameters
- the RPE parameters

These parameters are quantised into 260 bits for transmission. The encoder output parameters are shown in Table C.1. The encoded parameters of the half-rate coder are

- energy parameters (R0 and GSP0)
- spectral parameters (LPC and INT_LPC)
- excitation parameters (LAG and CODE)

These parameters are quantised into 112 bits for transmission. The encoder output parameters and a brief description are shown in and Appendix E.

The input speech frame of both coders consists of uniform 13-bit PCM signed samples converted from 8-bit A-law companded format. For the full-rate coder, the speech signal is initially proprocessed. After A-law to linear conversion (or directly from the A to D converter) the following input sample (in two's complement format) is obtained,

S.v.v.v.v.v.v.v.v.v.v.v.v.x.x.x

where **S** is the signed bit, **v** a valid bit and **x** a "don't care" bit [ETS94]. The input samples are down-scaled by a factor of two and a notch filter is applied in order to remove the offset of the signal s_0 to produce the offset-free signal s_{of}.

$$s_{of}(k) = s_0(k) - s(k-1) + \alpha s_{of}(k-1) \tag{5.3}$$

where $\alpha = 32735 \times 2^{-15}$. The signal s_{of} is then applied to a first order FIR pre-emphasis filter leading to the input signal s of the analysis section,

$$s(k) = s_{of}(k) - \beta s_{of}(k-1) \tag{5.4}$$

where $\beta = 28180 \times 2^{-15}$. The speech signal $s(k)$ is divided into non-overlapping frames having a period of 20 ms (16 samples).

For the half-rate speech coder, the 13 bit linear PCM input speech, $x(n)$, is filtered by a fourth order pole-zero high pass filter which suppress frequencies below 120 Hz. The filter is implemented as a cascade of two second-order Infinite Impulse Response (IIR) filters [ETS95]. Incorporated into the filter coefficients is a gain of 0.5. The difference equation for the first filter is

$$\tilde{y}(n) = \sum_{i=0}^{2} b_{1i} x(n-i) + \sum_{j=1}^{2} a_{1j} y(n-j) \tag{5.5}$$

where b_{10} = 0.335052, b_{11} = -0.669983, b_{12} = 0.335052, a_{11} = 0.926117, a_{12} = -0.429413. The difference equation for the second filter is

$$y(n) = \sum_{i=0}^{2} b_{2i} y(n-i) + \sum_{j=1}^{2} a_j y(n-j) \tag{5.6}$$

where b_{10} = 0.335052, b_{11} = -0.669434, b_{12} = 0.335052, a_{11} = 0.965332, a_{12} = -0.469513.

A sample buffer containing the previous 195 input high pass filtered speech samples, $y(n)$, is shifted so that the oldest 160 samples are shifted out while the next 160 input

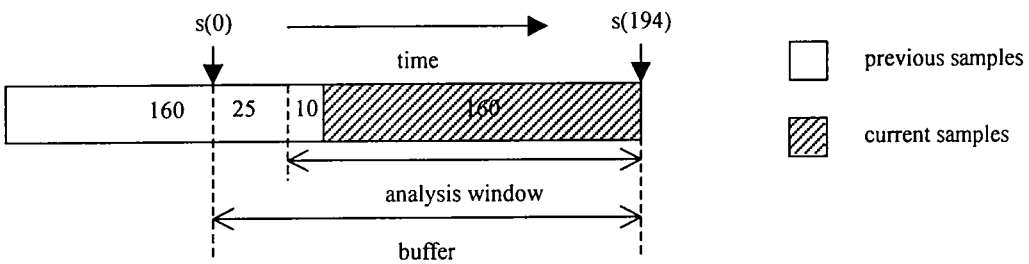


Figure 5.4 Segmentation of speech signal.

samples are shifted in. The oldest 160 samples in the buffer correspond to the next frame of samples to be encoded. The analysis interval comprises the most recent 170 samples in the buffer [ETS95]. The samples in the buffer are labelled as $s(n)$ where $0 \le n \le 194$ and $s(0)$ is the oldest sample as shown in Figure 5.4.

The half-rate algorithm introduces new routines fixed point lattice technique (FLAT) and autocorrelation fixed point lattice technique (AFLAT) [ETS95] which requires the divide operation that is not included in most DSPs.

Table 5.2 Windowing coefficients for the FLAT algorithm.

$w(0)$	0.998966	$w(5)$	0.974915
$w(1)$	0.996037	$w(6)$	0.969054
$w(2)$	0.991663	$w(7)$	0.963060
$w(3)$	0.986399	$w(8)$	0.956796
$w(4)$	0.980722	$w(9)$	0.950127

The FLAT algorithm determines the reflection coefficients. Let r_j be the jth reflection coefficient. The procedure is shown in Figure 5.5. The windowing coefficients, $w(|i-k|)$, are found in Table 5.2.

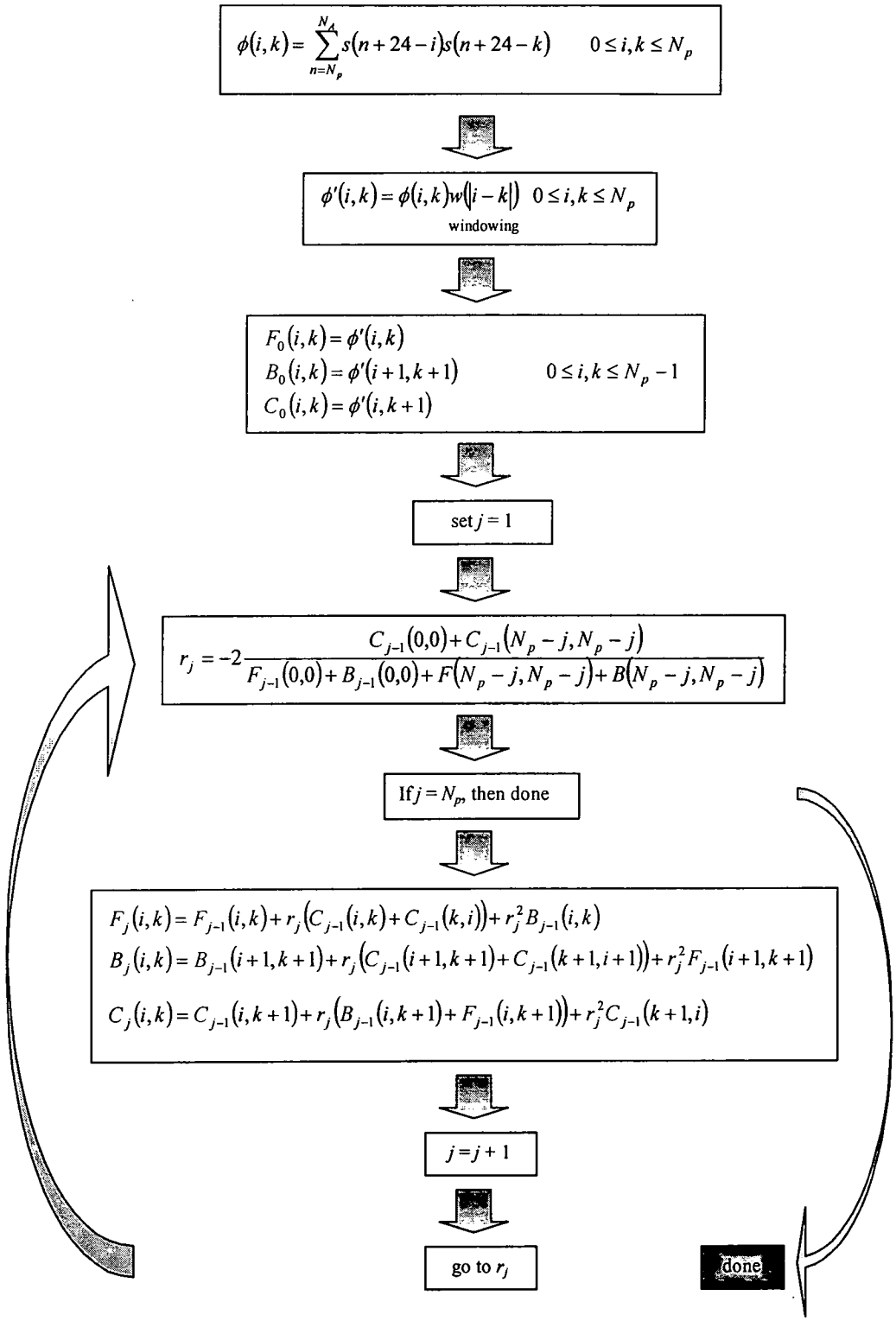


Figure 5.5 Fixed Point Lattice Technique (FLAT).

The algorithm can be simplified by noting that the ϕ', F, and B correlation matrices are symmetric such that only the upper triangular part of the matrices need to be computed or updated.

An autocorrelation version of the FLAT algorithm, AFLAT, is used to compute the residual error energy for a reflection coefficient vector being evaluated. The autocorrelation

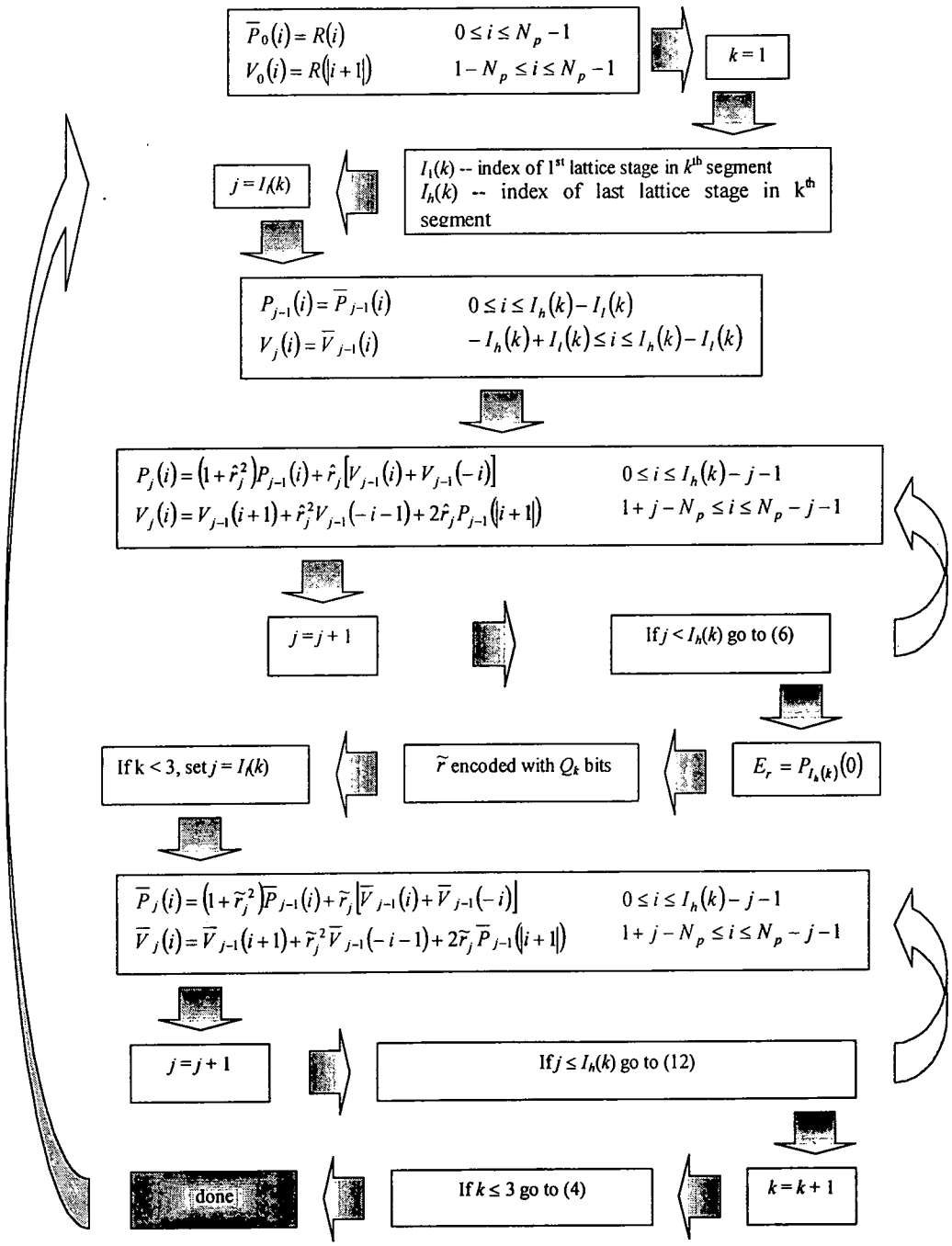


Figure 5.6 Autocorrelation Fixed Point Lattice Technique (AFLAT).

sequence $R(i)$, is computed from the optimal reflection coefficients, r_j, over the range $0 \le i \le N_p$. The procedure is shown in Figure 5.6.

In order to minimise the storage requirements for the reflection coefficient vector quantiser, eight bit codes for the individual reflection coefficients are stored in the vector

quantiser table, instead of the actual reflection coefficient values. The codes are used to look up the values of the reflection coefficients from a scalar quantisation table with 256 entries.

The full-rate coder uses LPC analysis to find the eight reflection coefficients. They are converted to log area ratios. Since the LAR parameters have different dynamic ranges and probability distribution functions, they are encoded with different number of bits. A description of the procedure is shown in Figure A.1.

The half-rate coder also uses the voicing mode selection which is not used in the full-rate coder. The new voiced and unvoiced modes reduces the bit rate needed but makes the algorithm more complex. A multimode gain {*P*0,*GS*} codebook [ETS95], where *P*0 is the power contribution of the pitch prediction vector as a fraction of the total excitation power at a subframe, and *GS* is the energy tweak factor which bridges the gap between the actual energy in the coder excitation and its estimated value, contains the values needed to determine the gain factors for the excitation vectors of a given subframe. The index of the corresponding codebook entry is assigned to *GSP*0_*x*. The half-rate coder is a multimode speech coder, defined by four voicing modes

MODE = 0 unvoiced

MODE = 1 slightly voiced

MODE = 2 moderately voiced

MODE = 3 strongly voiced

If MODE = 0, the adaptive codebook (long-term predictor) and the VSELP codebook are replaced by two other VSELP codebooks [ETS95].

The half-rate uses a lot of intensive codebook search as explained in 5.2. The vector quantisation used is also a computationally intensive operation (see 2.6.4). In order to reduce the coding delay, the half-rate coder requires four times the computing power of the full-rate. Its high circuit complexity and component count will also likely compromise system reliability.

5.6.3 Cost of Implementation

System cost is very important in the selection an algorithm. A product's market typically determines the product's price which determines the amount of MIPS, RAM and ROM that can be used by the product's subsystem [WON96]. Algorithm resource requirements vary drastically. Products that require standard algorithm must be able to compete in the market given the costs associated with that algorithm's resource requirements.

The half-rate codec requires a lot of internal memory for data storage such as the coefficients used in the algorithm and the previous samples needed to process the current frame, which some of these are not required in the full-rate codec. The power consumption will be very high with the powerful processor in order to cope with the complexity of the code. These resource requirements will considerably increase the cost of implementation of the half-rate codec compared with the full-rate.

5.7 Summary

A brief description of VSELP coding technique and the GSM half-rate speech codec is given in this chapter. ETSI specified the GSM half-rate codec with a bit rate of 5.6 kbps in 1995. The algorithm is based on Motorola's VSELP technology similar to IS-54 full-rate. It uses two 7-bit codebooks for unvoiced speech and one 9-bit codebook for voiced segments. A comparison to the full-rate algorithm is discussed.

The half-rate algorithm takes advantage of more efficient speech compression than full-rate techniques do, shrinking the bandwidth timeslice that each user requires. Although the full-rate codec operates at 13 kbps and half-rate at 5.6 kbps, they both offer near toll speech quality comparable or better than analogue cellular networks. The half-rate compression introduces new algorithms such as FLAT and AFLAT and requires more computationally intensive operations compared with the full-rate. Hence a four times more powerful processor will be needed for the half-rate codec. The cost of the implementation of half-rate codec will also be considerably higher than full-rate. In the next chapter, a multirate approach to speech coding will be investigated.

6 The Application of Multirate Techniques for Speech Coding

6.1 Introduction

In many practical applications of digital signal processing (DSP), one is faced with the problem of changing the sampling rate of signals, either increasing or decreasing them by some amount. In telecommunication systems that transmit and receive different speech source, there is a requirement to process the various data at different frequencies that involves sample rate conversion [PRO92]. For example, a 1 kHz speech signal sampled at 2 kHz is four times more efficient than one sampled at 8 kHz.

This chapter investigates a multirate optimisation technique in digitised speech that is based on the idea of eliminating redundant computation. In speech processing, multirate techniques can be used to reduce the storage space required or the transmission rate of speech data. Estimates of speech parameters are computed at a very low sampling rate for storage or transmission. When required, the original speech is reconstructed from the low bit rate representation at much higher rates using the multirate approach.

6.2 Multirate Digital Signal Processing

A simple way to changing the sampling rate of a digital signal is to convert it back into analogue and then to redigitise it at the new rate. Errors inherent in digital-analogue-digital conversion processes, such as quantisation and aliasing errors, would degrade the signal. As the signal is already in a digital form, it is best to process it digitally throughout until conversion to analogue is mandatory. Multirate processing [CRO93] [IFE93] is an efficient technique for changing the sampling frequency of a signal digitally.

The processes of decimation and interpolation are the fundamental operations in multirate signal processing, and they allow the sampling frequency to be decreased or increased without significant, undesirable effects of errors such as quantisation and aliasing.

The process of sampling rate conversion in the digital domain can be viewed as a linear filtering operation as illustrated in Figure 6.1. The input signal $x(n)$ is characterised by the sampling rate $F_x = 1/T_x$ and the output signal $y(m)$ is characterised by the sampling rate $F_y = 1/T_y$, where T_x and T_y are the corresponding sampling intervals. The ratio of F_x/F_y is constrained to be rational,

$$\frac{F_x}{F_y} = \frac{L}{M} \tag{6.1}$$

where M and L are relatively prime integers. The linear filter can be shown to be characterised by a time-variant impulse response, $h(n,m)$. Hence the input $x(n)$ and the output $y(m)$ are related by the convolution summation for time-variant systems. The two

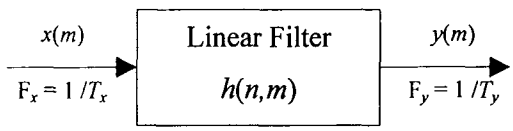


Figure 6.1 Sampling rate conversion viewed as a linear filtering process.

special cases of sampling rate conversion are the process of reducing the sampling rate by a factor M (downsampling by M), called decimation, and the process of increasing the sampling rate by a factor L (upsampling by L) is called interpolation [CRO93].

6.2.1 Filter Design for Multistage Approach to Sampling Rate Conversion

When large changes in the sampling rate are required it is more efficient to change the rate in two or more stages than in one stage as described previously. The performance of a multirate system depends critically on the type and quality of the filter used. A digital filter for anti-alisaing in sampling rate converters is needed. Either finite impulse response (FIR) or infinite impulse response (IIR) filters can be used for decimation or interpolation.

In multirate processing, the computation efficiency of the FIR filter is comparable with that of IIR filters. Furthermore, FIR filters have many desirable attributes such as linear phase response, low sensibility to finite word length effects and simple to implement [IFE93]. Therefore, FIR filters are generally used. The implementation of a FIR filter is

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) \tag{6.2}$$

where $x(n-k)$ are sampled data inputs are sampled data inputs, $y(n)$ is the filtered output, and $h(k)$ are the filter coefficients that together represent the impulse response.

For an FIR filter, the overall requirements for decimation, to avoid alisaing after rate reduction are,

passband	$0 \geq f \geq f_p$	(6.3a)
stopband	$F_s/M \geq f \geq F_s/2$	(6.3b)
passband deviation	δ_p	(6.3c)
stopband deviation	δ_s	(6.3d)

where $f_p < F_s/2M$, and F_s is the original sampling frequency and f_p is the highest frequency of interest in the original signal.

In the case of interpolation, the anti-imaging filter must remove all but the useful information by bandlimiting the modified data to $F_s/2$ or less. Although the highest valid frequency after raising the rate to F_s is $LF_s/2$, according to the sampling theorem, it is necessary to bandlimit to $F_s/2$ as this is the highest valid frequency in $x(n)$. The overall requirements for interpolation are

passband	$0 \leq f \leq f_p$	(6.4a)
stopband	$F_s/M \leq f \leq F_s/2$	(6.4b)
passband deviation	δ_p	(6.4c)
stopband deviation	δ_s	(6.4d)

where $f_p < F_s/2$. A gain of L is necessary in the passband to compensate for the amplitude reduction by the interpolation process.

6.2.2 Filter Requirement for Individual Stages

The optimal filter [IFE93] is often used for sampling rate conversion. The tolerance scheme for an optimal lowpass filter is depicted in Figure 6.2(a).

For a multistage decimator shown in Figure 6.2(c) the filter requirements for each stage to ensure that the overall filter requirements (shown in Figure 6.2(b)) are met are

passband	$0 \leq f \leq f_p$	(6.5a)
stopband	$(F_i - F_s/2M) < f < F_{i-1}/2,\ i = 1, 2, \ldots, I$	(6.5b)
passband deviation	δ_p/I	(6.5c)
stopband deviation	δ_s	(6.5d)
filter length	$N \approx \frac{D_\infty(\delta_p, \delta_s)}{\Delta f_i} - f(\delta_p, \delta_s)\Delta f_i + 1$	(6.5e)

where F_i, N_i and Δf_i are the output sampling frequency, the filter length and the normalised transition width for the ith-stage decimator respectively. The parameters $D_\infty(\delta_p,\delta_s)$ and $f(\delta_p,\delta_s)$ are

$$D_\infty(\delta_p,\delta_s)=(\log_{10}\delta_s)\left[a_1(\log_{10}\delta_p)^2+a_2(\log_{10}\delta_p)+a_3\right]+a_4(\log_{10}\delta_p)^2 + a_5(\log_{10}\delta_p)+a_6 \tag{6.6a}$$

$$f(\delta_p,\delta_s)=11.01217+0.51244(\log_{10}\delta_p-\log_{10}\delta_s) \tag{6.6b}$$

where

$$a_1 = 5.309\times10^{-3}; \quad a_2 = 7.114\times10^{-2};$$

$$a_3 = -4.761\times10^{-1}; \quad a_4 = -2.660\times10^{-3};$$

$$a_5 = -5.941\times10^{-1}; \quad a_6 = -4.278\times10^{-1}.$$

The output sampling frequency for stage i is given by

$$F_i = F_{i-1}/M_i \tag{6.7}$$

where i = 1, 2, ..., I and M_i is the decimation factor for the stage. The initial and final sampling rates are $F_0 = F_s$ and $F_I = F_s/M$ respectively.

For multistage decimation, a lower passband deviation is necessary for each stage to ensure that the overall passband deviation is δ_p. The stopband deviation for each stage is the

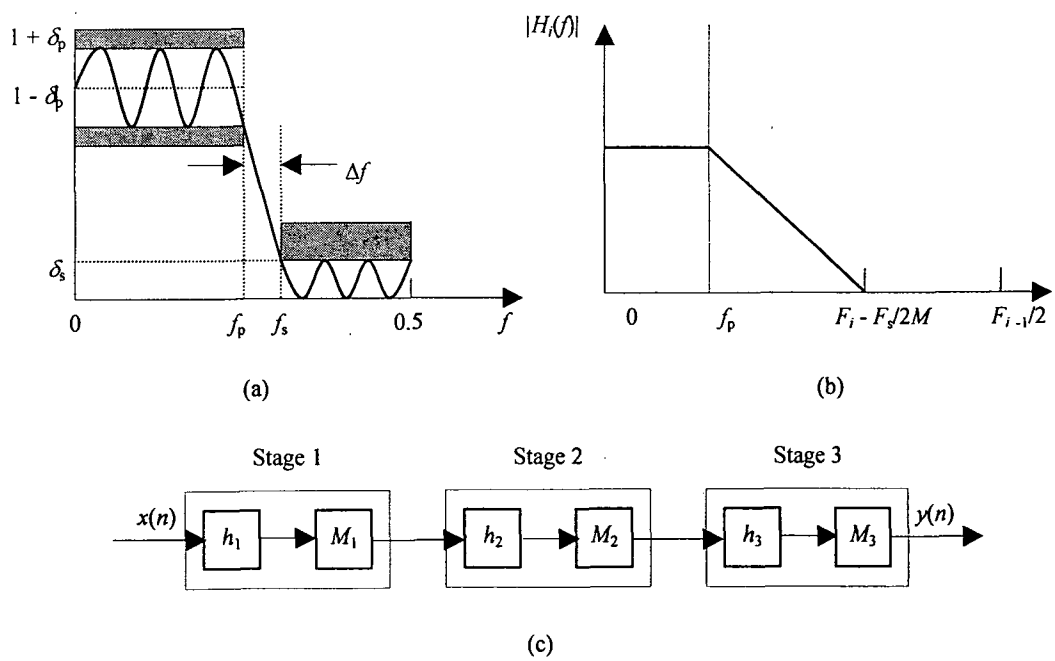


Figure 6.2 Filter requirement. (a) Tolerance scheme for an optimal lowpass filter; (b) multistage structure; (c) filter specifications for stage i, i = 1, 2, ..., I.

same as the overall stopband deviation because as the signal goes from stage to stage the stopband components are attenuated further. For a one-stage decimator, the filter requirements are the same as (6.4).

6.3 Subband Coding of Speech Signals

A variety of techniques have been developed to efficiently represent speech signals in digital form for either transmission or storage as described in the previous chapters. Since most of the speech energy is contained in the lower frequencies, more bits are needed to encode the lower-frequency band than the high-frequency band. The speech signal can be subdivided into several frequency bands allowing each band to be digitally encoded separately.

A block diagram of a subband speech coder coder [PRO92] [CRO93] [MARV93] is shown in Figure 6.3. Let the sampling frequency of the speech signal F_s kHz. The first frequency subdivision splits the signal spectrum into two equal-width segments, a lowpass signal ($0 \leq F \leq F_s/4$) and a high pass signal ($F_s/4 \leq F \leq F_s/2$). The second frequency

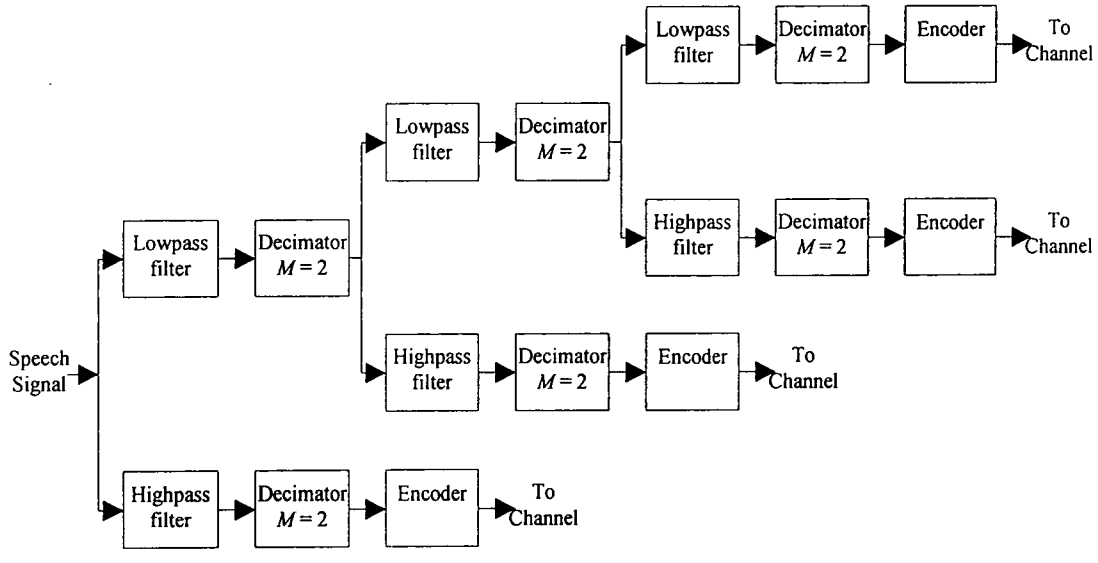


Figure 6.3 Block diagram of a subband speech coder.

subdivision splits the lowpass signal from the first stage into two equal bands, a low pass signal ($0 \leq F \leq F_s/8$) and a highpass signal ($F_s/8 \leq F \leq F_s/4$). Finally, the third frequency subdivision splits the lowpass signal from the second stage into two equal bandwidth signals. Thus the signal is subdivided into four frequency bands, covering three octaves, as shown in Figure 6.4.

Decimation by a factor of two is performed after frequency subdivision. By allocating a different number of bits per sample to the signal in the four subbands, a reduction in the bit rate of the digitised speech signal can be achieved.

Filter design [WIS96] is particularly important in achieving good performance. Aliasing resulting from decimation of the filtered signal must be negligible. Quadrature mirror filters (QMF) [SPO92] are usually used as brickwall filters. The frequency response of such a filter is shown in Figure 6.5.

The synthesis method for the subband encoded speech signal is the reverse of the encoding process. The signal adjacent lowpass and highpass frequency bands are

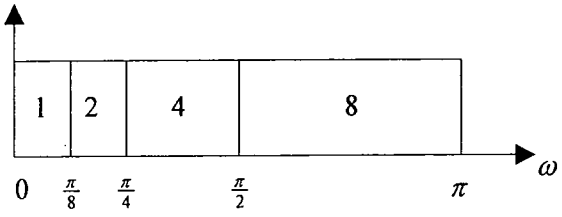
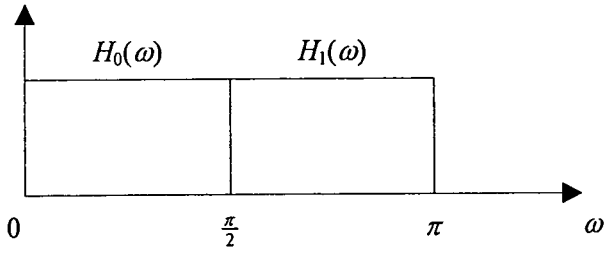


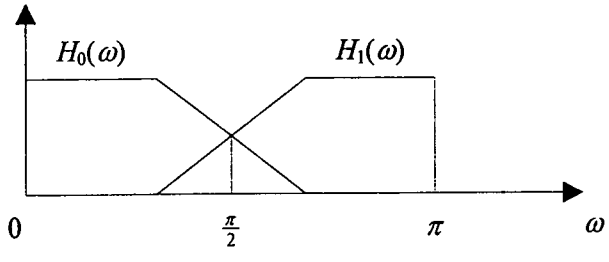
Figure 6.4 The frequency bands of the subband speech coder.

interpolated, filtered and combined as shown in Figure 6.6. A pair of QMF is used in the signal synthesis for each octave of the signal.

Subband coding of signals is an effective method for achieving bandwidth compression in a digital representation of the signal, when the signal energy is concentrated in a particular



(a) Brickwall filters



(a) QMF

Figure 6.5 Filter characteristics for subband coding.

region of the frequency band. Multirate signal processing notions provide efficient implementations of the subband encoder. It was considered worthwhile to investigate its use

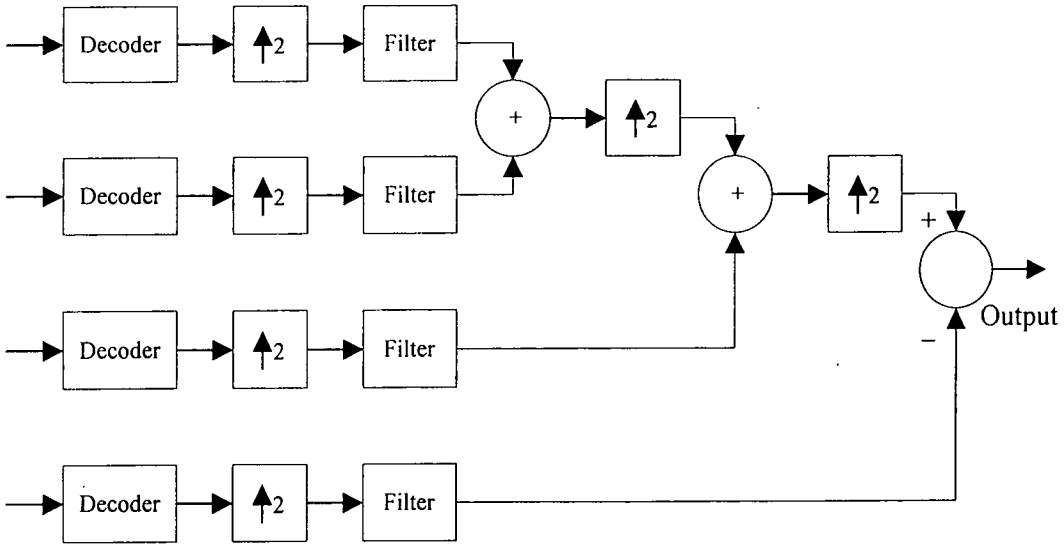


Figure 6.6 Synthesis of a subband-encoded speech signal.

in speech coding, and in particular their combination with other audio compression techniques. More information on subband coding can be found in 6.5.

6.4 MPEG Audio Compression

A discussion of audio compression would not be complete without a consideration of the Motion Picture Experts Group (MPEG) standard [PAN93] [PAN95]. Over the last five to ten years, subband coding systems have been developed by many of the key companies and laboratories in the audio industry. Beginning in the late 1980's, a standardisation body of the International Standard Organisation (ISO) called MPEG developed generic standards for coding of both audio and video.

MPEG audio is a group of three different SBC schemes, called layers. Each layer is a self-contained SBC coder with its own time-frequency mapping, psychoacoustic model, and quantiser, as shown in the Figure 6.7. Layer 1 is the simplest, but gives the poorest compression. Layer 3 is the most complicated and difficult to compute, but gives the best compression. An application of MPEG audio can use whichever layer that gives the best tradeoff between computational burden and compression performance. Audio can be encoded in any one layer. A standard MPEG decoder for any layer is also able to decode lower layers of encoded audio. MPEG audio is intended to take a PCM audio signal

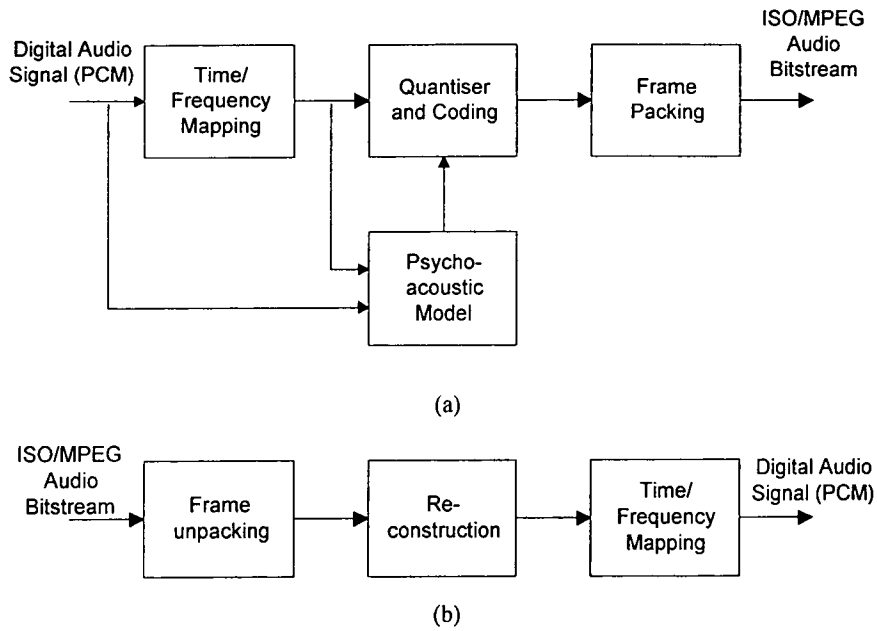


Figure 6.7 MPEG. (a) Encoder. (b) Decoder.

sampled at a rate of 32, 44.1 or 48 kHz, and encode it at a bit rate of 32 to 192 kbps per audio channel depending on layer.

MPEG encoders use time-frequency mapping to decompose the input signal into subbands. The psychoacoustic model looks at these subbands as well as the original signal, and determines masking thresholds using psychoacoustic information. Using these masking thresholds, each of the subband samples is quantised and encoded so as to keep the quantisation noise below the masking threshold. The final step is to assemble all these quantised samples into frames, so that the decoder can figure it out without getting lost.

There is no need for a psychoacoustic model in the decoder. The frames are unpacked, subband samples are decoded, and finally turned back into a single output audio signal by a frequency-time mapping.

A simple block diagram of the process is shown in Figure 6.7. For practical systems that need to run in real time, computation is a major issue, and is usually the main constraint.

6.4.1 MPEG Audio Layer 1

The layer 1 [PAN95] time-frequency mapping is a polyphase filter bank with 32 subbands. Polyphase filters combine low computational complexity with flexible design and implementation options. However, the subbands are equally spaced in frequency.

The layer 1 psychoacoustic model uses a 512-point fast fourier transform (FFT) to get detailed spectral information about the signal. The output of the FFT is used to find both

tonal (sinusoidal) and non-tonal (noise) *maskers* in the signal. Each masker produces a masking threshold depending on its frequency, intensity, and tonality. For each subband, the individual masking thresholds are combined to form a global masking threshold. The masking threshold is compared to the maximum signal level for the subband, producing a signal-to-masker ratio (SMR) which is the input to the encoder.

The layer 1 encoder first examines each subband's samples, finds the maximum absolute value of these samples, and quantizes it to 6 bits. This is called the scale factor for the subband. Then it determines the bit allocation for each subband by minimising the total noise-to-mask ratio with respect to the bits allocated to each subband. It is possible for heavily masked subbands to end up with zero bits, so that no samples are encoded. Finally, the subband samples are linearly quantised to the bit allocation for that subband.

Layer 1 processes the input signal in frames of 384 PCM samples. At 48 kHz, each frame carries 8 ms of sound. The MPEG specification does not specify the encoded bit rate, allowing implementation flexibility. Highest quality is achieved with a bit rate of 384k bps. Typical applications of layer 1 include digital recording on tapes, hard disks, or magneto-optical disks, which can tolerate high bit rate.

6.4.2 MPEG-1 Audio Layer 2

The layer 2 [PAN95] time-frequency mapping is the same as in Layer 1 which is a polyphase filter bank with 32 subbands.

The layer 2 psychoacoustic model is similar to the Layer 1 model, but it uses a 1024-point FFT for greater frequency resolution. It uses the same procedure as the Layer 1 model to produce signal-to-masker ratios for each of the 32 subbands.

The layer 2 encoder is similar to that used in Layer 1. It generates 6-bit scale factors for each subband. However, layer 2 frames are three times as long as layer 1 frames, so layer 2 allows each subband a sequence of three successive scale factors, and the encoder uses one, two, or all three, depending on how much they differ from each other. This gives a factor of 2 reduction in bit rate for the scale factors on average compared to layer 1. Bit allocations are computed in a similar way to layer 1.

The layer 2 frame packer uses the same header and CRC structure as layer 1. The number of bits used to describe bit allocations varies with subband with 4 bits for the low subbands, 3 bits for the middle subbands, and 2 bits for the high subbands. The scale factors, one, two or three depending on the data, are encoded along with a 2-bit code describing which combination of scale factors is being used. The subband samples are quantised according to bit allocation, and then combined into groups of three, called

granules. Each granule is encoded with one code word. This allows layer 2 to capture much more redundant signal information than layer 1.

Layer 2 processes the input signal in frames of 1152 PCM samples. At 48 kHz, each frame carries 24 ms of sound. Highest quality is achieved with a bit rate of 256 kbps, but quality is often good down to 64 kbps. Typical applications of Layer 2 include audio broadcasting, television, consumer and professional recording, and multimedia.

6.4.3 MPEG-1 Audio Layer 3

Layer 3 [PAN95] is substantially more complicated than layer 2. It uses both polyphase and discrete cosine transform filter banks, a polynomial prediction psychoacoustic model, and sophisticated quantisation and encoding schemes allowing variable length frames. The frame packer includes a bit reservoir which allows more bits to be used for portions of the signal that need them.

Layer 3 is intended for applications where a critical need for low bit rate justifies the expensive and sophisticated encoding system. It allows high quality results at bit rates as low as 64 kbps. Typical applications are in telecommunication and professional audio, such as commercially published music and video.

6.5 Application of Multirate Processing to GSM Full-Rate Speech Codec

From the discussions in this and previous chapters, it was considered appropriate to apply multirate filtering to GSM full-rate speech compression. It was hoped that higher data compression ratio will be obtained by this method, hence a lower bit rate and a decrease in required network bandwidth. The results of this investigation was compared with conventional GSM coding.

6.5.1 Experimental Procedure

Various speech samples were passed through a subband speech coder implemented in MATLAB. The initial implementation was kept as simple as possible. A block diagram of the subband analysis filter used is shown in Figure 6.8. The sampling frequency of the speech signals F_s was 8 kHz. The signal spectrum is divided into 3 frequency bands, a lowpass signal ($0 \le F \le F_s/8$), a bandpass signal ($F_s/8 \le F \le F_s/4$) and a highpass signal ($F_s/4 \le F \le F_s/2$).

Decimation by factors of two and four is performed in the bandpass and lowpass frequency band respectively. Filter design is particularly important in achieving good performance. Aliasing resulting from decimation of the filtered signal must be negligible.

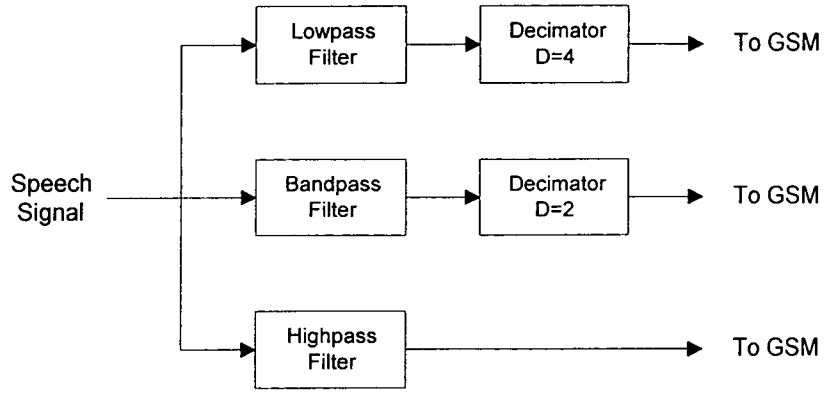


Figure 6.8 Block diagram of the subband coding analysis filter.

Three 10^{th} order FIR filters were designed with bandpasses from 0 to 1 kHz, 1 to 2 kHz and 2 to 4 kHz to separate the different frequencies. The filter was created with the `FIR1` command in MATLAB,

```
A1=[1 0 0 0 0 0 0 0 0 0];
B1=fir1(10,0.25);
x1=filter(B1,A1,s);
A2=[1 0 0 0 0 0 0 0 0 0];
B2=fir1(10,[0.25 0.5]);
x2=filter(B2,A2,s);
A3=[1 0 0 0 0 0 0 0 0 0];
B3=fir1(10,0.5,'high');
x3=filter(B3,A3,s);
```

The frequency response of the three filters is shown in Figure 6.9. The output of the three subbands was read by the bit-exact version of the GSM full-rate C encoder written for the

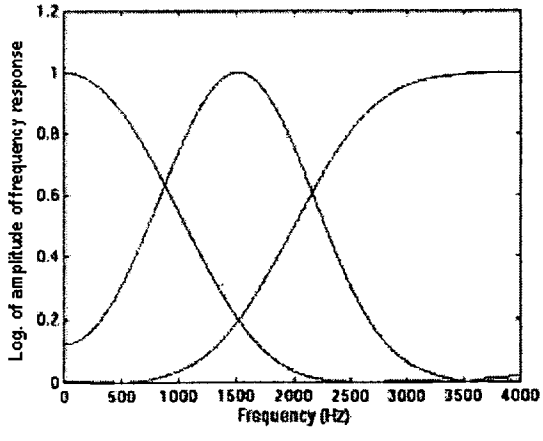


Figure 6.9 Frequency response of the 1 kHz, 2kHz and 4 kHz filters.

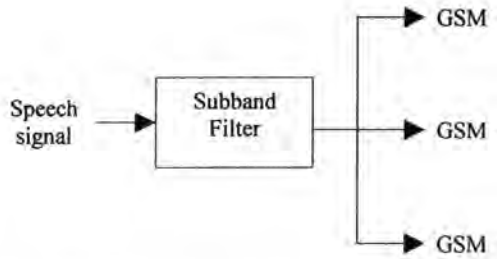


Figure 6.10 Architecture of the multirate GSM full-rate encoder.

GEPARD implementation mentioned in chapter 4, producing three separate compressed speech signals of different sizes. The architecture of the multirate GSM full-rate encoder is shown in Figure 6.10. The C code is listed in Appendix F. The number of input samples

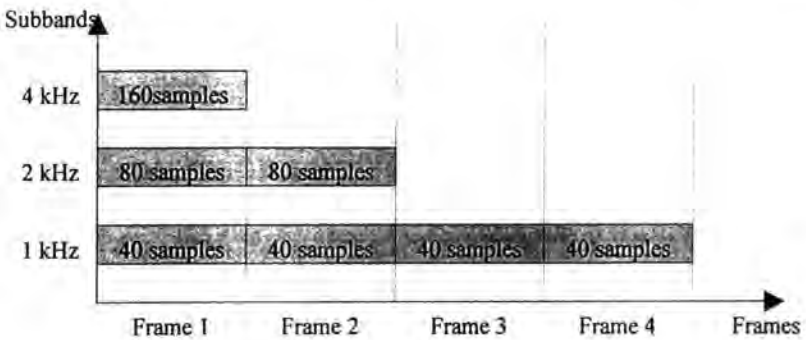


Figure 6.11 Synthesis of the subband coded speech signal.

were kept the same as conventional GSM full-rate codec. The frame rates is shown in Figure 6.11. They were then GSM decoded using the C decoder.

The three files were passed through the subband speech decoder. The synthesis method shown in Figure 6.12 which was the reverse of the analysis process. The signals in lowpass

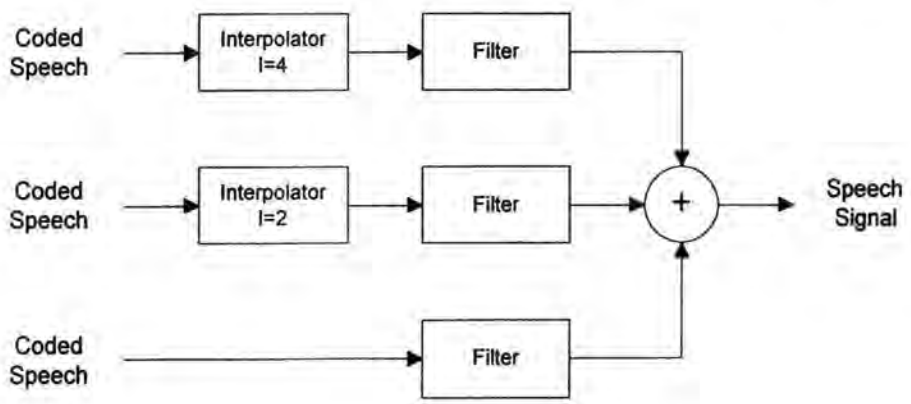


Figure 6.12 Different frame rates of the 3 subbands.

and bandpass frequency bands were interpolated, filtered and combined. The output signals were compared with the originals. This experiment was not done in real time.

6.5.2 Results

The original waveform of one of the files used for this experiment is shown in Figure 6.13. The FFT of this waveform was obtained using

```
S=fft(s,512);
w=(0:255)/256*(Fs/2);
plot(w,abs([S(1:256)]),'w');
hold;
xlabel('Frequency (Hz)');
ylabel('Mag. of Fourier transform');
```

The result is shown in Figure 6.14. After the three frequency bands were filtered and their frequency contents were compared with the original plot (see Figure 6.13). The three bands were then GSM encoded and decoded. The three decoded bands were compared with their originals, both in time and frequency domain. The output signals were synthesised by the subband decoder and the reconstructed signal was obtained. The results are shown in Figure 6.15-23.

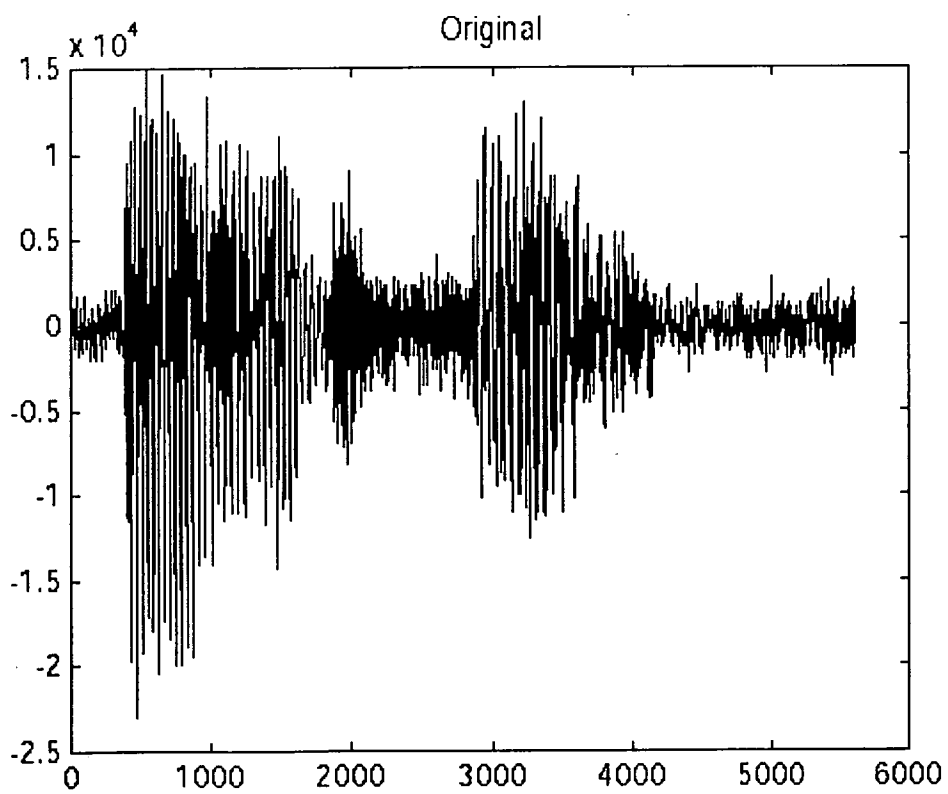


Figure 6.13 The waveform of "eat him".

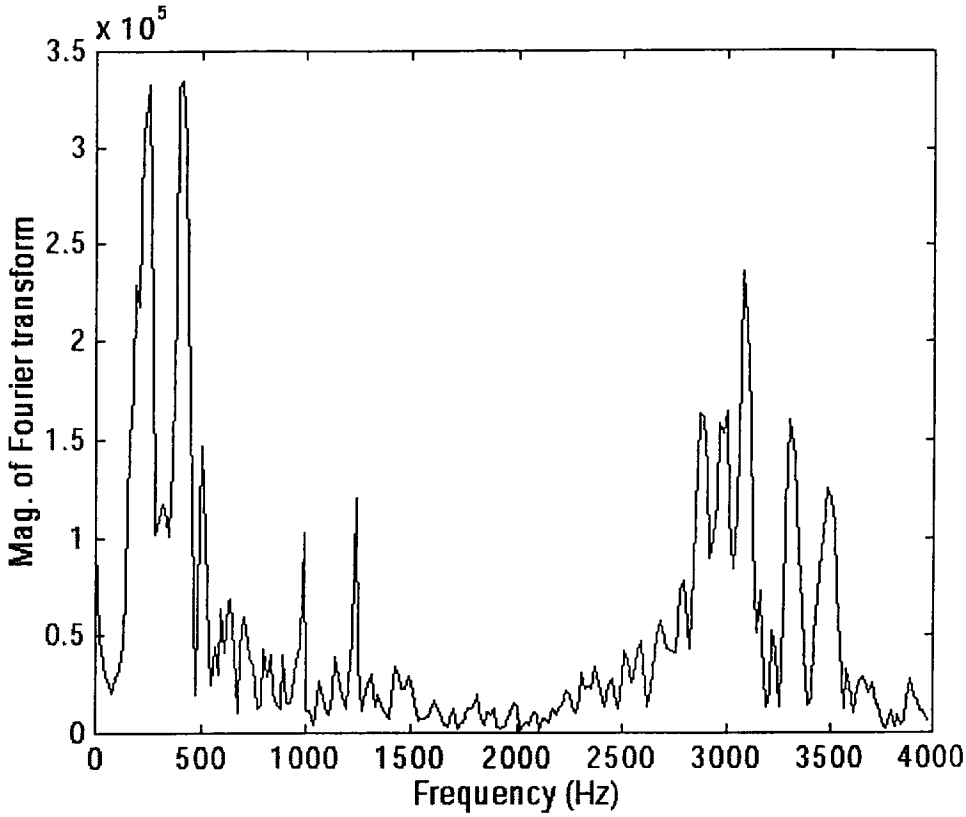


Figure 6.14 The FFT of the waveform of "eat him".

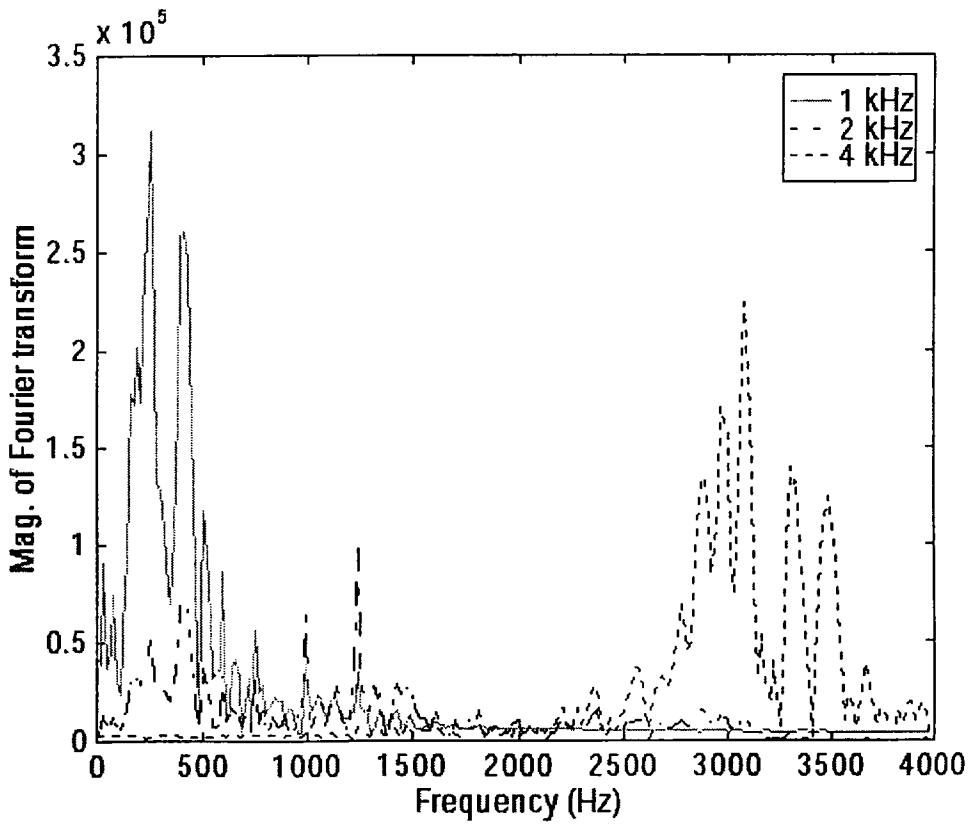


Figure 6.15 The FFT of the 1 kHz, 2 kHz and 4 kHz subbands.

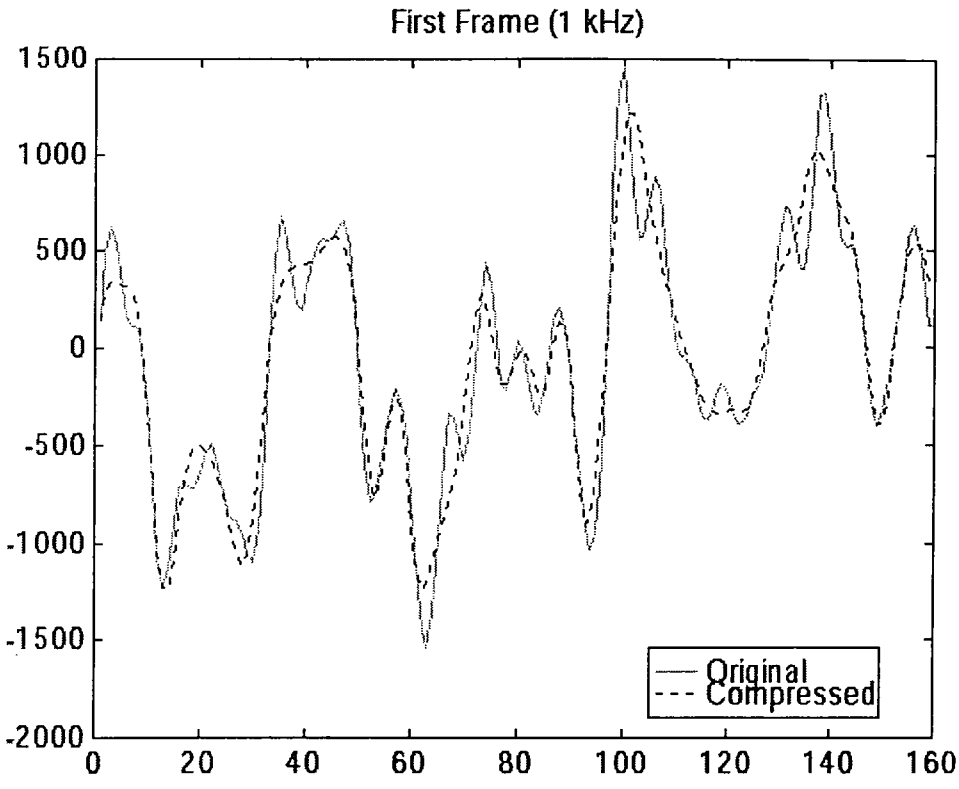


Figure 6.16 The original and compressed waveform of the 1 kHz subband.

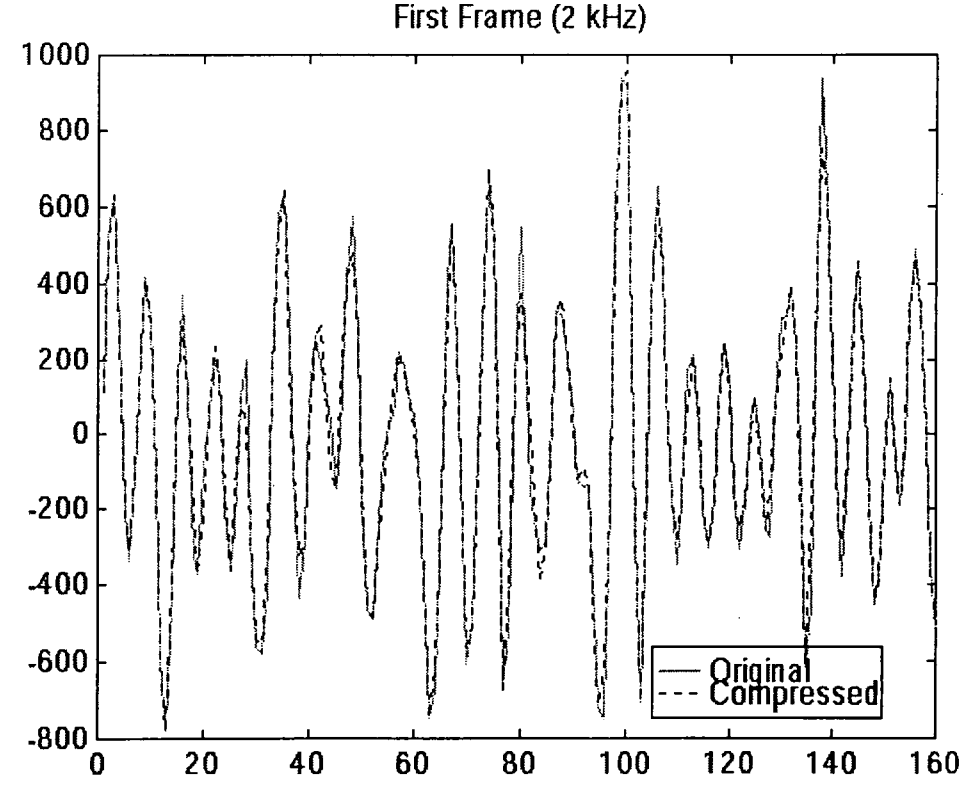


Figure 6.17 The original and compressed waveform of the 2 kHz subband.

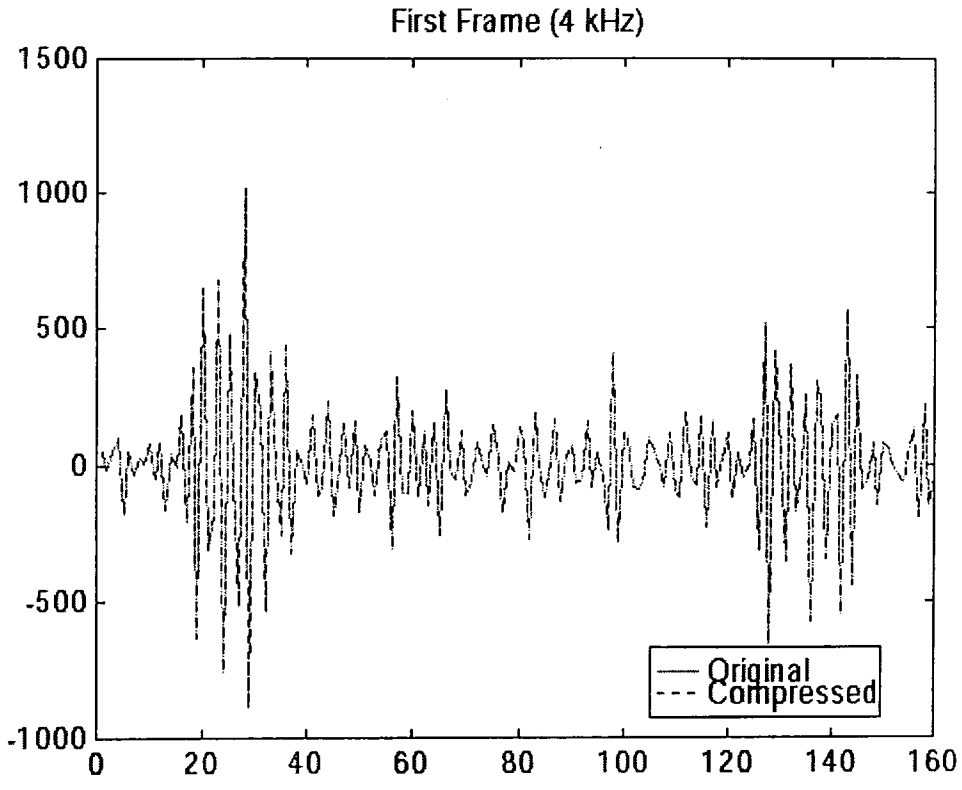


Figure 6.18 The original and compressed waveform of the 4 kHz subband.

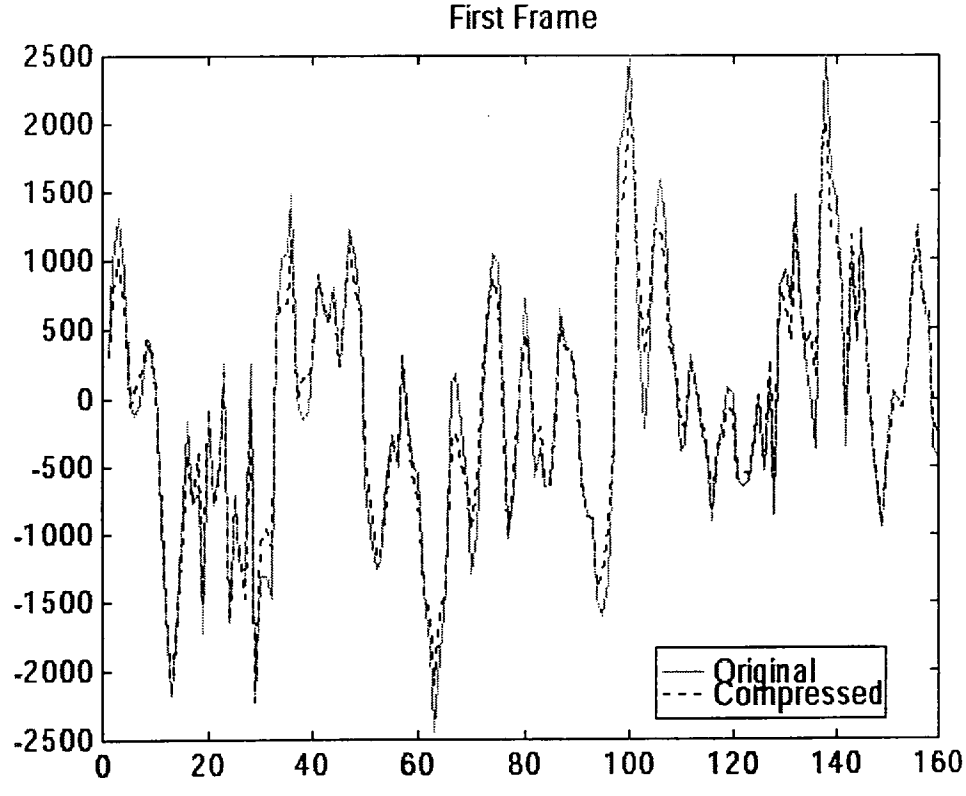


Figure 6.19 The original and compressed waveform.

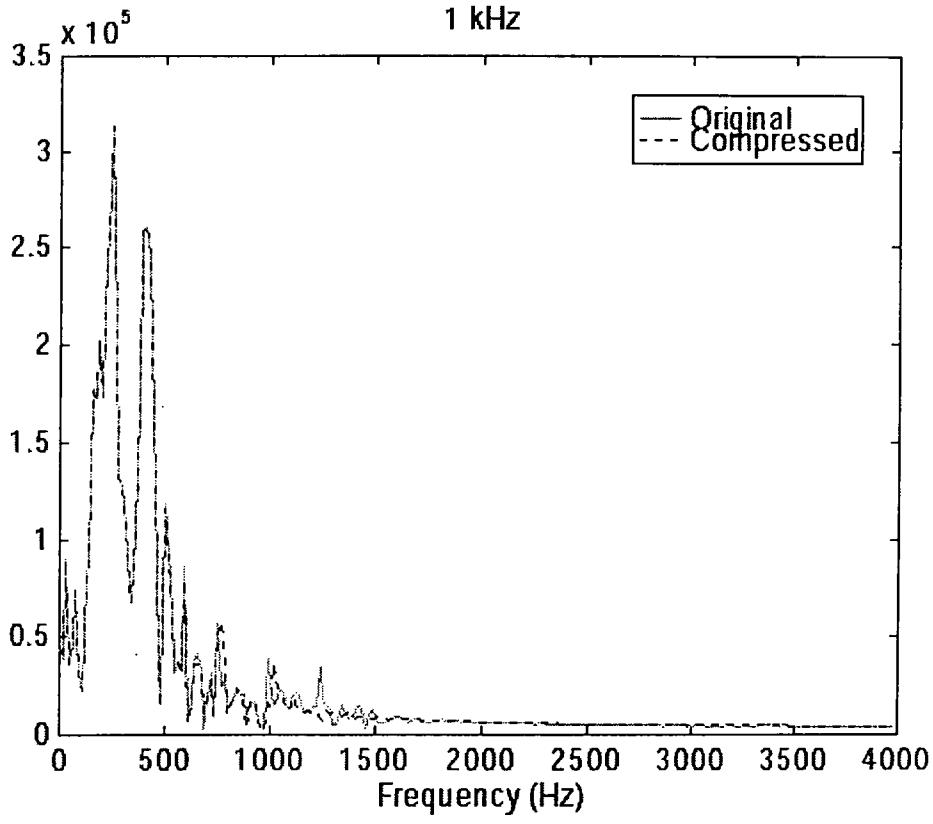


Figure 6.20 The FFT original and compressed waveform of the 1 kHz subband.

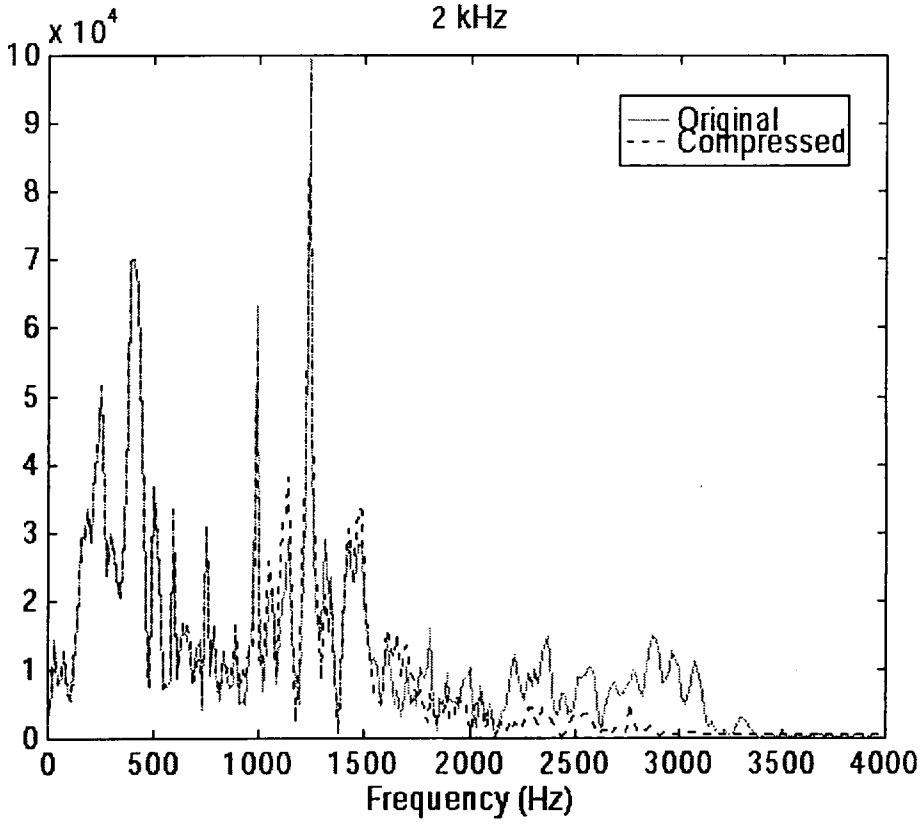


Figure 6.21 The FFT original and compressed waveform of the 2 kHz subband.

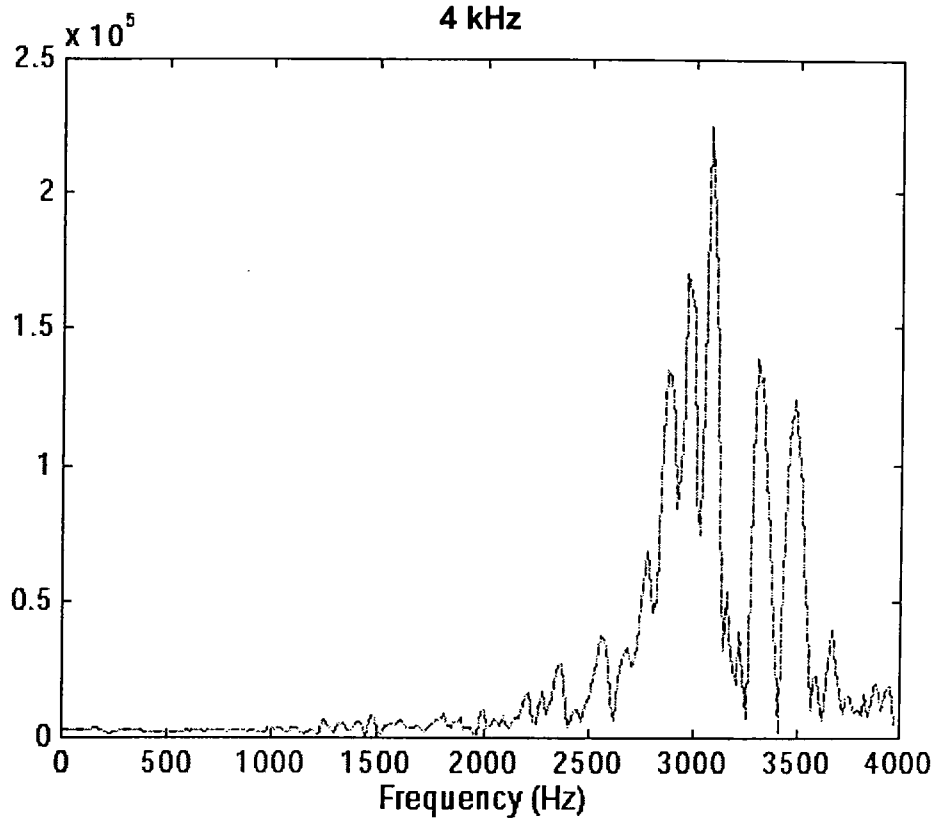


Figure 6.22 The FFT original and compressed waveform of the 4 kHz subband.

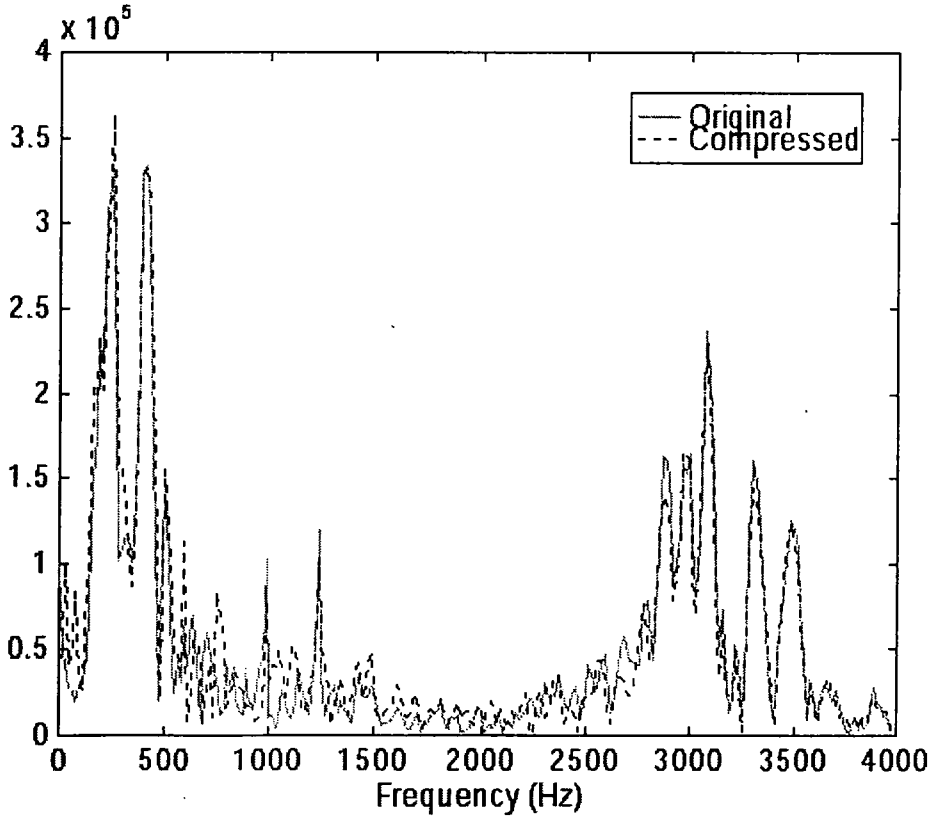


Figure 6.23 The FFT original and compressed waveform.

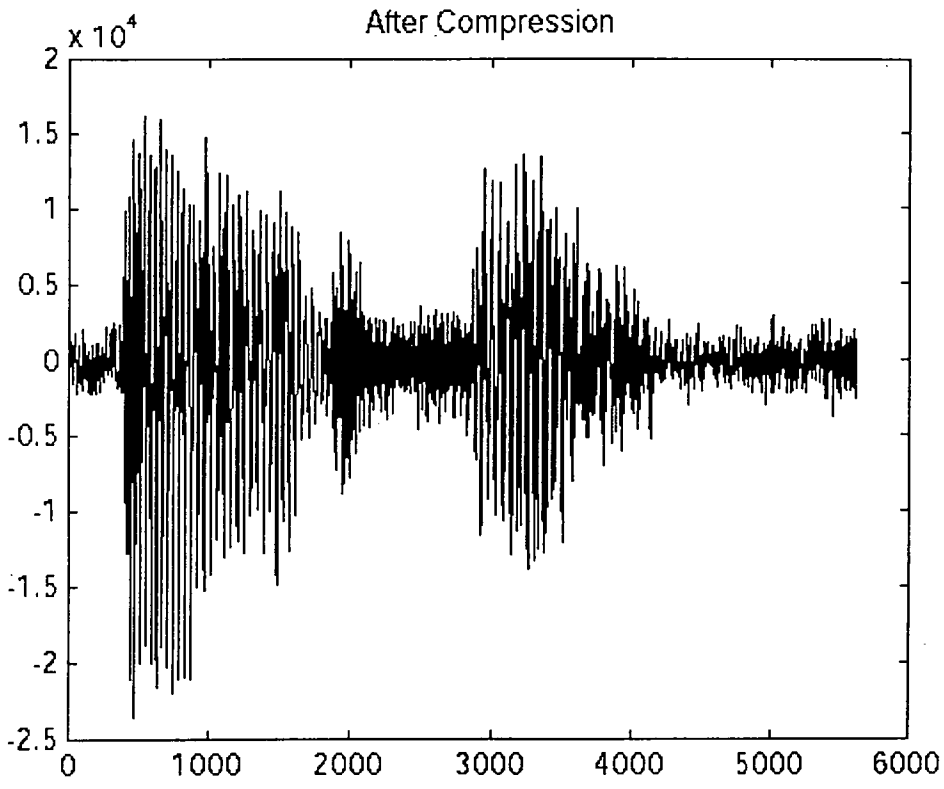


Figure 6.24 The reconstructed waveform "eat him".

The reconstructed signals were compared with the original ones. A plot of one of the signals can be found in Figure 6.24.

The modified speech files were played. The degradation of the signals was quite noticeable since both GSM full-rate speech compression and subband coding are lossy techniques. Even though the input data of the GSM encoder is a lot smaller after subband coding, the bit rate was not lower than GSM due to the constraints specified by ETSI and the number of bits produced per frame is restricted. Since GSM full-rate required 160 samples per frame, the frame rate of the subbands were different, ranging from 12.5 Hz to the usual 50 Hz. The signal could not be reconstructed by the subband decoder until all of the required data have been GSM decoded.

Figure 6.11 shows that for the 4 kHz subband, it operates at the same frame rate as GSM. However it has to wait for three more frames before all of its data is processed and ready to be read by the subband decoder. This introduces longer coding delay and therefore the algorithm will not be real-time. Other 4 kHz and 2 kHz frames can be processed at the same time while waiting for the 1 kHz frame but this also involves more processing power.

The 4 kHz frame produces the same number of bits as GSM full-rate codec. Therefore there will be an increase in bit rate from the output of the other two subbands. However, this time division multirate filtering technique is still a good way to use the spare capacity. In the next section, further enhancement of this codec and the possibility of a more efficient implementation are discussed.

6.5.3 Discussion

Multirate signal processing cannot be directly applied GSM full-rate speech compression as this requires more processing power, causing longer coding delay but does not appreciably improve the bit rate. The quality of the speech signal can be improved by designing better filters including using QMFs. However, this will not solved the other problems caused by other factors.

The GSM full-rate speech codec is very restricted and gives no allowances for any deviations. No modification of the algorithm can be made as it requires bit exactness. Since the output number of bits is specified, the bit rate cannot be reduced. Therefore, a direct application multirate filtering to GSM full-rate speech compression is not possible. Subband coding is more flexible and the number of quantisation bits can be altered.

The different properties of the three frames produced by the subband coder can be investigated. The complexity of the algorithm can be simplified as the capacity of the frames are reduced. The full-rate algorithm can be modified to concentrate on the different characteristics of each frame. For example, less reflection coefficients would be needed for the higher subbands since most of the speech energy is contained in the lower frequencies. They could also be sent at a lower resolution. More emphasis can be put on the RPE section of the algorithm as there could be more information about the unvoiced speech signal at this frequency. The output samples of the three GSM encoded frames can be sent at a lower bit rate as they are only the components of the original speech signal.

For better compression to be achieved, the GSM full-rate mathematical algorithm can be used instead of the standardised ETSI recommendation. Some changes including the number of quantisation bits has to be made before the application of multirate signal processing. Hence the bit rate of the three subbands would be different, with more bits assigned to the algorithm which best describes the property of speech at a particular frequency. In order to combine multirate filtering and GSM codec a new standard will be required.

The ETSI Special Mobile Group (SMG) has been studying into the feasibility of the Adaptive Multirate (AMR) speech concept for the past year. AMR was initially aimed at

enhanced full-rate (EFR) GSM codec. The EFR-GSM voice codec developed by Nokia in 1995 [MARS95] is a coding algorithm that facilitates the conversion of analogue speech into a digital data stream that can be modulated for transmission over the radio channel. The algorithm is fully compatible with a GSM 13 kbit per second speech channel. However, there are other candidates including UMTS and other ITU third generation systems which are being defined over the next two years [GAS98a] [GAS98b].

6.6 Summary

In telecommunication systems that transmit and receive different speech source, there is a requirement to process the various data at different frequencies that involves sample rate conversion. Multirate signal processing discussed in this chapter is an efficient technique for changing the sampling frequency of a signal digitally. Filter design and requirement for sample rate conversion have been explained. Subband coding and MPEG audio compression which use the multirate technique have been described. The possibility of applying multirate filtering to GSM full-rate speech compression has been investigated. The results showed that multirate signal processing cannot be directly applied GSM full-rate speech compression since this method requires more processing power, causing longer coding delay but did not appreciably improve the bit rate. However, this time division multirate filtering technique is still a good way to use the spare capacity.

Further enhancement of this codec can be made. For lower bit rate to be achieved, the standardised ETSI recommendation needs to be modified. The method used can be based on the exisitng GSM full-rate mathematical algorithm. The different properties of the three frames produced by the subband coder can be investigated. The complexity of the algorithm can be simplified as the capacity of the frames are reduced. Some changes including the number of quantisation bits has to be made before the application of multirate signal processing. Therefore in order to combine multirate filtering and GSM codec a new standard will be required. The ETSI Special Mobile Group (SMG) has been studying into the feasibility of the Adaptive Multirate (AMR) speech concept and the codec will be defined in the near future.

7 Conclusions and Further Work

Speech coding is the field concerned with obtaining compact digital representations of speech signals for the purpose of efficient transmission. They achieve this by taking advantage, to varying degrees, of redundancies in the speech signals, and the digital storage of speech signals.

The work presented in this thesis concerns the background theory of speech and the current speech coding schemes, a real time implementation on DSP and an application of multirate filtering to GSM full-rate speech compression.

The properties and perception of speech and the DSP mathematical algorithms used in speech coding including sampling theory, short-term spectral analysis, quantisation techniques, linear predictive coding and pitch prediction have been described. The objectives and requirements of speech compression which includes the quality and capacity, coding delay, complexity and cost have been discussed. The current speech coding schemes have been described in the following categories: the high quality waveform coders, the low bit rate vocoders and the hybrid coders that attempts to fill the gap between waveform coders and vocoders.

An overview of GSM full-rate speech compression have been presented. The GSM full-rate speech compression algorithm is a lossy technique which is based on a residually excited linear predictive coder (RELP) and this is further enhanced by using a long term predictor (LTP). This improves speech quality by removing the structure from the vowel sounds prior to coding the residual data. It compresses frames of 160 13-bit signed samples to 260-bit compressed frames.

One of the major objectives of this work was to use the GSM algorithm to investigate optimisation technique for an ASIC processor core GEPARD, which is produced by AMS. GEPARD is an embedded software programmable DSP core for telecommunication, consumer and industrial applications. The hardware requirement, implementation strategies, optimisation techniques, testing and debugging of code and the performance of the assembly code for this implementation have been discussed. The GEPARD code has been tested with all of the test sequences provided by ETSI and the results are bit-exact. The code has been fully optimised and the transcoding delay is 28 ms which is lower than the ETSI 30 ms requirement. Further optimisation can be made by adding new instructions with overflow control and saturation.

A brief review of GSM half-rate speech compression have been presented. ETSI specified the GSM half-rate codec with a bit rate of 5.6 kbps in 1995. The algorithm is based on Motorola's VSELP technology similar to IS-54 full-rate. It uses two 7-bit codebooks for unvoiced speech and one 9-bit codebook for voiced segments. A comparison with the full-rate algorithm is discussed.

The half-rate algorithm takes advantage of more efficient speech compression than full-rate techniques do, shrinking the bandwidth timeslice that each user requires. Although GSM full-rate codec operates at 13 kbps and half-rate at 5.6 kbps, they both offer near toll speech quality comparable or better than analogue cellular networks. The half-rate compression introduces new algorithms such as FLAT and AFLAT and requires more computationally intensive operations compared with the full-rate. Hence a four times more powerful processor will be needed for the half-rate codec. The cost of the implementation of half-rate codec will also be considerably higher than full-rate.

A description of multirate signal processing and its application on speech (SBC) and speech/audio (MPEG) has been given. Multirate signal processing is an efficient technique for changing the sampling frequency of a signal digitally. An investigation into the possibility of combining multirate filtering and GSM full-rate speech algorithm. The results showed that multirate signal processing cannot be directly applied GSM full-rate speech compression since this method requires more processing power, causing longer coding delay but did not appreciably improve the bit rate. However, this time division multirate filtering technique is still a good way to use the spare capacity.

Further enhancement of this codec can be made. For lower bit rate to be achieved, the standardised ETSI recommendation needs to be modified. The method used can be based on the existing GSM full-rate mathematical algorithm. Therefore in order to combine multirate filtering and GSM codec a new standard will be required.

Other new research in the field of low bit rate speech coding includes variable frame rate speech coding [CHU94], forward-backward waveform prediction [YAN95], parametric filtering [LI96], formant vocoders using mixtures of Gaussian [ZOL96] [ZOL97a] [ZOL97b], variable-rate CELP [MCCL97] and adaptive multirate codec[GAS98a] [GAS98b].

References

[AMS96] AMS, *GEPARD Simulator User Manual*, Release 1.5, AMS, November 1996.

[AMS97a] AMS, *GEPARD Instruction Set Summary*, Release 1.5, AMS, April 1997.

[AMS97b] AMS, *GEPARD Symbolic Assembler User Manual*, Release 1.5.3, AMS, April 1997.

[BLA96] Black A. W., Kondoz A. M. and Evans B. G., "Improved Pulsed Residual Excited Linear Prediction: A New Excitation for CELP Based Speech Coders", *Electronics Letters*, vol. 32, no. 6, pp. 520-521, March 1996.

[CAR86] Carlson A. B., *Communication Systems: An Introduction to Signals and Noise in Electrical Communication*, Third Edition, McGraw-Hill, Singapore, 1986.

[CHA96] Chan C. F. and Yu E. W. M., "Improving Pitch Estimation for Efficient Multiband Excitation Coding of Speech", *Electonics Letters*, vol. 32, no. 10, pp. 870-872, May 1996.

[CHU94] Chung C.J. and Chen S. H., "Variable Frame Rate Speech Coding Using Optimal Interpolation", *IEEE Trans. on Communications*, vol. 42, no. 6, pp. 2215-2218, June 1994.

[CRO93] Crochiere R. E. and Rabiner L. R., *Multirate Digital Signal Processing*, Prentice-Hall, New Jersey, USA, 1993.

[CUC96] Cucchi S., Fratti M. and Ronchi M., "On Improving Performance of Analysis by Synthesis Speech Coders", *IEEE Trans. on Speech and Audio Processing*, vol. 4, no. 3, May 1996.

[DEG94] Degener J., "Digital Speech Compression", on the World Wide Web at http://www.ddj.com/ddj/1994/1994_12/degener.htm, December 1994.

[ETS94] ETSI, *European Digital Cellular Telecommunications System (Phase 2); Full Rate Speech Transcoding (GSM 06.10)*, ETSI, 1994.

[ETS95] ETSI, *European Digital Cellular Telecommunications System; Half Rate Speech Part 2: Half Rate Speech Transcoding (GSM 06.20)*, ETSI, 1995.

[FUR96] Furber S., *ARM System Architecture*, Addison-Wesley, Essex, UK, 1996.

[GAB47] Gabor D. "Acoustical Quanta and the Theory of Hearing", *Nature*, vol. 159, no. 4044, pp. 591-592, May 1947.

[GAS98a] Gaskell P., "Adaptive Multi-Rate Speech Codec - Combining Wireline Quality with Half-Rate Capacity", on the World Wide Web at http://www.etsi.org/smg/smg11/gaskell.htm, June 1998.

[GAS98b] Gaskell P., "Special Mobile Group 11 (SMG 11): Speech Aspects", on the World Wide Web at http://www.etsi.org/smg/smg11/smg11.htm, June 1998.

[HIL97] Hillebrand F., "GSM, the Advanced and Proven Basis for Success in Mobile Markets", on the World Wide Web at http://www.etsi.org/smg/gsm_bas.htm, October 1997.

[IFE93] Ifeacher E. C. and Jervis B. W., *Digital Signal Processing : A Practical Approach*, Addison-Wesley, Wokingham, UK, 1993.

[KLE97] Klein P. K., "Audio/Voice Compression Algorithm Criteria Applied to Telecommunications", on the World Wide Web at http://www.dspse.com/PAPERS/VOCODER/voc_sel.htm, 1997.

[KON95] Kondoz A., *Digital Speech for Low Bit Rate Communications*, Wiley, New York, USA, 1995.

[KUN85] Kung S.Y., Whitehouse H. J. and Kailath T., *VLSI and Modern Signal Processing*, Prentice-Hall, New Jersey, USA, 1985.

[LI96] Li T. H. and Gibson J. D., "Speech Analysis and Segmentation by Parametric Filtering", *IEEE Trans. on Speech and Audio Processing*, vol. 4, no. 2, pp. 203-213, May 1996.

[MAK75] Makhoul J., "Linear Prediction: A Tutorial Review", *Proceedings of the IEEE*, vol. 63, no. 4, pp. 561-580, April 1975.

[MARS95] Marston D. F., Wong D. T. K. and Wong W. T. K., "Speech-and-Channel Coding Scheme for Enhanced Full-Rate GSM Standard", *Electronics Letters*, vol.31, no.24, November 1995.

[MARV93] Marvin C. and Ewers G., *A Simple Approach to Digital Signal Processing*, Texas Instruments, 1993.

[MCCL97] McClellan S. and Gibson J. D., "Variable-Rate CELP Based on Subband Flatness", *IEEE Trans. on Speech and Audio Processing*, vol. 3, no. 2, pp. 120-130, March 1997.

[MCCR95] McCree A. V. and Barnwell T. P., "A Mixed Excitation LPC Vocoder Model for Low Bit Rate Speech Coding", *IEEE Trans. on Speech and Audio Processing*, vol. 3, no. 4, pp. 242-249, July 1995.

[MCE95] McElroy C., "*Speech Production and Perception*", on the World Wide Web at http://wwwdsp.ucd.ie/speech/tutorial/speech_coding/speech.html, November, 1995.

[PAI96] Painter E. and Spanias A., "A MATLAB Software Tool for the Introduction of Speech Coding Fundamentals in a DSP Course", presented at the *IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP-96)*, Atlanta, and in *IEEE Trans. Engineering Education*, available on the World Wide Web at http://www.eas.asu.edu/~spanias/papers/icassp96.ps, 1996.

[PAN93] Pan D., "Digital Audio Compression", available on the World Wide Web at ftp://ftp.digital.com/pub/Digital/info/DTJ/v5n2/Digital_Audio_Compression_01oct93DTJA03P8.ps, 1993.

[PAN95] Pan D., "A Tutorial on MPEG/Audio Compression", published in *IEEE Multimedia Journal*, available on the World Wide Web at http://www.bok.net/~pan/, Summer 1995.

[PRO92] Proakis J. G. and Manolakis D. G., *Digital Signal Processing: Principles, Algorithms and Applications*, 2nd Edition, Macmillan, New York, USA, 1992.

[RAP96] Rappaport T. S., *Wireless Communications: Principles and Practice*, Prentice-Hall, New Jersey, USA, 1996.

[ROB97] Robinson T., "Speech Analysis", available on the World Wide Web at http://svr-www.eng.cam.ac.uk/~ajr/SpeechAnalysis/ajrspeech.ps, 1997.

[SCH92] Schroeter J., *Surviving the ASIC Experience*, Prentice-Hall, New Jersey, USA, 1992.

[SCO95] Scourias J., "Overview of the Global System for Mobile Communications", available on the World Wide Web at http://conga.waterloo.ca/~jscourias/GSM/gsmreport.ps, 1995.

[SHA48] Shannon C. D., "A Mathematical Theory of Communications", *Bell Systems Technical Journal*, vol. 27, pp. 379 - 423 and 623 - 656, 1948.

[SHO87] Short K. L., *Microprocessors and Programmed Logic*, Prentice-Hall, New Jersey, USA, 1987.

[SPA94] Spanias A. S., "Speech Coding: A Tutorial Review", *Proc. of the IEEE*, vol.82, no.10, pp. 1541-1579, October 1994.

[SPO92] Sporer Th., Brandenburg Kh. and Edler B., "The Use of Multirate Filter Banks for Coding of High Quality Digital Audio", *6th European Signal Processing Conference (EUSIPCO)*, Amsterdam, vol. 1, pp 211-214, June 1992.

[STR89] Strum R. D. and Kirk D. E., *First Principles of Discrete Systems and Digital Signal Processing*, Addison-Wesley, Massachusetts, USA, 1989.

[TEM96] Tempelaars S., *Signal Processing, Speech and Music*, Swets, Lisse, 1996.

[TIP91] Tipler P. A., *Physics: For Scientists and Engineers*, 3rd Edition, Entended Version, Worth, New York, USA, 1991.

[USA98] Usai P., "Quality Assessment of New Speech Codecs Deployed in Mobile Networks", on the World Wide Web at http://www.etsi.org/smg/smg11/usai.htm, 1998.

[VET92] Vettering W. T., Teukolsky S. A., Press W. H. and Flannery B. P., *Numerical Recipes Example Book (C)*, 2nd Edition, Cambridge University Press, Cambridge, UK, 1992.

[WIS96] Wiseman J., " A Complete DSP Design Example Using FIR Filters", Published in *QEX* magazine, on the World Wide Web at http://members.aol.com/johnw39038/qex_fir.htm, July 1996.

[WON96] Wong W. T. K., Mack R. M. and Cheetham B. M. G. and Sun X. Q., "Low Rate Speech Coding for Telecommunications", *BT Technology Journal*, vol. 14, no. 1, pp. 28-44, January 1996.

[XIA96] Xia X. G. and Suter B. W., "Multirate Filter Banks with Block Sampling", *IEEE Trans. on Signal Processing*, vol. 44, no. 3, pp. 484-496, March 1996.

[YAN95] Yang G. and Boite R., "Voiced Speech Coding at Very Low Bit Rate Based on Forward-Backward Waveform Prediction", *IEEE Trans. on Speech and Audio Processing*, vol. 3, no. 1, pp. 40-47, January 1995.

[ZOL96] Zolfaghari P. and Robinson T., "A Formant Vocoder Based on Mixtures of Gaussians", published in *Proceedings ICASSP'97*, available on the World Wide Web at http://svr-www.eng.cam.ac.uk/~psz1000/papers/icslp96.ps, 1996.

[ZOL97a] Zolfaghari P. and Robinson T., "A Segmental Formant Vocoder Based on Linearly Varying Mixture of Gaussians", published in *Proc. EUROSPEECH'97*, available on the World Wide Web at http://svr-www.eng.cam.ac.uk/~psz1000/papers/euro97.ps, 1997.

[ZOL97b] Zolfaghari P. and Robinson T., "Formant Analysis Using Mixtures of Gaussians", published in *ICSLP'96*, available on the World Wide Web at http://svr-www.eng.cam.ac.uk/~psz1000/papers/icassp97.ps, 1997.

Appendix A Full Rate Speech Transcoding (GSM 06.10)

A.1 Introduction

The GSM full-rate speech compression algorithm is a lossy technique which is based on a residually exited linear predictive coder (RELP) and this is further enhanced by using a long term predictor (LTP). It compresses frames of 160 13-bit signed samples to 260-bit compressed frames. A full description of GSM full-rate speech transcoding can be found in [ETS94].

A.2 Encoder

The input speech frame consisted of uniform 13-bit PCM signed samples converted from 8-bit A-law companded format. The encoder output parameters are shown in Table D.1. A block diagram of the Encoder is shown in Figure B.1.

A.2.1 Preprocessing Section

After A-law to linear conversion (or directly from the A to D converter) the following input sample (2's complement format) is obtained

S.v.v.v.v.v.v.v.v.v.v.v.v.x.x.x

where **S** is the signed bit, **v** a valid bit and **x** a "don't care" bit. The input samples are downscaled by a factor of two and a notch filter is applied in order to remove the offset of the signal s_0 to produce the offset-free signal s_{of}.

$$s_{of}[k] = s_0[k] - s[k-1] + \alpha s_{of}[k-1] \tag{A.1}$$

where $\alpha = 32735 \times 2^{-15}$. The signal s_{of} is then applied to a first order FIR preemphasis filter leading to the input signal s of the analysis section,

$$s[k] = s_{of}[k] - \beta s_{of}[k-1] \tag{A.2}$$

where $\beta = 28180 \times 2^{-15}$.

A.2.2 LPC Section Analysis

The speech signal $s[k]$ is divided into non-overlapping frames having a period of 20 ms (160 samples). A new LPC analysis of order $p = 8$ is performed for each frame. The first nine values of the autocorrelation function are calculated by

$$ACF[k] = \sum_{i=k}^{159} s[i]s[i-k] \tag{A.3}$$

where $0 \le k \le 8$. The reflection coefficients are calculated as shown in Figure A.1 using the Schur Recursion algorithm. The reflection coefficients $r(i)$, where $1 \le i \le 8$, calculated by the Schur algorithm, are in the range $-1 \le r[i] \le +1$. Due to the favourable quantisation characteristics, the reflection coefficients are converted to log area ratios which are defined in (2.13).

Since it is the companding characteristic of this transformation that is of importance, the following segmented approximation is used.

$$\begin{aligned} LAR[i] &= r[i] && ; \quad |r[i]| < 0.675 \\ &= sign\{r[i]\}\{2|r[i]| - 0.675\}; && 0.675 \le |r[i]| < 0.950 \\ &= sign\{r[i]\}\{8|r[i]| - 0.375\}; && 0.975 \le |r[i]| \le 1.000 \end{aligned} \tag{A.4}$$

with the result that instead of having to divide and obtain the logarithm of particular values, it is merely necessary to multiply, add and compare these values. The following equation is used for the inverse transformation.

$$\begin{aligned} r[i] &= LAR'[i] && ; \quad |r[i]| < 0.675 \\ &= sign\{LAR'[i]\}\{0.500|LAR'[i]| + 0.337500\}; && 0.675 \le |r[i]| < 1.225 \\ &= sign\{LAR'[i]\}\{0.125|LAR'[i]| + 0.796875\}; && 0.975 \le |r[i]| \le 1.625 \end{aligned} \tag{A.5}$$

The log area ratios $LAR[i]$ had different dynamic ranges and different asymmetric distribution densities. For this reason, the transformed coefficients $LAR[i]$ is limited and quantised differently according to (A.6), with $LAR_c[i]$ denoting the quantised and integer coded version of $LAR[i]$.

$$LAR_c[i] = \text{Nint}\{A[i]LAR[i] + B[i]\} \tag{A.6a}$$

with

$$\text{Nint}\{z\} = \text{int}\{z + 0.5sign\{z\}\} \tag{A.6b}$$

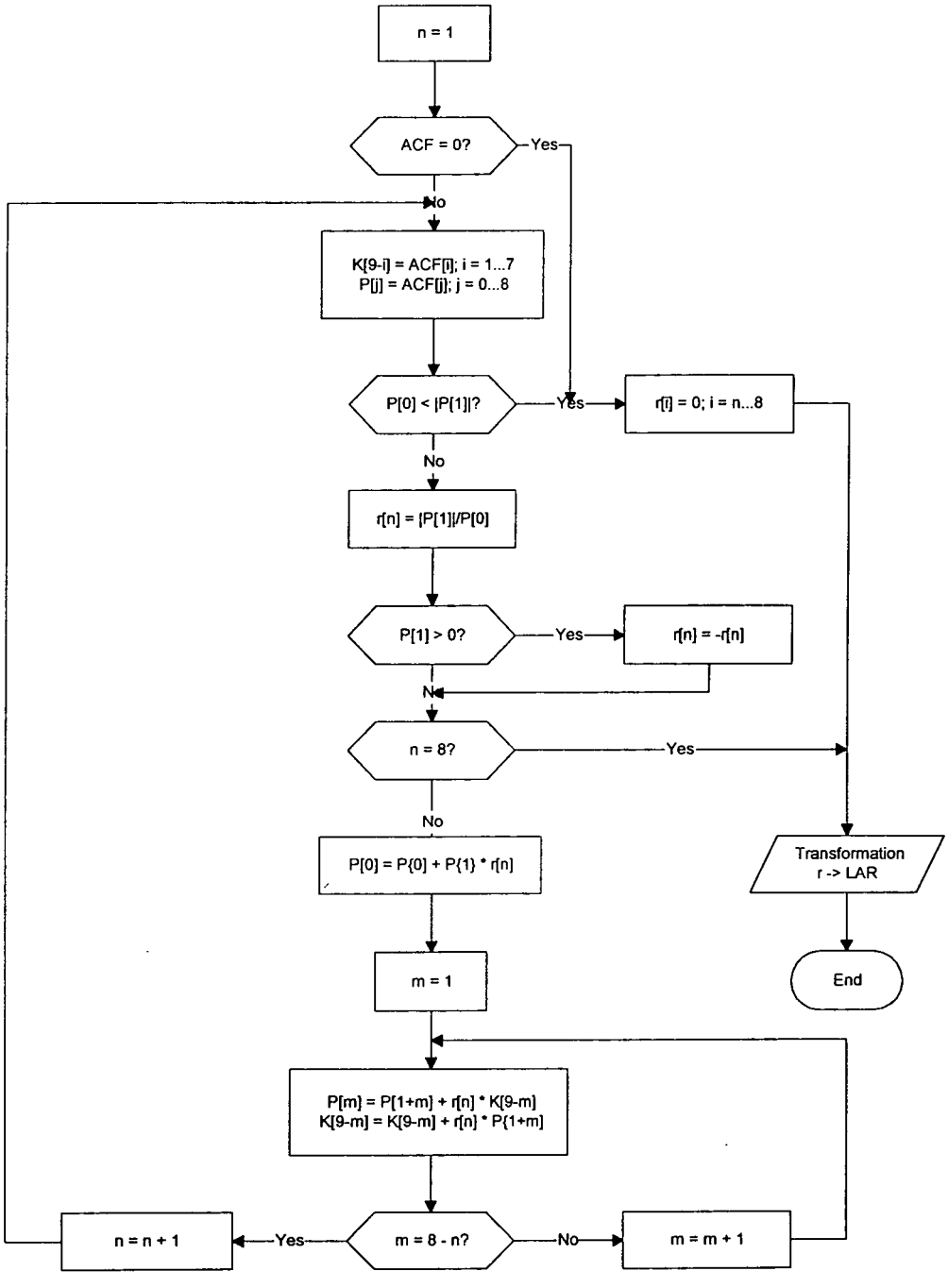


Figure A.1 LPC analysis using Schur recursion.

Function Nint defined the rounding to the nearest integer value with the coefficients $A[i]$, $B[i]$, and the different extreme values of LAR_c $[i]$ for each coefficient $LAR[i]$ given in Table C.1.

A.2.3 Short Term Analysis Filtering Section

The current frame of the speech signal s is retained in memory until calculation of the LPC parameters $LAR[i]$ is completed. The frame is then read out and fed to the short term analysis filter of order $p=8$. However, prior to the analysis filtering operation, the filter coefficients are decoded and preprocessed by interpolation. In this block the quantised and coded LAR_c $[i]$ are decoded according to (A.7).

$$LAR''[i]=(LAR_c[i]-B[i])/A[i] \tag{A.7}$$

To avoid spurious transients which might occur if the filter coefficients are changed abruptly, two subsequently sets of log area ratios are interpolated linearly. Within each

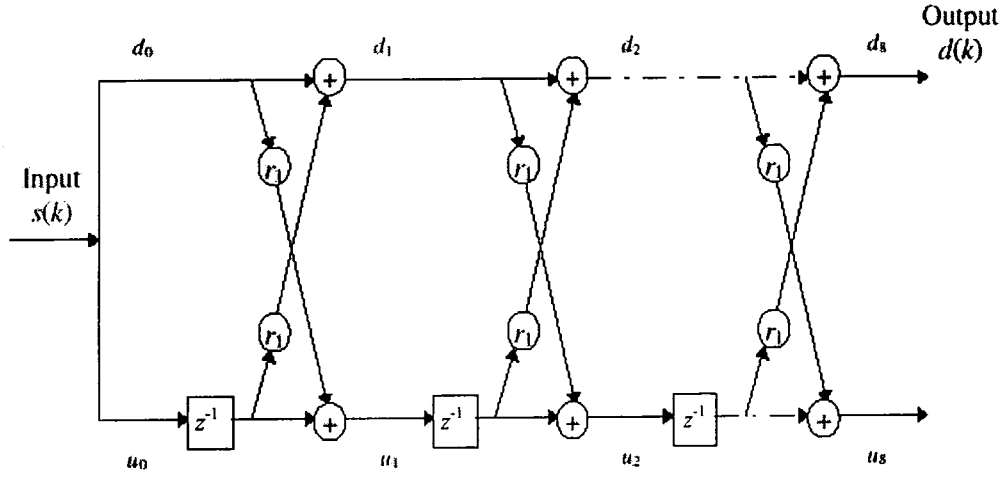


Figure A.2 Short term analysis filter.

frame of 160 analysed speech samples the short term analysis filter and the short term synthesis filter operate with four different sets of coefficients derived according to Table C.2.

The reflection coefficients are finally determined using the inverse transformation according to (A.5). The short term analysis filter is implemented according to the lattice structure shown in Figure A.2.

$$d_0[k]=s[k] \tag{A.8a}$$

$$u_0[k]=s[k] \tag{A.8b}$$

$$d_i[k]=d_{i-1}[k]+r_i'u_{i-1}[k-1] \tag{A.8c}$$

$$u_i[k]=u_{i-1}[k-1]+r_i'd_{i-1}[k] \tag{A.8d}$$

$$d[k]=d_8[k] \tag{A.8e}$$

A.2.4 Long Term Prediction Analysis Section

Each input frame of the short term residual signal contained 160 samples, corresponding to 20 ms. The long term correlation is evaluated four times per frame, for each 5 ms subsegment. For each of the four subsegments a long term correlation lag N_j and an associated gain factor b_j, where $0 \le j \le 3$, are determined. The crosscorrelation $R_j[\lambda]$ of the current subsegment of short term residual signal $d[k_j + i]$, $0 \le i \le 39$, and the previous samples of the reconstructed short term residual signal is evaluated signal $d'[k_j + i]$, $-120 \le i \le -1$, are evaluated.

$$R_j[\lambda] = \sum_{i=0}^{39} d[k_j + i] d'[k_j + i - \lambda] \tag{A.9}$$

where $0 \le j \le 3$, $k_j = k_0 + 40_j$, $0 \le \lambda \le 120$. The crosscorrelation is evaluated for lags greater than or equal to 40 and less than or equal to 120, i.e. corresponding to samples outside the current subsegment and not delayed by more than two subsegments.

The position n_j of the peak of the crosscorrelation function within this interval is then found

$$R_j[N_j] = \max\{R_j[\lambda]\} \tag{A.10}$$

The gain factor b_j is evaluated according to

$$b_j = R_j[N_j] / S_j[N_j] \tag{A.11a}$$

with

$$S_j[N_j] = \sum_{i=0}^{39} d'^2[k_j + i - N_j] \tag{A.11b}$$

The last 120 samples of the reconstructed short term residual signal $d'[k_j + i]$ is retained until the next subsegment.

The long term correlation lags N_j had values in the range between 40 and 120, and is coded using 7 bits with $N_{cj} = N_j$. At the receiving end, assuming an error free transmission, the decoding of these values would restore the actual lags N_j'.

The long term prediction gains b_j are encoded with 2 bits each, according to the following algorithm,

$$\begin{aligned} &\text{if} && b_j \le DBL[i] && \text{then } b_{cj}=0; \quad i=0 \\ &\text{if} \quad DBL[i-1] < b_j \le DBL[i] && && \text{then } b_{cj}=0; \quad i=1,2 \\ &\text{if} \quad DBL[i-1] < b_j && && \text{then } b_{cj}=0; \quad i=3 \end{aligned} \tag{A.12}$$

where $DBL[i]$, $0 \le i \le 2$ denoted the decision levels of the quantiser, and b_{cj} represents the coded gain value. Decision levels and quantising levels are given is Table C.3. The decoding rule is implemented according to

$$b'_j = QLB\lfloor b_{cj} \rfloor \tag{A.13}$$

where $QLB[i]$ denoted the quantising levels, and b_j' represented the decoded gain value.

The short term residual signal $d\lfloor k_0 + k \rfloor$, $0 \le k \le 159$, is processed by subsegments of 40 samples. From each of the four subsegments of short term residual samples, denoted here $d\lfloor k_j + k \rfloor$, and estimate $d''\lfloor k_j + k \rfloor$ of the signal is subtracted to give the long term residual signal $e\lfloor k_j + k \rfloor$.

$$e\lfloor k_j + k \rfloor = d\lfloor k_j + k \rfloor - d''\lfloor k_j + k \rfloor \tag{A.14}$$

where $0 \le j \le 3$, $0 \le k \le 39$, $k_j = k_0 + 40j$. Prior to this subtraction, the estimated samples $d''\lfloor k_j + k \rfloor$ are computed from the previously reconstructed short term residual samples d', adjusted to the current sub-segment LTP lag Nj' and weighted with the subsegment LTP gain b_j'.

$$d''\lfloor k_j + k \rfloor = b'_j d'\lfloor k_j + k - N' \rfloor \tag{A.15}$$

where $0 \le j \le 3$, $0 \le k \le 39$, $k_j = k_0 + 40j$. The reconstructed long term residual signal $e'\lfloor k_j + k \rfloor$ is processed by subsegments of 40 samples. To each subsegments, $e'\lfloor k_j + k \rfloor$, the estimated $d''\lfloor k_j + i \rfloor$ of the signal is added to give the reconstructed short term residual signal $d'\lfloor k_j + k \rfloor$.

$$d'\lfloor k_j + k \rfloor = e'\lfloor k_j + k \rfloor + d''\lfloor k_j + k \rfloor \tag{A.16}$$

where $0 \le j \le 3$, $0 \le k \le 39$, $k_j = k_0 + 40j$.

A.2.5 RPE Encoding Section

A FIR block filter algorithm is applied to each subsegment by convolving 40 samples $e[k]$ with the impulse response $H[i]$, with $0 \le i \le 10$, see Table C.4. The conventional

convolution of a sequence having 40 samples with an 11-tap impulse response would produce 50 samples. In contrast to this, the block filter algorithm produces the 40 notational convenience the block filtered version of each subsegment is denoted by $x[k]$

$$x[k] = \sum_{i=0}^{10} H[i]e[k+5-i] \tag{A.17}$$

with $e[k+5-i]=0$ for $[k+5-i]<0$ and $[k+5-i]>39$. The filtered signal x is then downsampled by a ratio of 3 resulting in 3 interleaved sequences of lengths 14, 13 and 13, which are split up again into 4 subsequences x_m of length 13.

$$x_m[i] = x[k_j + m + 3i] \tag{A.18}$$

with $0 \le i \le 12$ and $0 \le m \le 3$. m represented the position of the decimation grid. According to the explicit solution of the RPE mean squared error criterion, the optimum candidate subsequence x_M is selected which is the one with maximum energy.

$$E_M = \max_m \sum_{i=0}^{12} x_m^2[i] \tag{A.19}$$

The optimum grid position M is coded as M_c with 2 bits. The selected subsequence $x_M[i]$ is quantised, applying APCM (Adaptive Pulse Code Modulation). For each RPE sequence consisting of a set of 13 samples $x_M[i]$, the maximum x_{max} of the absolute values $|x_{max}[i]|$ is selected and quantised logarithmically with 6 bits as x_{maxc}.

For the normalisation, the 13 samples are divided by the decoded version x'_{max} of the block maximum. Finally the normalised samples

$$x'[i] = x_M[i] / x'_{max} \tag{A.20}$$

are quantised uniformly with 3 bits to $x_{Mc}[i]$ as given in Table C.5. The $x_{Mc}[i]$ are decoded to $x_M'[i]$ and denormalised using the decided value of x'_{maxc} leading to the decoded subsequence $x'_M[i]$. The quantised subsequence is upsampled by a ratio of 3 by inserting zero values according to the grid position given with *Mc*.

A.3 The Decoder

Most of the subblocks used in the encoder are also used here. Only the short tern synthesis filter and the deemphasis filter are added in the decoder as new subblocks. A block diagram of the decoder is shown in Figure B.2.

A.3.1 RPE Decoding Section

The input signal of the long term synthesis filter (reconstruction of the long term residual signal) is formed by decoding and denormalising the RPE samples and by placing them in the correct time position. At this stage, the sampling frequency is increased by a factor of 3 by inserting the appropriate number of intermediate zero-valued samples.

A.3.2 Long Term Prediction Synthesis Section

The reconstructed long term residual signal e_r' is applied to the long term synthesised filter which produced the reconstructed short term residual signal d_r' for the short term synthesiser.

A.3.3 Short Term Synthesis Filtering Section

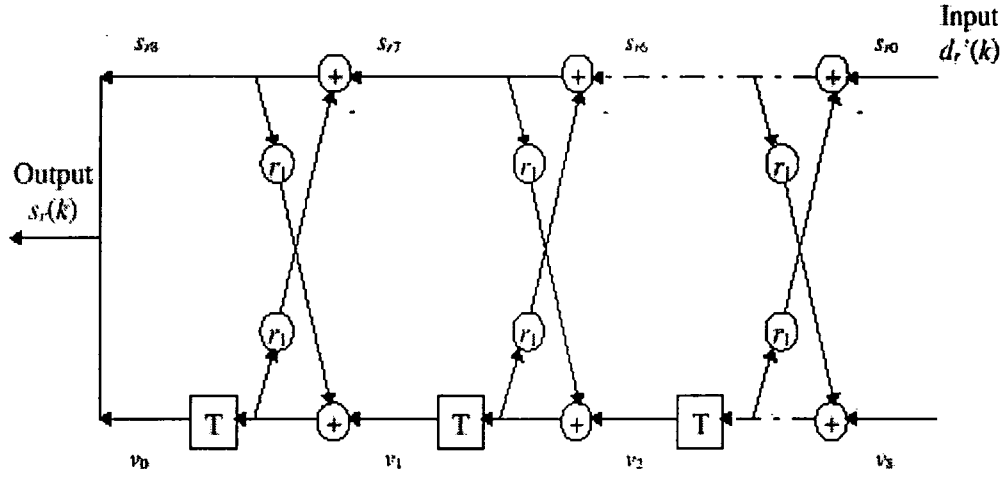


Figure A.3 Short term synthesis filter.

The coefficients of the short tern synthesis filter are reconstructed applying the identical procedure to that in the encoder. The short tern synthesis filter implemented according to the lattice structure shown in Figure A.3.

$$s_{r[0]}[k] = d_r'[k] \tag{A.21a}$$

$$s_{r[i]}[k] = s_{r[i-1]}[k] + r_r' v_{8-i}[k-1] \tag{A.21b}$$

$$v_{9-i}[k] = u_{9-i}[k-1] + r_i' s_{r[i]}[k] \tag{A.21c}$$

$$s_r'[k] = s_{r[8]}[k] \tag{A.21d}$$

$$v_0[k] = s_{r[8]}[k] \tag{A.21e}$$

A.3.4 Postprocessing Section

The output of the synthesis filter $s_r[k]$ is fed into the IIR deemphasis filter leading to the output signal s_{ro}.

$$s_{ro}[k] = s_r[k] - \beta s_{ro}[k-1] \tag{A.22}$$

Appendix B Block Diagrams of GSM Full-Rate Speech Compression

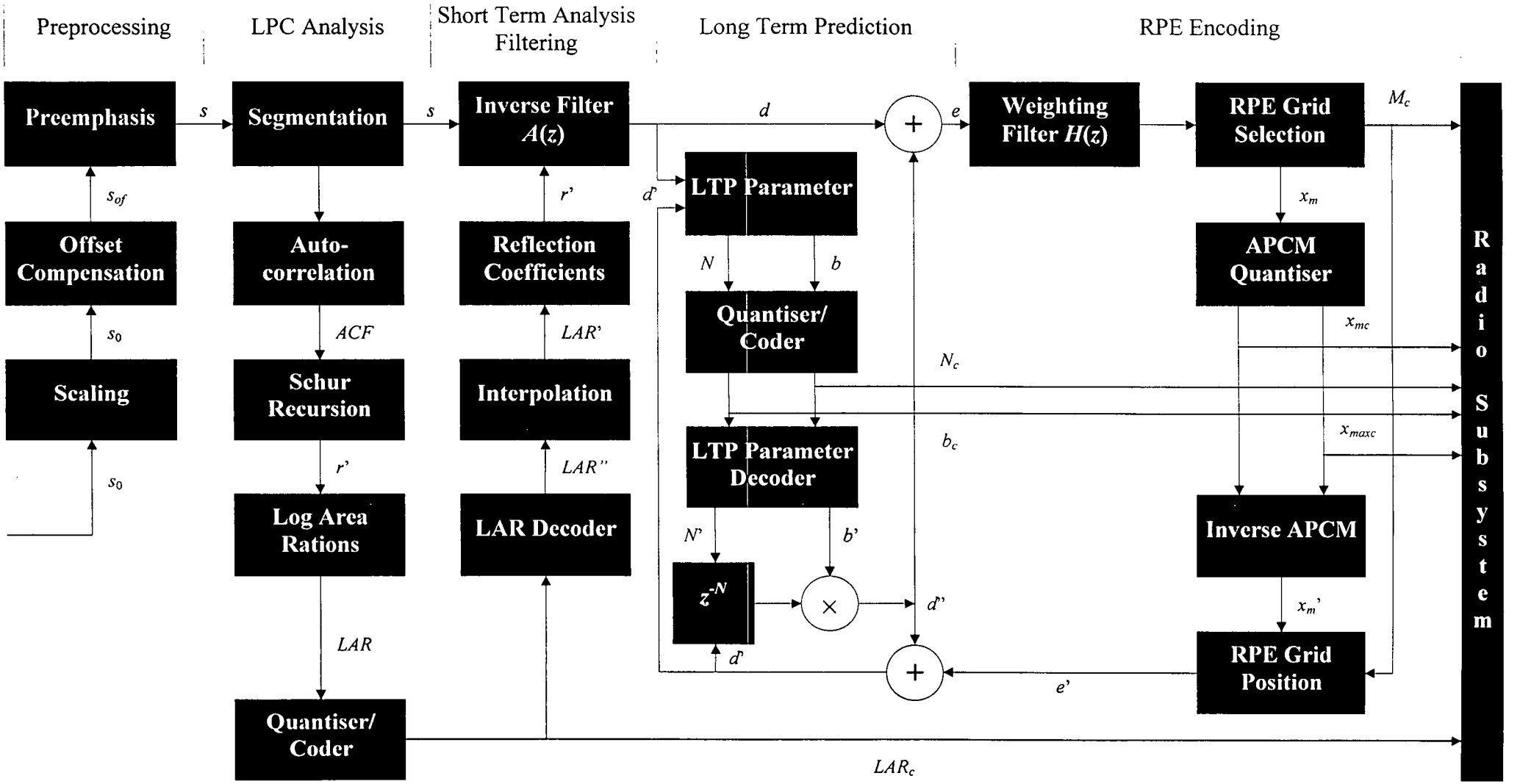


Figure B.1 Block Diagram of the GSM full-rate encoder.

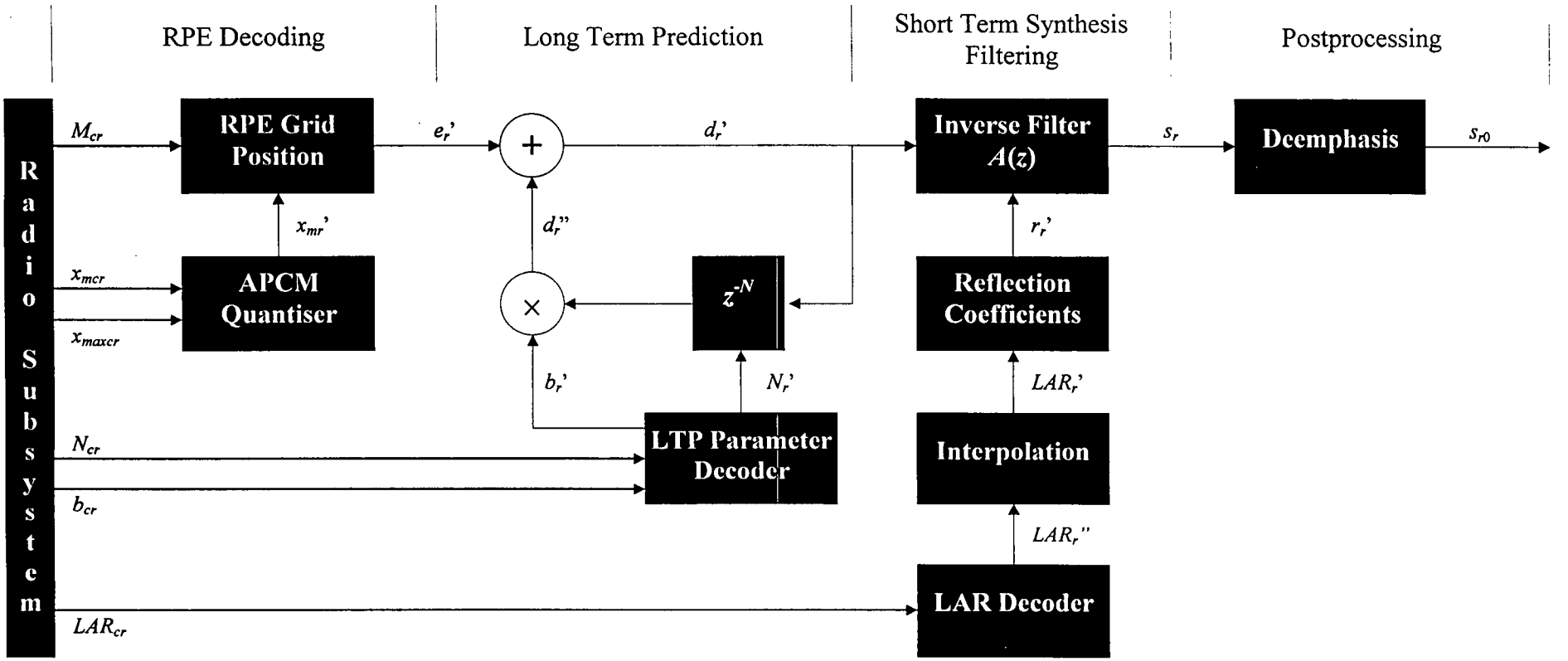


Figure B.2 Block Diagram of the GSM full-rate decoder.

Appendix C Tables Used in the Implementation of the GSM Full-Rate Codec

Table C.1 Quantisation of the log area ratios.

i	A[i]	B[i]	MIC[i]	MAC[I]
1	20480	0	-32	31
2	20480	0	-32	31
3	20480	2048	-16	15
4	20480	-2560	-16	15
5	13964	94	-8	7
6	15360	-1792	-8	7
7	8534	-341	-4	3
8	9036	-1144	-4	3

Table C.2 Tabulation of a/A[1...8].

I	INVA[i]
1	13107
2	13107
3	13107
4	13107
5	19223
6	17476
7	31454
8	29708

Table C.3 Decision level of the LTP gain quantiser.

bc	DLB[bc]
0	6554
1	16384
2	26214
3	32767

Table C.4 Quantisation levels of the LTP gain quantiser.

bc	QLB[bc]
0	3277
1	11469
2	21299
3	32767

Table C.5 Coefficients of the weighting filter.

i	H[i]
0	-134
1	-374
2	0
3	2054
4	5741
5	8192
6	5741
7	2054
8	0
9	-374
10	-134

Table C.6 Normalised inverse mantissa used to compute xM/xmax.

i	NRFAC[i]
0	29128
1	26215
2	23832
3	21846
4	20165
5	18725
6	17476
7	16384

Table C.7 Normalised direct mantissa used to compute xM/xmax.

i	FAC[i]
0	18431
1	20479
2	22527
3	24575
4	26623
5	28671
6	30719
7	32767

Appendix D GSM Full-Rate Encoder Output Parameters

Table D.1 GSM full-rate encoder output parameters in order of occurrence and bit allocation within the speech frame of 260 bits/20 ms.

Parameter	Parameter Number	Parameter Name	Variable Name	Number of Bits	Bits Number (LSB-MSB)
Filter Parameters	1	Log-Area ratios 1-8	LAR 1	6	b1 - b6
	2		LAR 2	6	b7 - b12
	3		LAR 3	5	b13 - b17
	4		LAR 4	5	b18 - b22
	5		LAR 5	4	b23 - b26
	6		LAR 6	4	b27 - b30
	7		LAR 7	3	b31 - b33
	8		LAR 8	3	b34 - b36
Sub-frame no. 1					
LTP Parameters	9	LTP lag	N1	7	b37 - b43
	10	LTP gain	b1	2	b44 - b45
RPE Parameters	11	RPE grid positions	M1	2	b46 - b47
	12	Block amplitude	xmax	6	b48 - b53
	13	RPE - pulse no. 1	x1(0)	3	b54 - b56
	14	RPE - pulse no. 2	x1(1)	3	b57 - b59
	15	RPE - pulse no. 3	x1(2)	3	b60 - b62
	16	RPE - pulse no. 4	x1(3)	3	b63 - b65
	17	RPE - pulse no. 5	x1(4)	3	b66 - b68
	18	RPE - pulse no. 6	x1(5)	3	b69 - b71
	19	RPE - pulse no. 7	x1(6)	3	b72 - b74
	20	RPE - pulse no. 8	x1(7)	3	b75 - b77
	21	RPE - pulse no. 9	x1(8)	3	b78 - b80
	22	RPE - pulse no. 10	x1(9)	3	b81 - b83
	23	RPE - pulse no. 11	x1(10)	3	b84 - b86
	24	RPE - pulse no. 12	x1(11)	3	b87 - b89
	25	RPE - pulse no. 13	x1(12)	3	b90 - b92

Sub-frame no. 2					
LTP Parameters	26	LTP lag	N2	7	b93 - b99
	27	LTP gain	b2	2	b100 - b101
RPE Parameters	28	RPE grid positions	M2	2	b102 - b103
	29	Block amplitude	xmax2	6	b104 - b109
	30	RPE - pulse no. 1	x2(0)	3	b110 - b112
	31	RPE - pulse no. 2	x2(1)	3	b113 - b115
	32	RPE - pulse no. 3	x2(2)	3	b116 - b118
	33	RPE - pulse no. 4	x2(3)	3	b119 - b121
	34	RPE - pulse no. 5	x2(4)	3	b122 - b124
	35	RPE - pulse no. 6	x2(5)	3	b125 - b127
	36	RPE - pulse no. 7	x2(6)	3	b128 - b130
	37	RPE - pulse no. 8	x2(7)	3	b131 - b133
	38	RPE - pulse no. 9	x2(8)	3	b134 - b136
	39	RPE - pulse no. 10	x2(9)	3	b137 - b139
	40	RPE - pulse no. 11	x2(10)	3	b140 - b142
	41	RPE - pulse no. 12	x2(11)	3	b143 - b145
	42	RPE - pulse no. 13	x2(12)	3	b146 - b148
Sub-frame no. 3					
LTP Parameters	43	LTP lag	N3	7	b149 - b155
	44	LTP gain	b3	2	b156 - b157
RPE Parameters	45	RPE grid positions	M3	2	b158 - b159
	46	Block amplitude	xmax3	6	b160 - b165
	47	RPE - pulse no. 1	x3(0)	3	b166 - b168
	48	RPE - pulse no. 2	x3(1)	3	b169 - b171
	49	RPE - pulse no. 3	x3(2)	3	b172 - b174
	50	RPE - pulse no. 4	x3(3)	3	b175 - b177
	51	RPE - pulse no. 5	x3(4)	3	b178 - b180
	52	RPE - pulse no. 6	x3(5)	3	b181 - b183
	53	RPE - pulse no. 7	x3(6)	3	b184 - b186
	54	RPE - pulse no. 8	x3(7)	3	b187 - b189
	55	RPE - pulse no. 9	x3(8)	3	b190 - b192
	56	RPE - pulse no. 10	x3(9)	3	b193 - b195
	57	RPE - pulse no. 11	x3(10)	3	b196 - b198
	58	RPE - pulse no. 12	x3(11)	3	b199 - b201
	59	RPE - pulse no. 13	x3(12)	3	b202 - b204

Sub-frame no. 4					
LTP Parameters	60	LTP lag	N4	7	b205 - b211
	61	LTP gain	b4	2	b212 - b213
RPE Parameters	62	RPE grid positions	M4	2	b214 - b215
	63	Block amplitude	xmax4	6	b216 - b221
	64	RPE - pulse no. 1	x4(0)	3	b222 - b224
	65	RPE - pulse no. 2	x4(1)	3	b225 - b227
	66	RPE - pulse no. 3	x4(2)	3	b228 - b230
	67	RPE - pulse no. 4	x4(3)	3	b213 - b233
	68	RPE - pulse no. 5	x4(4)	3	b234 - b236
	69	RPE - pulse no. 6	x4(5)	3	b237 - b239
	70	RPE - pulse no. 7	x4(6)	3	b240 - b242
	71	RPE - pulse no. 8	x4(7)	3	b243 - b245
	72	RPE - pulse no. 9	x4(8)	3	b246 - b248
	73	RPE - pulse no. 10	x4(9)	3	b249 - b251
	74	RPE - pulse no. 11	x4(10)	3	b252 - b254
	75	RPE - pulse no. 12	x4(11)	3	b255 - b257
	76	RPE - pulse no. 13	x4(12)	3	b258 - b260

Appendix E GSM Half-Rate Encoder Output Parameters

Table E.1 shows a list of all the parameters which are coded for each 20 ms speech frame. The data rate of the speech coder is 5.6 kbps. Therefore 20 ms speech frame consists of 112 bits.

Table E.1 GSM half-rate encoder output parameters in order of occurrence and bit allocation within the speech frame of 112 bits/20 ms.

Parameter	Parameter Number	Parameter Name	Variable Name	Number of Bits	Bits Number (LSB-MSB)
Filter Parameters	1	Voicing mode	MODE	2	b1 - b2
	2	Frame energy	R0	5	b3 - b7
	3	Reflection coefficient vector r1 - r3	LPC1	11	b8 - b18
	4	Reflection coefficient vector r4 - r6	LPC2	9	b19 - b27
	5	Reflection coefficient vector r4- r10	LPC3	8	b28 - b35
	6	Soft interpolation bit for frame	INT_LPC	1	b36
Subframe Bits (MODE = 1, 2, or 3)	7	Lag for first subframe	LAG_1	8	b37 - b44
	8	Lag delta code for second subframe	LAG_2	4	b45 - b48
	9	Lag delta code for third subframe	LAG_3	4	b49 - b52
	10	Lag delta code for fourth subframe	LAG_4	4	b53 - b56
	11	Codebook, I, for first subframe	CODE_1	9	b57 - b65
	12	Codebook, I, for second subframe	CODE_2	9	b66 - b74
	13	Codebook, I, for third subframe	CODE_3	9	b75 - b83
	14	Codebook, I, for fourth subframe	CODE_4	9	b84 - b92
	15	{P0,GS} code for first subframe	GSP0_1	5	b93 - b97
	16	{P0,GS} code for second subframe	GSP0_2	5	b98 - b102
	17	{P0,GS} code for third subframe	GSP0_3	5	b103 - b107
	18	{P0,GS} code for fourth subframe	GSP0_4	5	b108 - b112

Subframe Bits (MODE = 0)	7	Codebook, I, for first subframe	CODE1_1	7	b37 - b43
	8	Codebook, H, for first subframe	CODE2_1	7	b44 - b50
	9	Codebook, I, for second subframe	CODE1_2	7	b51 - b57
	10	Codebook, H, for second subframe	CODE2_2	7	b58 - b64
	11	Codebook, I, for third subframe	CODE1_3	7	b65 - b71
	12	Codebook, H, for third subframe	CODE2_3	7	b72 - b78
	13	Codebook, I, for fourth subframe	CODE1_4	7	b79 - b85
	14	Codebook, H, for fourth subframe	CODE2_4	7	b86 - b92
	15	{P0,GS} code for first subframe	GSP0_1	5	b93 - b97
	16	{P0,GS} code for second subframe	GSP0_2	5	b98 - b102
	17	{P0,GS} code for third subframe	GSP0_3	5	b103 - b107
	18	{P0,GS} code for fourth subframe	GSP0_4	5	b108 - b112

E.1 MODE

The speech coder is defined by four voicing modes. MODE is a 2 bit code which specifies which of the four voiving modes is used at the current frame. The MODE indicates which definition of the frame bits to apply to the current frame.

E.2 R0

R0 represents the average signal power of the input speech for the frame. The average signal power is computed using an analysis window which is centred over the last 100 samples of the frame.

E.3 LPC1 - LPC3

The 10 reflection coefficients are vector quantised in three vector segments. The first vector segment codes reflection coefficients r1 - r3, the second vector segment codes coefficients r4 - r6, the third vector segment codes coefficients r7 - r10.

E.4 LAG_1 - LAG_4

LPG_1, the lag for the first subframe, can take on the value in the range of 21 to 142. Eight bits are used to encode the lag which may be fractional in value. Each of the remaining lag values (LAG_2 through LAG_4) is delta coded relative to the preceding subframe's coded

value of the lag, with a deviation of -8 to +7 allowable lag value levels specified by a four bit code.

E.5 CODEx_1 - CODEx_4

If MODE_0, the code value for the VSELP codebook is the codework I as derived by the codebook search procedure. If MODE = 0, two VSELP codebooks are sequentially searched, with codeword I, specifying the codevector from the first VSELP codebook, assigned onto CODE1_x, and codeword H, specifying the codeword selected from the second VSELP codebook, assigned onto CODE2_x, which x is the subframe number.

E.6 GSP0_1 - GSP0_4

The {*P0*,*GS*} codebook contains the values needed to determine the gain factors for the excitation vectors of a given subframe. The index of the corresponding codebook entry is assigned to *GSP0_x*. The half-rate coder is a multimode speech coder, defined by four voicing modes

MODE = 0 unvoiced

MODE = 1 slightly voiced

MODE = 2 moderately voiced

MODE = 3 strongly voiced

If MODE = 0, the adaptive codebook (long-term predictor) and the VSELP codebook are replaced by two other VSELP codebooks.

Appendix F GSM Full-Rate Speech Compression C code

```
/*************************************************************************
                    gsmdef.h --- global declarations
*************************************************************************/
typedef int word;
typedef int l_word;

/*************************************************************************
       gsmoper.c -- low-level operations needed in the GSM Speech Codec
*************************************************************************/
word add (word var1, word var2);
word sub (word var1, word var2);
word mult (word var1, word var2);
word mult_r (word var1, word var2);
word Abs (word var);
word div (word var1, word var2);
l_word l_mult (word var1, word var2);
l_word l_add (l_word var1, l_word var2);
l_word l_sub (l_word var1, l_word var2);
word norm (l_word var);
l_word s_to_l (word var);
word l_to_s (l_word var);

/*************************************************************************
              gsmio.c -- binary input/output procedures
*************************************************************************/
int read_data (word sop[160]);
int read_code (word LARc[9], word Nc[4], word bc[4], word Mc[4],
               word xmaxc[4], word xMc[4][13]);
void write_data (word sop[160]);
void write_code (word LARc[9], word Nc[4], word bc[4], word Mc[4],
                 word xmaxc[4], word xMc[4][13]);

/*************************************************************************
              gsmtab.c -- tables for GSM Full Rate Codec
*************************************************************************/
extern word A[9];
extern word B[9];
extern word MIC[9];
extern word MAC[9];
extern word INVA[9];
extern word DLB[4];
extern word QLB[4];
extern word H[11];
extern word NRFAC[8];
extern word FAC[8];
```

```
/*************************************************************************
              gsmtab.c -- tables for GSM Full Rate Codec
*************************************************************************/

#include "gsmdef.h"

word A[9] = {0, 20480, 20480, 20480, 20480, 13964, 15360, 8534, 9036};
word B[9] = {0, 0, 0, 2048, -2560, 94, -1792, -341, -1144};
word MIC[9] = {0, -32, -32, -16, -16, -8, -8, -4, -4};
word MAC[9] = {0, 31, 31, 15, 15, 7, 7, 3, 3};
word INVA[9] = {0, 13107, 13107, 13107, 13107, 19223, 17476, 31454, 29708};
word DLB[4] = {6554, 16384, 26214, 32767};
word QLB[4] = {3277, 11469, 21299, 32767};
word H[11] = {-134, -374, 0, 2054, 5741, 8192, 5741, 2054, 0, -374, -134};
word NRFAC[8] = {29128, 26215, 23832, 21846, 20165, 18725, 17476, 16384};
word FAC[8] = {18431, 20479, 22527, 24575, 26623, 28671, 30719, 32767};

/*************************************************************************
       gsmoper.c -- low-level operations needed in the GSM Speech Codec
*************************************************************************/

#include<stdio.h>
#include "gsmdef.h"


/*************************************************************************
                        add -- perform 16-bit addition
*************************************************************************/
word add (word var1, word var2)
{
  word tmp;

  tmp = var1 + var2;

  if (tmp < -32768)
    tmp = -32768;
  else if (tmp > 32767)
    tmp = 32767;

  return tmp;
}

/*************************************************************************
                        sub -- perform 16-bit subtraction
*************************************************************************/
word sub (word var1, word var2)
```

```
{
  word tmp;

  tmp = var1 - var2;
  if (tmp < -32768)
    tmp = -32768;
  else if (tmp > 32767)
    tmp = 32767;

  return tmp;
}

/*****************************************************************************
              mult -- perform 16-bit multiplication
*****************************************************************************/
word mult (word var1, word var2)
{
  if ((var1 == -32768) && (var2 == -32768))
    return 32767;
  else
    return (var1 * var2) >> 15;
}

/*****************************************************************************
       mult_r  -- perform 16-bit multiplication with rounding
*****************************************************************************/
word mult_r (word var1, word var2)
{
  l_word tmp;

  if ((var1 == -32768) && (var2 == -32768))
    return 32767;
  else {
    tmp = l_add(var1 * var2, 16384);
    return tmp >> 15;
  }
}

/*****************************************************************************
              Abs -- perform 16-bit absolute value-operation
*****************************************************************************/
word Abs (word var)
{
  if (var == -32768)
    return 32767;
  else
    return ((var < 0) ? -var : var);
}

/*****************************************************************************
              div -- perform 16-bit fractional integer division
*****************************************************************************/
word div (word num, word denum)
{
  int k;
  l_word l_num, l_denum;
  word res;

  if (num == denum)
    return 32767;
  else {
    l_num = s_to_l(num);
    l_denum = s_to_l(denum);
    res = 0;

    for (k=0; k < 15; k++) {
      res = res << 1;
      l_num = l_num << 1;
      if (l_num >= l_denum) {
        l_num = l_sub(l_num, l_denum);
        res = add(res, 1);
      }
    }

    return res;
  }
}

/*****************************************************************************
              l_mult -- perform 32-bit multiplication
*****************************************************************************/
l_word l_mult (word var1, word var2)
{
  if ((var1 == -32768) && (var2 == -32768)) {
    exit(1);
  } else
    return (var1 * var2) << 1;
}

/*****************************************************************************
              l_add -- perform 32-bit addition
*****************************************************************************/
l_word l_add (l_word var1, l_word var2)
{
  l_word tmp;
  tmp = var1 + var2;
  if ((var1 > 0) && (var2 > 0))       /* same signs -> possible problems */
    if (tmp <= 0)                     /* sign change == overflow */
      tmp = ~(-1 << 31);              /* 2147483647 */
  if ((var1 < 0) && (var2 < 0)) /* same signs -> possible problems */
    if (tmp >= 0)                     /* sign change == overflow */
      tmp = -1 << 31;                 /* -2147483648 */
```

```
  return tmp;
}

/*****************************************************************************
                l_sub -- perform 32-bit subtraction
*****************************************************************************/
l_word l_sub (l_word var1, l_word var2)
{
  l_word tmp;

  tmp = var1 - var2;
  if ((var1 > 0) && (var2 < 0))       /* different signs -> possible problems
*/
    if (tmp <= 0)                     /* sign change == overflow */
      tmp = ~(-1 << 31);              /* 2147483647 */
  if ((var1 < 0) && (var2 > 0)) /* different signs -> possible problems */
    if (tmp >= 0)                     /* sign change == overflow */
      tmp = -1 << 31;                 /* -2147483648 */

  return tmp;

}

/*****************************************************************************
  norm -- counts the number of left shift operations needed to normalize
          a 32-bit value.
*****************************************************************************/
word norm (l_word var)
{
  int n = 0;

  if (var < 0)                  /* handle negative by converting to positive */
    if (var != (-1 << 31)) /* -2147483648 (this gives 0 by definition) */
      var = -var;

  if (var > 0)
    while ((var & (1 << 30)) == 0) { /* shift left while bit #30 == 0 */
      var = var << 1;
      n++;
    }

  return n;
}

/*****************************************************************************
       s_to_l -- converts a 16-bit integer to 32-bit integer with sign
                 extension
*****************************************************************************/
l_word s_to_l (word var)
{
  return (l_word) var;
}

/*****************************************************************************
       l_to_s -- extacts the 16 LSB of a 32-bit value for conversion to
                 16-bit integer (word)
*****************************************************************************/
word l_to_s (l_word var)
{
  return (word) var;
}

/*****************************************************************************
               gsmio.c -- binary input/output procedures
*****************************************************************************/

#include <stdio.h>
#include "gsmdef.h"

#define MAX_BUF 320

extern FILE *f_read, *f_write;

/*****************************************************************************
       do_read --- reads words from binary file via FILE *f_read
*****************************************************************************/
int do_read (word *data, int len)
{
  /* extern FILE *f_read; */
  char buff[MAX_BUF];
  int i;

  if (fread(buff, 2, len,f_read ) == len) {
    for (i = 0; i < len; i++) {
      data[i] = (buff[i*2] & 0xff) | ((buff[i*2 + 1] << 8) & ~0xff);
    }
    return 1;
  }
  return(0);
}

/*****************************************************************************
              do_write --- writes words to a binary file
*****************************************************************************/
void do_write (word *data, int len)
{
  char buff[MAX_BUF];
  int i;

  for (i = 0; i < len; i++) {
    buff[i*2]     = (char)data[i]; /* & 0xff;  */
```

```
    buff[i*2 + 1] = (char)(data[i] >> 8); /* & 0xff;  */
  }
  fwrite(buff, 1, 2*len, f_write);
}

/*****************************************************************************
       read_data --- reads a speech data frame using 'do_read'
*****************************************************************************/
int read_data(word sop[160])
{
  return do_read(sop, 160);
}

/*****************************************************************************
       read_code --- reads a coded speech data frame using 'do_read'
*****************************************************************************/
int read_code (word LARc[9], word Nc[4], word bc[4], word Mc[4],
               word xmaxc[4], word xMc[4][13])
{
  word temp[76];
  int i, j;

  if (do_read(temp, 76)) {
    for (i = 0; i < 8; i++) {
      LARc[i+1] = temp[i];
    }
    for (i = 0; i < 4; i++) {
      Nc[i]    = temp[8+i*17];
      bc[i]    = temp[8+i*17+1];
      Mc[i]    = temp[8+i*17+2];
      xmaxc[i] = temp[8+i*17+3];
      for (j = 0; j < 13; j++) {
        xMc[i][j] = temp[8+i*17+4+j];
      }
    }
    return 1;
  }
  return 0;
}

/*****************************************************************************
       write_data --- writes speech data frame using 'do_write'
*****************************************************************************/
void write_data (word sop[160])
{
  do_write(sop, 160);
}

/*****************************************************************************
       write_code --- writes a coded speech data frame using 'do_write'
*****************************************************************************/
void write_code (word LARc[9], word Nc[4], word bc[4], word Mc[4],
                 word xmaxc[4], word xMc[4][13])
{
  word temp[76];
  int i, j;

  for (i = 0; i < 8; i++) {
    temp[i] = LARc[i+1];
  }
  for (i = 0; i < 4; i++) {
    temp[8+i*17] = Nc[i];
    temp[8+i*17+1] = bc[i];
    temp[8+i*17+2] = Mc[i];
    temp[8+i*17+3] = xmaxc[i];
    for (j = 0; j < 13; j++) {
      temp[8+i*17+4+j] = xMc[i][j];
    }
  }
  do_write(temp, 76);
}

/*****************************************************************************
                gsmenc.c -- GSM Full Rate Encoder
*****************************************************************************/

#include <stdio.h>
#include "gsmdef.h"
#include "gsmenc.h"

/* Global Declarations */
FILE *f_read, *f_write;

/*****************************************************************************
                          main -- The main program
*****************************************************************************/
void main (int argc, char *argv[])
{
  int i,k;

  word sop[160];                                          /* input  */
  word last_LARpp[9], dp[121];                            /* memory */
  word LARc[9], Nc[4], bc[4], Mc[4], xmaxc[4], xMc[4][13]; /* output */

  if ((f_read=fopen(argv[1],"rb")) == NULL) {
    fprintf(stderr, "%s: can't open %s\n", argv[0], argv[1]);
    exit(1);
  }
  if ((f_write=fopen(argv[2],"wb")) == NULL) {
    fprintf(stderr, "%s: can't open %s\n", argv[0], argv[2]);
    exit(1);
    fclose(f_read);
```

```
  }

  /* initalization of arrays */
  for (i = 1; i < 9; i++) {
    last_LARpp[i] = 0;
  }

  for (i = 1; i < 121; i++) {
    dp[i] = 0;
  }

  while(read_data(sop)) {
    code_block(sop, last_LARpp, dp, LARc, Nc, bc, Mc, xmaxc, xMc);
    write_code(LARc, Nc, bc, Mc, xmaxc, xMc);
  }
       fclose(f_read);
       fclose(f_write);
}

/*************************************************************************
              code_block -- encodes one block of 160 samples
*************************************************************************/
void code_block (word sop[160], word last_LARpp[9], word dp[121],
                 word LARc[9], word Nc[4], word bc[4], word Mc[4],
                 word xmaxc[4], word xMc[4][13])
{
  int i, k_start, k_end;
  l_word l_ACF[9];

  word exp, mant;
  word so[160], sof[160], s[160];
  word LAR[9], LARpp[9], LARp[9];
  word r[9], rp[9];
  word d[160], dpp[40];
  word e[40], ep[40];
  word x[40];
  word xM[13], xMp[13];

  /* PREPROCESSING */
  scaling(sop, so);
  offset(so, sof);
  preemp(sof, s);

  /* SHORT TERM CODING SECTION */
  autocorr(s, l_ACF);
  schur(l_ACF, r);
  lars(r, LAR);
  code_lar(LAR, LARc);
  decode_lar(LARc, LARpp);

  /* PERFORM THE SHORT TERM ANALYSIS FILTERING (in four parts) */
  /* for start_k = 0 to end_k = 12 */
  inte_lar(0, last_LARpp, LARpp, LARp);
  coeffs(LARp, rp);
  i_filter(0, 12, rp, s, d);

  /* for start_k = 13 to end_k = 26 */
  inte_lar(13, last_LARpp, LARpp, LARp);
  coeffs(LARp, rp);
  i_filter(13, 26, rp, s, d);

  /* for start_k = 27 to end_k = 39 */
  inte_lar(27, last_LARpp, LARpp, LARp);
  coeffs(LARp, rp);
  i_filter(27, 39, rp, s, d);

  /* for start_k = 40 to end_k = 159 */
  inte_lar(40, last_LARpp, LARpp, LARp);
  coeffs(LARp, rp);
  i_filter(40, 159, rp, s, d);

  /* update the last_LARpp array */
  for (i = 1; i < 9; i++) {
    last_LARpp[i] = LARpp[i];
  }

  /* LONG TERM CODING SECTION */
  /* once for each sub-frame */
  for(i = 0; i < 4; i++) {
    ltp_calc(d+(i*40), dp, bc+i, Nc+i);
    ltp_filter(d+(i*40), dp, bc[i], Nc[i], e, dpp);
    w_filter(e, x);
    rpe_select(x, xM, Mc+i);
    apcm_quant(xM, xMc[i], xmaxc+i, &exp, &mant);
    apcm_iquant(exp, mant, xMc[i], xMp);
    rpe_posit(Mc[i], xMp, ep);
    update_dp(ep, dpp, dp);
  }
}

/*************************************************************************
                          PREPROCESSING SECTION
*************************************************************************/

/*************************************************************************
                          scaling -- sec. 4.2.1
*************************************************************************/
void scaling (word sop[160], word so[160])
{
  int k;

  for (k = 0; k < 160; k++) {
```

```
    so[k] = sop[k] >> 3;
    so[k] = so[k]  << 2;
  }
}

/*************************************************************************
              offset --- Offset compensation. sec. 4.2.2
*************************************************************************/
void offset (word so[160], word sof[160])
{
  int k;
  word s1, msp, lsp, temp;
  l_word l_s2;
  static word z1   = 0;
  static l_word l_z2 = 0;

  for (k = 0; k < 160; k++) {
    /* compute the non-recursive part */
    s1 = sub(so[k], z1);
    z1 = so[k];

    /* compute recursive part */
    l_s2 = s_to_l(s1);
    l_s2 = l_s2 << 15;

    /* execution of a 31 by 16 muiltiplication */
    msp = l_to_s(l_z2 >> 15);
    lsp = l_to_s(l_sub(l_z2, s_to_l(msp) << 15));
    temp = mult_r(lsp, 32735);
    l_s2 = l_add(l_s2, s_to_l(temp));
    l_z2 = l_add(l_mult(msp, 32735) >> 1, l_s2);

    /* Compute sof[k] with rounding */
    sof[k] = l_to_s(l_add(l_z2, 16384) >> 15);
  }
}

/*************************************************************************
              preemp -- Pre-emphasis, sec. 4.2.3
*************************************************************************/
void preemp (word sof[160], word s[160])
{
  int k;
  static word mp = 0;

  for (k = 0; k < 160; k++) {
    s[k] = add(sof[k], mult_r(mp, -28180));
    mp = sof[k];
  }
}

/*************************************************************************
                        LPC ANALYSIS SECTION
*************************************************************************/

/*************************************************************************
              autocorr -- Autocorrelation, sec. 4.2.4
*************************************************************************/
void autocorr (word s[160], l_word l_ACF[9])
{
  int    i, j, k;
  word   smax, temp, scalauto;
  l_word l_temp;

  /* dynamic scaling of the array s[0..159] */

  /* search for the maximum */
  smax = 0;
  for (k = 0; k < 160; k++) {
    temp = Abs(s[k]);
    if (temp > smax)
      smax = temp;
  }

  /* computation of the scaling factor */
  if (smax == 0)
    scalauto = 0;
  else
    scalauto = sub(4, norm(s_to_l(smax) << 16));

  /* scaling the array s[0..159] */
  if (scalauto > 0) {
    temp = 16384 >> sub(scalauto, 1);
    for (k = 0; k < 160; k++) {
      s[k] = mult_r(s[k], temp);
    }
  }

  /* compute l_ACF[..] */
  for (k = 0; k < 9; k++) {
    l_ACF[k] = 0;
    for (i = k; i < 160; i++) {
      l_temp = l_mult(s[i], s[i-k]);
      l_ACF[k] = l_add(l_ACF[k], l_temp);
    }
  }

  /* rescaling of the array s[0..159] */
  if (scalauto > 0) {
    for (k = 0; k < 160; k++) {
      s[k] = s[k] << scalauto;
    }
```

```
  }
}

/*************************************************************************
schur -- Computation of the reflection coefficients, sec. 4.2.5

input:  l_ACF
output: r
used:
*************************************************************************/
void schur (l_word l_ACF[9], word r[9])
{
  int i, k, m, n;
  word ACF[9], K[9], P[9], temp;

  if (l_ACF[0] == 0) {
    for (i = 1; i < 9; i++) {
      r[i] = 0;
    }
    return;
  }

  temp = norm(l_ACF[0]);
  for (k = 0; k < 9; k++) {
    ACF[k] = l_to_s((l_ACF[k] << temp) >> 16);
  }

  /* initalize arrays P[] and K[] for the recursion */
  for (i = 1; i < 8; i++) {
    K[9-i] = ACF[i];
  }

  for (i = 0; i < 9; i++) {
    P[i] = ACF[i];
  }

  /* compute reflection coefficients */
  for (n = 1; n < 9; n++) {
    if (P[0] < Abs(P[1])) {
      for (i = n; i < 9; i++) {
        r[i] = 0;
      }
      return;
    }

    r[n] = div(Abs(P[1]), P[0]);

    if (P[1] > 0)
      r[n] = sub(0, r[n]);

    if (n == 8)
      return;

    /* The Shur recursion */
    P[0] = add(P[0], mult_r(P[1], r[n]));
    for (m = 1; m < (8 - n) + 1; m++) {
      P[m] = add(P[m+1], mult_r(K[9-m], r[n]));
      K[9-m] = add(K[9-m], mult_r(P[m+1], r[n]));
    }
  }
}

/*************************************************************************
lars -- transformation of reflection coefficients to Log.-Area Ratios
        sec. 4.2.6
*************************************************************************/
void lars (word r[9], word LAR[9])
{
  int i;
  word temp;

  for (i = 1; i < 9; i++) {
    temp = Abs(r[i]);
    if (temp < 22118)
      temp = temp >> 1;
    else if (temp < 31130)
      temp = sub(temp, 11059);
    else
      temp = sub(temp, 26112) << 2;

    LAR[i] = temp;
    if (r[i] < 0)
      LAR[i] = sub(0, LAR[i]);
  }
}

/*************************************************************************
code_lar -- Quantization and coding of the Log. Area Ratios, sec. 4.2.7
*************************************************************************/
void code_lar (word LAR[9], word LARc[9])
{
  int i;
  word temp;

  for (i = 1; i < 9; i++) {
    temp = mult(A[i], LAR[i]);
    temp = add(temp, B[i]);
    temp = add(temp, 256);
    LARc[i] = temp >> 9;

    /* check if LARc[i] lies between MIN and MAX */
    if (LARc[i] > MAC[i])
```

```
      LARc[i] = MAC[i];
    if (LARc[i] < MIC[i])
      LARc[i] = MIC[i];

    LARc[i] = sub(LARc[i], MIC[i]);
  }
}

/*************************************************************************
              SHORT TERM ANALYSIS FILTERING SECTION
*************************************************************************/

/*************************************************************************
decode_lar -- Decoding of the coded Log. Area Ratios, sec. 4.2.8
*************************************************************************/
void decode_lar (word LARc[9], word LARpp[9])
{
  int i;
  word temp;

  for (i = 1; i < 9; i++) {
    temp = add(LARc[i], MIC[i]) << 10;
    temp = sub(temp, B[i] << 1);
    temp = mult_r(INVA[i], temp);
    LARpp[i] = add(temp, temp);
  }
}

/*************************************************************************
       inte_lar -- Interpolation of the LARpp[1..8] to get LARp[1..8],
       sec. 4.2.9.1
*************************************************************************/
void inte_lar (int k_start, word last_LARpp[9], word LARpp[9], word LARp[9])
{
  int i;

  switch(k_start) {
  case 0:
    for (i = 1; i < 9; i++) {
      LARp[i] = add(last_LARpp[i] >> 2, LARpp[i] >> 2);
      LARp[i] = add(LARp[i], last_LARpp[i] >> 1);
    }
    break;

  case 13:
    for (i = 1; i < 9; i++) {
      LARp[i] = add(last_LARpp[i] >> 1, LARpp[i] >> 1);
    }
    break;

  case 27:
    for (i = 1; i < 9; i++) {
      LARp[i] = add(last_LARpp[i] >> 2, LARpp[i] >> 2);
      LARp[i] = add(LARp[i], LARpp[i] >> 1);
    }
    break;

  case 40:
    for (i = 1; i < 9; i++) {
      LARp[i] = LARpp[i];
    }
    break;

  default:
    exit(1);
  }
}

/*************************************************************************
coeffs -- Computation of the rp[1..8] from the interpolated LARp[1..8]
sec. 4.2.9.2
*************************************************************************/
void coeffs (word LARp[9], word rp[9])
{
  int i;
  word temp;

  for (i = 1; i < 9; i++) {
    temp = Abs(LARp[i]);
    if (temp < 11059)
      temp = temp << 1;
    else if (temp < 20070)
      temp = add(temp, 11059);
    else
      temp = add(temp >> 2, 26112);

    rp[i] = temp;
    if (LARp[i] < 0)
      rp[i] = sub(0, rp[i]);
  }
}

/*************************************************************************
       i_filter -- Short term analysis filtering, sec. 4.2.10
*************************************************************************/
void i_filter (int k_start, int k_end, word rp[9], word s[160], word d[160])
{
  int k, i;
  word sav, temp, di;
  static word u[8] = {0, 0, 0, 0, 0, 0, 0, 0};

  for (k = k_start; k <= k_end; k++) {
```

```
    di = s[k];
    sav = di;
    for (i = 1; i < 9; i++) {
      temp = add(u[i-1], mult_r(rp[i], di));
      di = add(di, mult_r(rp[i], u[i-1]));
      u[i-1] = sav;
      sav = temp;
    }
    d[k] = di;
  }
}

/************************************************************************
              LONG TERM PREDICTION (LTP) SECTION
************************************************************************/

/************************************************************************
       ltp_calc -- Calculation of the LTP parameters, sec. 4.2.11
************************************************************************/
void ltp_calc (word d[40], word dp[121], word *bc_p, word *Nc_p)
{
  int k, lambda;
  word dmax, temp, scal, wt[40], R, S;
  l_word l_max, l_result, l_power, l_temp;

  /* Search of the optimum scaling of d[0..39] */
  dmax = 0;
  for (k = 0; k < 40; k++) {
    temp = Abs(d[k]);
    if (temp > dmax)
      dmax = temp;
  }

  temp = 0;
  if (dmax == 0)
    scal = 0;
  else
    temp = norm(dmax << 16);

  if (temp > 6)
    scal = 0;
  else
    scal = sub(6, temp);

  /* initalization of a working array wt */
  for (k = 0; k < 40; k++) {
    wt[k] = d[k] >> scal;
  }

  /* Search of the maximum cross-corelation and coding of the LTP lag */
  l_max = 0;
  *Nc_p = 40;
  for (lambda = 40; lambda < 121; lambda++) {
    l_result = 0;
    for (k = 0; k < 40; k++) {
      l_temp = l_mult(wt[k], dp[lambda - k]);
      l_result = l_add(l_temp, l_result);
    }
    if (l_result > l_max) {
      *Nc_p = lambda;
      l_max = l_result;
    }
  }

  /* rescaling of l_max */
  l_max = l_max >> sub(6, scal);

  /* initalization of a working array wt[0..39] */
  for (k = 0; k < 40; k++) {
    wt[k] = dp[*Nc_p - k] >> 3;
  }

  /* compute the power of the reconstructed short term residual signal dp[] */
  l_power = 0;
  for (k = 0; k < 40; k++) {
    l_temp = l_mult(wt[k], wt[k]);
    l_power = l_add(l_temp, l_power);
  }

  /* Normalization of l_max and l_power */
  if (l_max <= 0) {
    *bc_p = 0;
    return;
  }

  if (l_max >= l_power) {
    *bc_p = 3;
    return;
  }

  temp = norm(l_power);
  R = l_to_s((l_max << temp) >> 16);
  S = l_to_s((l_power << temp) >> 16);

  /* coding of the LTP gain */
  for (*bc_p = 0; *bc_p < 3; (*bc_p)++) {
    if (R <= mult(S, DLB[*bc_p]))
      return;
  }
  *bc_p = 3;
}
```

```
/*************************************************************************
       ltp_filter -- Long term analysis filtering, sec. 4.2.12
*************************************************************************/
void ltp_filter (word d[40], word dp[121], word bc, word Nc,
                 word e[40], word dpp[40])
{
  int k;
  word bp;

  /* Decoding of the coded LTP-gain */
  bp = QLB[bc];

  /* Calculating the array e[0..39] and the array dpp[0..39] */
  for (k = 0; k < 40; k++) {
    dpp[k] = mult_r(bp, dp[Nc-k]);
    e[k] = sub(d[k], dpp[k]);
  }
}

/*************************************************************************
                        RPE ENCODING SECTION
*************************************************************************/

/*************************************************************************
               w_filter -- Weighting filter, sec. 4.2.13
*************************************************************************/
void w_filter (word e[40], word x[40])
{
  int i, k;
  word wt[50];
  l_word l_result, l_temp;

  /* initalization of a temporary working array wt[49] */
  for (k = 0; k < 5; k++) {
    wt[k] = 0;
  }

  for (k = 5; k < 45; k++) {
    wt[k] = e[k-5];
  }

  for (k = 45; k < 50; k++) {
    wt[k] = 0;
  }

  /* compute the signal x[0..39] */
  for (k = 0; k < 40; k++) {
    l_result = 8192;
    for (i = 0; i < 11; i++) {
      l_temp = l_mult(wt[k+i], H[i]);
      l_result = l_add(l_result, l_temp);
    }
    l_result = l_add(l_result, l_result);
    l_result = l_add(l_result, l_result);
    x[k] = l_to_s(l_result >> 16);
  }
}

/*************************************************************************
              rpe_select -- RPE grid selection, sec. 4.2.14
*************************************************************************/
void rpe_select (word x[40], word xM[13], word *Mc_p)
{
  int i, m;
  word temp;
  l_word EM, l_result, l_temp;

  *Mc_p = 0;
  EM = 0;

  for (m = 0; m < 4; m++) {
    l_result = 0;
    for (i = 0; i < 13; i++) {
      temp = x[m+(3*i)] >> 2;
      l_temp = l_mult(temp, temp);
      l_result = l_add(l_temp, l_result);
    }

    if (l_result > EM) {
      *Mc_p = m;
      EM = l_result;
    }
  }

  /* Down sampling by a factor 3 toget the selected xM[0..12] RPE sequence */
  for (i = 0; i < 13; i++) {
    xM[i] = x[*Mc_p + (3*i)];
  }
}

/*************************************************************************
apcm_quant -- APCM quantization of the selected RPE sequence, sec. 4.2.15
*************************************************************************/
void apcm_quant (word xM[13], word xMc[13],
                 word *xmaxc_p, word *exp_p, word *mant_p)
{
  int i, itest;
  word temp, temp1, temp2, xmax, exp, mant;

  /* find the maximum absolute value xmax of xM[0..12] */
  xmax = 0;
  for (i = 0; i < 13; i++) {
```

```
  temp = Abs(xM[i]);
  if (temp > xmax)
    xmax = temp;
}

/* Quantizing and coding of xmax to get *xmaxc_p */
exp = 0;
temp = xmax >> 9;
itest = 0;

for (i = 0; i < 6; i++) {
  if (temp <= 0) {
    itest = 1;
  }
  temp = temp >> 1;
  if (itest == 0) {
    exp = add(exp, 1);
  }
}
temp = add(exp, 5);
*xmaxc_p = add(xmax >> temp, exp << 3);

/* Quantizing and coding of the xM[[0..12] RPE sequence to get
   the xMc[0..12] */
/* compute exponent and mantissa decoded version of *xmaxc_p */
exp = 0;
if (*xmaxc_p > 15)
  exp = sub(*xmaxc_p >> 3, 1);

mant = sub(*xmaxc_p, exp << 3);

/* normalize mantissa 0 <= mant <= 7 */
if (mant == 0) {
  exp = -4;
  mant = 15;
}
else {
  itest = 0;
  for (i = 0; i < 3; i++) {
    if (mant > 7)
      itest = 1;
    if (itest == 0) {
      mant = add(mant << 1, 1);
      exp = sub(exp, 1);
    }
  }
}
mant = sub(mant, 8);

/* direct computation of xMc[0..12] using table 4.5 (NRFAC) */
temp1 = sub(6, exp);
temp2 = NRFAC[mant];
for (i = 0; i < 13; i++) {
  temp = xM[i] << temp1;
  temp = mult(temp, temp2);
  xMc[i] = add(temp >> 12, 4);
}
*exp_p = exp;
*mant_p = mant;
}

/*****************************************************************************
       apcm_iquant -- APCM inverse quantization, sec. 4.2.16
*****************************************************************************/
void apcm_iquant (word exp, word mant, word xMc[13], word xMp[13])
{
  int i;
  word temp, temp1, temp2, temp3;

  temp1 = FAC[mant];
  temp2 = sub(6, exp);
  temp3 = (sub(temp2, 1) >= 0) ?
          (1 << sub(temp2, 1)) : (1 >> -sub(temp2, 1));

  for (i = 0; i < 13; i++) {
    temp = sub(xMc[i] << 1, 7);
    temp = temp << 12;
    temp = mult_r(temp1, temp);
    temp = add(temp, temp3);
    xMp[i] = (temp2 >= 0) ? (temp >> temp2) : (temp << -temp2);
  }
}

/*****************************************************************************
              rpe_posit -- RPE grig positioning, sec. 4.2.17
*****************************************************************************/
void rpe_posit (word Mc, word xMp[13], word ep[40])
{
  int i, k;

  for (k = 0; k < 40; k++) {
    ep[k] = 0;
  }

  for (i = 0; i < 13; i++) {
    ep[Mc + (3*i)] = xMp[i];
  }
}

/*****************************************************************************
       update_dp -- Update of the reconstructed short term residual
                    signal dp[-120..-1] (really, [120..1]), sec. 4.2.18
```

```
*****************************************************************************/
void update_dp (word ep[40], word dpp[40], word dp[121])
{
  int k;

  for (k = 0; k < 80;  k++) {
    dp[120-k] = dp[80-k];
  }

  for (k = 0; k < 40; k++) {
    dp[40-k] = add(ep[k], dpp[k]);
  }
}

/*****************************************************************************
                gsmdec.c --- GSM Full Rate Decoder
*****************************************************************************/

#include <stdio.h>
#include "gsmdef.h"
#include "gsmdec.h"

/* Global Declarations */
FILE *f_read, *f_write;

/*****************************************************************************
                        main -- The main program.
*****************************************************************************/
void main (int argc, char *argv[])
{
  int i, j;

  word LARc[9], bc[4], Nc[4], Mc[4], xmaxc[4], xMc[4][13];  /* input  */
  word last_LARpp[9];                                       /* memory */
  word sop[160];                                            /* output */

  if ((f_read=fopen(argv[1],"rb")) == NULL) {
   fprintf(stderr,"%s: can't open %s\n",argv[0],argv[1]);
   exit(1);
  }
  if ((f_write=fopen(argv[2],"wb")) == NULL) {
   fprintf(stderr,"%s: can't open %s\n",argv[0],argv[2]);
   exit(1);
   fclose(f_read);
  }

  /* initalization of arrays */
  for (i = 1; i < 9; i++)
    last_LARpp[i] = 0;

  while(read_code(LARc, Nc, bc, Mc, xmaxc, xMc)) {
    decode_block(LARc, Nc, bc, Mc, xmaxc, xMc, last_LARpp, sop);
    write_data(sop);
  }

  fclose(f_read);
  fclose(f_write);
}

/*****************************************************************************
              decode_block -- decodes one block of 160 samples
*****************************************************************************/
void decode_block (word LARc[9], word Nc[4], word bc[4], word Mc[4],
                   word xmaxc[4], word xMc[4][13], word last_LARpp[9],
                   word sop[160])
{
  int i, j;

  word exp, mant;
  word xMp[13], ep[40], rp[9], drp[160];
  word LARp[9], LARpp[9], so[160], s[160];

  static int k_start[4] = {0,  13, 27,  40};
  static int k_end[4]   = {12, 26, 39, 159};

  /* LONG TERM DECODING SECTION ** once for each sub-frame */
  for (i = 0; i < 4; i++) {
    rpe_decode(xmaxc[i], &exp, &mant);
    apcm_iquant(exp, mant, xMc[i], xMp);            /* gsmenc.c */
    rpe_posit(Mc[i], xMp, ep);                      /* gsmenc.c */
    ltp_synth(bc[i], Nc[i], ep, drp+(i*40));
  }

  /* SHORT TERM DECODING SECTION */
  decode_lar(LARc, LARpp);                          /* gsmenc.c */
  for (i = 0; i < 4; i++) {
    inte_lar(k_start[i], last_LARpp, LARpp, LARp);  /* gsmenc.c */
    coeffs(LARp, rp);                               /* gsmenc.c */
    s_filter(k_start[i], k_end[i], rp, drp, s);
  }
  /* update the last_LARpp array */
  for (i = 1; i < 9; i++)
    last_LARpp[i] = LARpp[i];

  /* POSTPROCESSING */
  deemph(s, so);
  upscale(so, sop);
  truncate(sop);
}

/*****************************************************************************
                        RPE DECODING SECTION
```

```
*****************************************************************************/

/*****************************************************************************
    rpe_decode -- get mant and exp of xmax, modified from sec. 4.2.15
                  according to sec. 4.3.1
*****************************************************************************/
void rpe_decode (word xmaxc, word *exp_p, word *mant_p)
{
  int i, itest;
  word exp, mant;

  /* compute exponent and mantissa decoded version of xmaxc */
  exp = 0;
  if (xmaxc > 15)
    exp = sub(xmaxc >> 3, 1);
  mant = sub(xmaxc, exp << 3);

  /* normalize mantissa 0 <= mant <= 7 */
  if (mant == 0) {
    exp = -4;
    mant = 15;
  }
  else {
    itest = 0;
    for (i = 0; i < 3; i++) {
      if (mant > 7)
        itest = 1;
      if (itest == 0) {
        mant = add(mant << 1, 1);
        exp = sub(exp, 1);
      }
    }
  }
  *mant_p = sub(mant, 8);
  *exp_p = exp;
}

/*****************************************************************************
       apcm_iquant -- APCM inverse quantization, sec. 4.2.16
*****************************************************************************/
void apcm_iquant (word exp, word mant, word xMc[13], word xMp[13])
{
  int i;
  word temp, temp1, temp2, temp3;

  temp1 = FAC[mant];
  temp2 = sub(6, exp);
  temp3 = (sub(temp2, 1) >= 0) ?
          (1 << sub(temp2, 1)) : (1 >> -sub(temp2, 1));

  for (i = 0; i < 13; i++) {
    temp = sub(xMc[i] << 1, 7);
    temp = temp << 12;
    temp = mult_r(temp1, temp);
    temp = add(temp, temp3);
    xMp[i] = (temp2 >= 0) ? (temp >> temp2) : (temp << -temp2);
  }
}

/*****************************************************************************
              rpe_posit -- RPE grig positioning, sec. 4.2.17
*****************************************************************************/
void rpe_posit (word Mc, word xMp[13], word ep[40])
{
  int i, k;

  for (k = 0; k < 40; k++) {
    ep[k] = 0;
  }

  for (i = 0; i < 13; i++) {
    ep[Mc + (3*i)] = xMp[i];
  }
}


/*****************************************************************************
                          LTP SYNTHESIS SECTION
*****************************************************************************/

/*****************************************************************************
       ltp_synth -- Long term synthesis filtering, sec. 4.3.2
*****************************************************************************/
void ltp_synth (word bcr, word Ncr, word erp[40], word drp2[40])
{
  int k;
  static word nrp = 40;
  static word drp1[121];
  static int drp_init = 0;
  word Nr, brp, drpp;

  /* Possibly initialize the array drp1 */
  if (!drp_init) {
    drp_init = 1;
    for (k = 0; k < 121; k++) {
      drp1[k] = 0;
    }
  }

  /* Check the limits of Nr */
  Nr = Ncr;
  if (Nr < 40)
```

```
    Nr = nrp;
  if (Nr > 120)
    Nr = nrp;
  nrp = Nr;

  /* Decoding of the LTP gain bcr */
  brp = QLB[bcr];

  /* Computation of the reconstructed short term residual signal drp[0..39] */
  for (k = 0; k < 40; k++) {
    drpp = mult_r(brp, drp1[Nr-k]);
    drp2[k] = add(erp[k], drpp);
  }

  /* Update the reconstructed short term residual signal drp1[1..120] */
  for (k = 0; k < 80; k++) {
    drp1[120-k] = drp1[80-k];
  }
  for (k = 0; k < 40; k++) {
    drp1[40-k] = drp2[k];
  }
}

/*************************************************************************
                    SHORT TERM SYNTHESIS SECTION
*************************************************************************/

/*************************************************************************
decode_lar -- Decoding of the coded Log. Area Ratios, sec. 4.2.8
*************************************************************************/
void decode_lar (word LARc[9], word LARpp[9])
{
  int i;
  word temp;

  for (i = 1; i < 9; i++) {
    temp = add(LARc[i], MIC[i]) << 10;
    temp = sub(temp, B[i] << 1);
    temp = mult_r(INVA[i], temp);
    LARpp[i] = add(temp, temp);
  }
}

/*************************************************************************
       inte_lar -- Interpolation of the LARpp[1..8] to get LARp[1..8],
       sec. 4.2.9.1
*************************************************************************/
void inte_lar (int k_start, word last_LARpp[9], word LARpp[9], word LARp[9])
{
  int i;

  switch(k_start) {
  case 0:
    for (i = 1; i < 9; i++) {
      LARp[i] = add(last_LARpp[i] >> 2, LARpp[i] >> 2);
      LARp[i] = add(LARp[i], last_LARpp[i] >> 1);
    }
    break;

  case 13:
    for (i = 1; i < 9; i++) {
      LARp[i] = add(last_LARpp[i] >> 1, LARpp[i] >> 1);
    }
    break;

  case 27:
    for (i = 1; i < 9; i++) {
      LARp[i] = add(last_LARpp[i] >> 2, LARpp[i] >> 2);
      LARp[i] = add(LARp[i], LARpp[i] >> 1);
    }
    break;

  case 40:
    for (i = 1; i < 9; i++) {
      LARp[i] = LARpp[i];
    }
    break;

  default:
    exit(1);
  }
}

/*************************************************************************
coeffs -- Computation of the rp[1..8] from the interpolated LARp[1..8]
sec. 4.2.9.2
*************************************************************************/
void coeffs (word LARp[9], word rp[9])
{
  int i;
  word temp;

  for (i = 1; i < 9; i++) {
    temp = Abs(LARp[i]);
    if (temp < 11059)
      temp = temp << 1;
    else if (temp < 20070)
      temp = add(temp, 11059);
    else
      temp = add(temp >> 2, 26112);

    rp[i] = temp;
```

```
    if (LARp[i] < 0)
      rp[i] = sub(0, rp[i]);
  }
}


/*************************************************************************
       s_filter -- Short term synthesis filtering section, sec. 4.3.4
*************************************************************************/
void s_filter (int k_start, int k_end, word rrp[9], word drp[160],
               word sr[160])
{
  int i, k;
  word sri;
  static word v[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};

  for (k = k_start; k < k_end+1; k++) {
    sri = drp[k];
    for (i = 1; i < 9; i++) {
      sri = sub(sri, mult_r(rrp[9-i], v[8-i]));
      v[9-i] = add(v[8-i], mult_r(rrp[9-i], sri));
    }
    sr[k] = sri;
    v[0] = sri;
  }
}

/*************************************************************************
                        POSTPROCESSING SECTION
*************************************************************************/

/*************************************************************************
              deemph -- Deemphasis filtering, sec. 4.3.5
*************************************************************************/
void deemph (word sr[160], word sro[160])
{
  int k;
  word temp;
  static word msr = 0;

  for (k = 0; k < 160; k++) {
    temp = add(sr[k], mult_r(msr, 28180));
    msr = temp;
    sro[k] = temp;
  }
}

/*************************************************************************
              upscale -- Upscaling the output signal, sec. 4.3.6
*************************************************************************/
void upscale (word sro[160], word srop[160])
{
  int k;

  for (k = 0; k < 160; k++) {
    srop[k] = add(sro[k], sro[k]);
  }
}

/*************************************************************************
       truncate -- Truncating of the output variable, sec. 4.3.7
*************************************************************************/
void truncate (word srop[160])
{
  int k;

  for (k = 0; k < 160; k++) {
    srop[k] = srop[k] >> 3;
    srop[k] = srop[k] << 3;
  }
}

/*************************************************************************
                                    dataconv.c
*************************************************************************/

#include <stdio.h>
#include <ctype.h>
static FILE *fp;

void main (int argc, char *argv[])
{
  int i;
  int data1, data2;
  int size = 1 << 30;

  fp = fopen(argv[i] , "rb");

  for(i = 0; i < size; i++) {
    data1 = getc(fp);
    data2 = getc(fp);
    printf("0x%04x\n", data1 + (data2 << 8));
    if (data2 == EOF) break;
  }
}
```

Appendix G GEPARD Macros

```
=======================================
                LOOPS
=======================================

for (i = 0; i < x; i++) {
  ...
}

/* normal */
   ldc   #-x, a0;
1:
   sub   a0, ONES, a0;
   jn a0, 1b;
   nop

/* repeated */
   ldc   #-(x-1), a0;
1:
   jn a0, 1b;
   sub   a0, ONES, a0;

/* complex */
   ldc   #(x-1), a0;
1:
   sub   a0, ONES, a0;
   jn a0, 2b;
   nop
   ...
2:
   j  1b;
   nop

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

for (i = 0; i < x; i++) {
  for (i = 0; i < x; i++) {
    ...
  }
}

   ldc   -x, a0;
   ldc   #-y, a3;
   add   NULL, NULL, a2;
1:
   sub   a0, ONES, a0;
   sub   a3, a2, a1;
   sub   a2, ONES, a2;
2:
   sub   a1, ONES, a1;
   jn    a1, 2b;
   nop

   jn    a0, 1b;
   nop

=======================================
                IFS (>)
=======================================

if (x > 0) {
  ...
}

/* jump if x ≤ 0 */
   add   x, ONES, z;
   jn z, 1f;
   nop
   ...
1:

/* jump if x > 0 */
   sub   NULL, x, z;
   jn z, 1f;
   ...
   ...
1:

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

if (x > y) {
  ...
}

/* jump if x ≤ y */
   sub   x, y, z;
   add   z, ONES, z;
   jn z, 1f;
   nop
   ...
1:

/* jump if x > y */
   sub   y, x, z;
   jn    z, 1f;
   ...
   ...
1:

32 Bit:

/* jump if x ≤ y */
   sub   y, x, z;
   not   z, z;
   jn z, 1f;
   ...
   ...
1:

=======================================
                IFS (≥)
=======================================

if (x ≥ 0) {
  ...
}

/* jump if x < 0 */
   jn x, 1f;
   nop
   ...
1:

/* jump if x ≥ 0 */
   not   x, z;
   jn z, 1f;
   ...
   ...
1:

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

if (x ≥ y) {
  ...
}

/* jump if x < y */
   sub   x, y, z;
   jn z, 1f;
   nop
   ...
1:

/* jump if x ≥ y */
   sub   y, x, z;
   add   z, ONES, z;
   jn z, 1f;
   ...
   ...
1:

32 Bit:

/* jump if x ≥ y */
   sub   x, y, z;
   jn z, 1f;
   ...
   ...
1:

=======================================
                IFS (<)
=======================================

if (x < 0) {
  ...
}

/* jump if x ≥ 0 */
   not x, z;
   jn z, 1f;
   nop
   ...
1:

/* jump if x < 0 */
   jn z, 1f;
   ...
   ...
1:

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

if (x < y) {
  ...
}

/* jump if x ≥ y */
   sub   y, x, z;
   add   z, ONES, z;
   jn z, 1f;
```

```
   nop
   ...
1:

/* jump if x < y */
   sub   x, y, z;
   jn z, 1f;
   ...
   ...
1:

32 Bit:

/* jump if x ≥ y */
   sub   x, y, z;
   not   z, z;
   jn z, 1f;
   ...
   ...
1:
```

```
   nop
   ...
1:

/* jump if x ≤ y */
   sub   x, y, z;
   add   z, ONES, z;
   jn z, 1f;
   ...
   ...
1:

32 Bit:

/* jump if x ≤ y */
   sub   y, x, z;
   jn z, 1f;
   ...
   ...
1:
```

```
   not   z, z;
   jn    z, 1f;
   nop
   ...
1:

/* jump if x = y */
   sub   x, y, z;
   cmpz  z, z;
   jn    z, 1f;
   ...
   ...
1:
```

```
=======================================
                IFS (≤)
=======================================

if (x ≤ 0) {
  ...
}

/* jump if x > 0 */
   not x, z;
   sub   z, ONES, z;
   jn z, 1f;
   nop
   ...
1:

/* jump if x ≤ 0 */
   add   x, ONES, z;
   jn z, 1f;
   ...
   ...
1:

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

if (x ≤ y) {
  ...
}

/* jump of x > y */
   sub   y, x, z;
   jn z, 1f;
```

```
=======================================
                IFS (=)
=======================================

if (x = 0) {
  ...
}

/* jump if x ≠ 0 */
   cmpz  x, y;
   not   y, y;
   jn    y, 1f;
   nop
   ...
1:

/* jump if x = 0 */
   cmpz  x, z;
   jn    z, 1f;
   ..
   ..
1:

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

if (x = y) {
  ...
}

/* jump if x ≠ y */
   sub   x, y, z;
   cmpz  z, z;
```

```
=======================================
              IF...ELSE
=======================================

if (x = 0) {
   ...
}
else {
   ...
}

/* more than 1 step */
   cmpz  x, y;
   not      y, y;
   jn    y, 1f;
   nop
   ...
   j     2f;
   nop
1:
   ...
2:

/* 1 step */
   cmpz  x, z;
   jn z, 1f;
   ...
   ...
1:

=======================================
```

```
=======================================
            IF...ELSEIF
=======================================

if (x = 0) {
  ...
}
else {
  if (x < y) {
    ...
  }
}

/* more than 1 step */
   cmpz  x, z;
   not      z, z;
   jn    z, 1f;
   nop
   ...
   j     2f;
   nop
1:
   sub      y, x, z;
   add      z, ONES, z;
   jn    z, 2f;
   nop
   ...
2:

/* 1 step */
   cmpz  x, z */
   jn z, 1f;
   ...
   sub   x, y, z;
   jn z, 1f;
   ...
   /* change back */
   ...
1:

=======================================
```

```
=======================================
                STACK
=======================================

   ldc   #IN, i2;
   ldc   #STORE, i4;
   ldx   (i4), i2;
   ldx   (i2)+1, a0;
   ...
   stx   a0, (i2)+1;
   ldx   (i2)-1, NULL;
   stx   i2, (i4);

=======================================
                MAX
=======================================

max = 0;
for (i = 0; i < x; i++) {
   temp = abs(y)
   if (temp > max) {
      max = temp
   }
}

   ldc   #-x, a0;
   add   NULL, NULL, max;
1:
   sub   a0, ONES<, a0;
   abs   y, y;
   sub   y, max, z;
   add   z, ONES, z;
   jn    z, 2f;
   nop
   add   y, NULL, max;
2:
   jn    a0, 1b;
   nop

=======================================
               SHIFTS
=======================================

x << 1

   lsl   x, y;

=======================================
```

```
=======================================
               SHIFTS
=======================================

.sect data_x   (<<)
.word 1,    2,    4,    8
      16,   32,   64,   128
      256,  512,  1024, 2048
      4096, 8192, 16384,32768

.sect data_y   (>>)
.word 0,    32768,16384,8192
      4096, 2048, 1024, 512
      256,  128,  64,   32
      16,   8,    4,    2

=======================================
             SHIFTS (<<)
=======================================

   ldc   #SHIFT, i2;
   ldx   [SHIFT], i3;
   ldx   (i2)*, NULL;
   ldx   (i2), c;
   mul   c, x;        not   c, a3;
   jn    a3, 1f;
   mv    pl, x;
   sub   NULL, x, x;
1:

/* no << 15 */
32 Bit:

   ldc   #SHIFT, i2;
   ldx   [SHIFT], i3;
   ldx   (i2)*, NULL;
   ldx   (i2), c;
   mul   c, x;        not   c, a3;
   jn    a3, 1f;
   mv    pl, x;
   sub   y  ONES, y;
1:
   mul   c, y;        mv ph, z;
   mv    pl, y;
   add   y, z, y;

=======================================
```

```
=======================================
            SHIFTS (>>)
=======================================

   ldc   #SHIFT, i2;
   ldx   [SHIFT], i3;
   ldx   (i2)*, NULL;
   ldy   (i2), c;
   cmpz  c, a3;
   not   a3, a3;      mul   c, x;
   jn    a3, 1f;
   mv    ph, a1;
   not   x, a3;
   jn    a3, 1f;
   lsr   x, a1;
   or    c, a1, a1;
1:

32 Bit:
   ldc   #SHIFT, i2;
   ldx   [SHIFT], i3;
   ldx   (i2)*, NULL;
   ldy   (i2), c; not   c, a2;
   add   c, ONES, a3;
   jn    a3, 1f;
   mul   c, x;
   mv    ph, a1;
   sub   y, ONES< y;
   sty   y, [y];
1:
   mul   c, y;    sty   z, [z];
   jn    a3, 2f;
   ldx   [x], c;  add   NULL, NULL, a1;
   sub   NULL, ONES, a1;
2:
   mac   c, a1;   sub   NULL, ONES, a2;
   ldy   [z], c;
   mac   c, a2;
   mv    ph, z;   ldy   [y], c;
   mv    pl, x;   sty   z, [z];
   mul   c, a1;   ldy   [z], c;
   mac   c, a2;
   mv    pl,y;
3:

=======================================

x >> 1

   lsr   x, y;
   xor   x, y, x;
   not   x, x;
   jn    x, 1f;
   ldc   #0x8000, x;
   or    x, y, y;
1:

=======================================
                MAC
=======================================

   ldc   #-(COUNT-1), i2;
   ldc   #DATA1, i4;
   ldc   #DATA2, i6;
   ldx   (i2), i5;
   ldx   (i4)*, NULL;
   ldc   #1, i7;
   ldy   (i6), c;
   ldc   #1, i5;
   ldx   (i4)*, d;
   sub   a0, ONES, a0;  mul   c, d; ldy
(i6), c; ldx   (i4)*, d;
1:
   sub   a0, ONES, a0;  mac   c, d; ldy
(i6), c; ldx   (i4)*, d;
   sub   a0, ONES, a0;  mac   c, d; ldy
(i6), c; ldx   (i4)*, d;
   sub   a0, ONES, a0;  mac   c, d; ldy
(i6), c; ldx   (i4)*, d;
   jn a0, 1b;
   sub   a0, ONES, a0;  mac   c, d; ldy
(i6), c; ldx   (i4)*, d;

   mv ph1, a1;
   mv pl, a2;
   lsl   a2, a2;

=======================================
                MUL
=======================================

   ldc   #COUNT, i2;
   ldx   (i2), a0;
   ldx   (i4), c;
   ldx   (i6)+1, d;
   mul   c, d;
   mv    pl, a3;      ldx   (i6)-1, d;
1:
   sub   a0, ONES, a0;  stx   a3,
(i6)+2;  mul   c, d;
   mv pl, a3;  ldx   (i6)-1, d;
   sub   a0, ONES, a0;  stx   a3,
(i6)+2;  mul   c, d;
   jn a0, 1b;
   mv pl, a3;  ldx   (i6)-1, d;

=======================================
                NORM
```

```
=======================================

temp = norm(yx)

/* all norm used are +ve */
   add   NULL, NULL, a3;
   cmpz  y, z;
   jn    z, 2f;
   ldc   #0x4000, c;
   sub   y, c, z;
   not   z, z;
   jn    z, 4f;
   nop
1:
   sub   y, c, z;
   sub   a3, ONES, a3;
   jn    z, 1b;
   lsl   y, y;
   j     4f;
   nop
2:
   jn    x, 4f;
   ldc   #15, a3;
3:
   sub   x, c, z;
   sub   a3, ONES, a3;
   jn    z, 3b;
   lsl   x, x;
4:

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

temp = norm(x << 16)

   add   NULL, NULL, a3;
   ldc   #0x4000, c;
1:
   sub   x, c, z;
   sub   a3, ONES, a3;
   jn    z, 1b;
   lsl   x, x;
2:

=======================================
```

```
=======================================
              NORM (32 Bit)
=======================================

temp = norm(yx)
for (i=0; i < a; i++) {
   b = l_to_s((e << temp) >> 16)
}

/* all norm used are +ve */
   add   NULL, NULL, a3;
   cmpz  y, z;
   jn    z, 2f;
   ldc   #0x4000, c;
   sub   y, c, z;
   not   z, z;
   jn    z, 4f;
   nop
1:
   sub   y, c, z;
   sub   a3, ONES, a3;
   jn    z, 1b;
   lsl   y, y;
   j     4f;
   nop
/* max   a3 = 14 */
2:
   ldc   #COUUNT, a0;
   ldc   #SHIFT, i2;
   ldc   #e, i6;
   stx   a3, [SHIFT];
   ldx   [SHIFT], i3;
   ldx   (i2)*, NULL;
   ldx   (i2), c;
3:
   ldx   (i6), x;       ldy   (i6), y;
   sub   a0, ONES, a0;  mul   c, x;
   xor   c, x, a3;
   not   a3, a3;
   jn    a3, 4f;
   mv    ph, z;
   sub   y, ONES, y;
4:
   mul   c, y;
   mv    pl, y;
   add   y, z, y;
   jn    a0, 3b;
   stx   y,    (i6)+1;
5:

=======================================
```

```
=======================================
              NORM (16 Bit)
=======================================

   /* no change */
   jn    x, 4f;
   ldc   #15, a3;
   add   NULL, NULL, a3;
1:
   sub   x, c, z;
   sub   a3, ONES, a3;
   jn    z, 1b;
   lsl   x, x;
2:
   ldc   #COUUNT, a0;
   ldc   #SHIFT, i2;
   ldc   #e, i6;
   stx   a3, [SHIFT];
   ldx   [SHIFT], i3;
   ldx   (i2)*, NULL;
   ldx   (i2), c;
   ldx   (i6)+1, x;
   mul   c, x;
   mv    pl, a3;        ldx   (i6)-1,
x;
3:
   sub   a0, ONES, a0;  stx   a3,
(i6)+2;  mul   c, x;
   mv    pl, a3;        ldx   (i6)-1,
x;
   sub   a0, ONES, a0;  stx   a3,
(i6)+2;  mul   c, x;
   mv    pl, a3;        ldx   (i6)-1,
x;
   sub   a0, ONES, a0;  stx   a3,
(i6)+2;  mul   c, x;
   mv    pl, a3;        ldx   (i6)-1,
x;
   sub   a0, ONES, a0;  stx   a3,
(i6)+2;  mul   c, x;
   jn    a0, 3b;
   mv    pl, a3;        ldx   (i6)-1,
x;
4:

=======================================
```

```
=======================================
          DIV (NUM, DENUM)
=======================================

s_div    num, denum, temp, temp, cycle,
res
.macro   s_div   u, v, w, x, y, z
         abs     u, w;
         jn      w, 5f;
         abs     v, x;
1:
         jn      x, 6f;
         ldc     #16, y;
2:
         xor     w, x, z;
         cmpz    z, z;
         jn      z, 7f;
         nop
3:
         add     y, ONES, y;
         jn      y, 8f;
         sub     w, x, w;
         jn      w, 4f;
         lsl     z, z;
         sub     z, ONES, z;
         j       3b;
         lsl     w, w;
4:
         add     w, x, w;
         j       3b;
         lsl     w, w;
5:
         j       1b;
         lsr     ONES, w;
6:
         j       2b;
         lsr     ONES, x;
7:
         lsr     ONES, z;
8:       .end

=======================================
          ADD(VAR1, VAR2)
=======================================

.macro   s_add   w, x, y, z
         xor     x, y, w;
         jn      w, 1f;
         add     x, y, z;
         not     x, x;
         xor     z, x, w;
         jn      w, 1f;
         nop
         jn      x, 1f;
```

```
        lsr     ONES, z;
        not     z, z;
1:      .end

=====================================
          SUB(VAR1, VAR2)
=====================================

.macro  s_sub   w, x, y, z
        xor     x, y, w;
        not     w, w;
        jn      w, 1f;
        sub     x, y, z;
        xor     z, y, w;
        jn      w, 1f;
        nop
        jn      y, 1f;
        lsr     ONES, z;
        not     z, z;
1:      .end

=====================================
         MUL_R(VAR1, VAR2)
=====================================

.macro  s_mur    w, x, y, z
        mul     x, y;
        xor     x, y, w;
        ldc     #0x4000, x;
        sub     NULL, ONES, z;
        mv      ph1, y;
        mac     x, z;
        cmpz    w, w;
        not     w, w;
        jn      w, 1f;
        mv      ph1, z;
        jn      w, 1f;
        not     y, w;
        jn      w, 1f;
        nop
        lsr     ONES, z;
1:      .end

=====================================
              ABS(VAR1)
=====================================

.macro  s_abs   w, x
        abs     w, x;
        not     x, w;
        jn      w, 1f;
        nop
        lsr     ONES, x;
1:      .end
```

```
=====================================
        L_SUB(L_VAR1, L_VAR2)
=====================================

.macro  l_sub   w, x, y, z
        ldc     #1, c;
        mul     c, w;
        jn      w, 3f;
        add     NULL, NULL, w;
1:
        jn      x, 4f;
        sub     NULL, x, x;
2:
        mac     c, x;
        mv      ph, x;
        add     w, x, x;
        sub     y, z, w;
        add     w, x, x;
        xor     y, z, w;
        not     w, w;
        jn      w, 5f;
        mv      pl, w;
        xor     x, z, z;
        jn      z, 5f;
        not     y, z;
        add     ONES, NULL, w;
        jn      z, 5f;
        lsr     ONES, x;
        add     NULL, NULL, w
        j       5f;
        not     x, x;
3:
        j       1b;
        sub     w, ONES, w;
4:
        j       2b;
        add     w, ONES, w;
5:      .end

=====================================
```

Appendix H GSM Full-Rate Speech Compression GEPARD Code

```
/* gsmoper.asm ver 3.1.1 Sept 1997 */

/*****************************************************************
        s_add           temp, u0, v0, w0
        s_sub           temp, u0, v0, w0
        s_mur           temp, u0, v0, w0
        s_abs           u0, w0
        s_div           num, denum, temp, temp, cycle, res
        l_sub           u0/w0, v0/w1, u1, v1
        l_nor           temp, u0, u1, w0
*****************************************************************/

/* add(var1, var2) */
/* performs the addition (var1 + var2) with overflow control and
saturation; the result is set at +32767 when overflow occurs or at
-32768 when underflow occurs */
.macro  s_add   w, x, y, z
        xor     x, y, w;

        /* jump if sign bits of operands different */
        jn      w, 1f;
        add     x, y, z;

        not     x, x;
        xor     z, x, w;

        /* jump if sign bits of operands and result same */
        jn      w, 1f;
        nop

        /* jump if operands pos */
        jn      x, 1f;
        lsr     ONES, z;
        not     z, z;

1:      .end

/* sub(var1, var2) */
/* performs the subtraction (var1 - var2) with overflow control and
saturation; the result is set at +32767 when overflow occurs or at
-32768 when underflow occurs */
.macro  s_sub   w, x, y, z
        xor     x, y, w;
        not     w, w;

        /* jump if sign bits of operands same */
        jn      w, 1f;
        sub     x, y, z;

        xor     z, y, w;

        /* jump if sign bits of operands and result different */
        jn      w, 1f;
        nop

        /* jump if operands pos */
        jn      y, 1f;
        lsr     ONES, z;
        not     z, z;

1:      .end

/* mul_r(var1, var2) */
/* performs the multiplication (var1 * var2) with rounding and gives
16 bits results which is scaled, ie mult_r(var1, var2) = ((var1 * var2)
+ 16384) >> 15 and mult_r(-32768, -32768) = 32767 */
.macro  s_mur   w, x, y, z
        mul     x, y;           xor     x, y, w;
        ldc     #0x4000, x;
        sub     NULL, ONES, z;
        mv      ph1, y;
        mac     x, z;           cmpz    w, w;
        not     w, w;

        jn      w, 1f;
        mv      ph1, z;



        /* jump if sign bits of operands different */
        jn      w, 1f;

        not     y, w;

        /* jump if operands pos */
        jn      w, 1f;
        nop
        lsr     ONES, z;

1:      .end

/* abs(var1) */
/* absolute value of var1; abs(-32768) = 32767 */
.macro  s_abs   w, x
        abs     w, x;
        not     x, w;

        /* jump if operands pos */
        jn      w, 1f;
        nop
        lsr     ONES, x;

1:      .end

/* div(num, denum) */
/* produces a result which si the fractional integer division
of var1 by var2; var1 and var2 must be positive and var2 must
be greater ot equal to var1; the result is positive (leading
bit equal to 0) and truncated to 16 bits.  If var1 == var2 then
div(var1, var2) = 32767 */
.macro  s_div   u, v, w, x, y, z
        abs     u, w;
        jn      w, 5f;
        abs     v, x;

1:
        jn      x, 6f;

        /* 16 cycles */
        ldc     #16, y;

2:
        xor     w, x, z;
        cmpz    z, z;
        jn      z, 7f;
        nop

3:
        add     y, ONES, y;
        jn      y, 8f;
        sub     w, x, w;
        jn      w, 4f;
```

```
        lsl     z, z;
        sub     z, ONES, z;
        j       3b;
        lsl     w, w;

4:
        add     w, x, w;
        j       3b;
        lsl     w, w;

5:
        j       1b;
        lsr     ONES, w;

6:
        j       2b;
        lsr     ONES, x;

7:
        lsr     ONES, z;

8:      .end


/* L_sub(L_var1, L_var2) */
/* 32 bits subtraction of two 32 bits variables (L-var1 -
L_var2) with overflow control and saturation; the result
is set at +2147483647 when overflow occurs and at
-2147483648 when underflow occurs */
.macro  l_sub   w, x, y, z
        ldc     #1, c;
        mul     c, w;
        jn      w, 3f;
        add     NULL, NULL, w;

1:
        jn      x, 4f;
        sub     NULL, x, x;

2:
        mac     c, x;
        mv      ph, x;
        add     w, x, x;
        sub     y, z, w;
        add     w, x, x;
        xor     y, z, w;
        not     w, w;
        jn      w, 5f;
        mv      pl, w;

        xor     x, z, z;
        jn      z, 5f;
        not     y, z;

        add     ONES, NULL, w;
        jn      z, 5f;
        lsr     ONES, x;

        add     NULL, NULL, w
        j       5f;
        not     x, x;

3:
        j       1b;
        sub     w, ONES, w;

4:
        j       2b;
        add     w, ONES, w;

5:      .end
```

```
/* norm(L_var1) */
/* produces the number of left shifts needed to normalise
the 32 bits variable L_var1 for positive values on the
interval with minimum of 1073741824 and maximum of 2147483647
and for negative values on the interval of -2147483648 and
interval maximum of -1073741824; in order to notmalise the
result, the following operation must be done: L_norm_var =
L_var1 << norm(L_var1) */
.macro  l_nor   w, x, y, z
        cmpz    y, w;
        jn      w, 3f;
        ldc     #0x4000, w;
        not     y, z;
        cmpz    z, z;

        /* jump if y = 0 */
        jn      z, 2f;
        abs     y, y;
        add     NULL, ONES, z;

1:
        sub     y, w, x;
        sub     z, ONES, z;

        /* jump if y neg */
        jn      x, 1b;
        lsl     y, y;
        j       5f;
        nop

2:
        cmpz    x, z;
        jn      z, 1b;
        sub     NULL, x, x;

3:
        jn      x, 5f;
        ldc     #15, z;

4:
        sub     x, w, y;
        sub     z, ONES, z;
        jn      y, 4b;
        lsl     x, x;

5:      .end


/* gsmtable.asm ver 2.0 Sept 1997*/

/*************************************************************************
        MAIN
*************************************************************************/
        .sect   data_x
        .org    0x0000

COUNT2: .word   -16
COUNT:  .word   -4
LINK:   .word   0
STORE:  .word   0
STORE2: .word   0

        .sect   data_y
        .org    0x0000
        .word   -16
        .word   -4
        .word   0
        .word   0
        .word   0

/*************************************************************************
        PRE
```

```
*****************************************************************************/
        .sect   data_x
        .org    0x0010

        .sect   data_y
        .org    0x0010

/****** SCALING ************************************************************/

/****** OFFSET *************************************************************/
        .sect   data_x
/*              0x8000, 32735,  1,      32735                           */
OFF:    .word   0x8000, 32735,  1,      32735

        .sect   data_y
/*              z1,     lsp,    16384,  msp                             */
        .word   0,      0,      16384,  0

/****** PREEMP *************************************************************/
        .sect   data_x
/*              28180,  1                                               */
PRE:    .word   0x6e14, 1

        .sect   data_y
/*              0,      0x4000                                          */
        .word   0,      0x4000

/*****************************************************************************
        LPC
*****************************************************************************/

/****** AUTOCORR ***********************************************************/
        .sect   data_x
/*              4,      scal,   temp,   0       */
AUT:    .word   0x0004, 0x0000, 0x0000, 0x0000
        .sect   data_y
/*              -160,   16384,  -9,     1       */
        .word   0xff60, 0x4000, 0xfff7, 0x0001

/****** SCHUR **************************************************************/
        .sect   data_x
/*              temp,   store   */
SCH:    .word   0x0000, 0x0000

        .sect   data_y
/*              -9,     empty   */
        .word   0xfff7, 0x0000

/****** LARS ***************************************************************/
        .sect   data_x
LAR:    .word   22118,  31130

        .sect   data_y
        .word   11059,  26112

/****** CODE_LAR ***********************************************************/
        .sect   data_x
COD:    .word   256

        .sect   data_y
        .word   128

/*****************************************************************************
        STA
*****************************************************************************/

/****** DECODE_LAR *********************************************************/

/****** INTE_LAR ***********************************************************/

/****** COEFFS *************************************************************/
        .sect   data_x
```



```
COE:    .word   11059,  20070

        .sect   data_y
        .word   11059,  26112

/****** I_FILTER ***********************************************************/
        .sect   data_x
/*              temp1,  sav,    1       */
IFI:    .word   0,      0,      1

        .sect   data_y
/*              -8,     temp2,  0x4000  */
        .word   -8,     0,      0x4000

/*****************************************************************************
        LTP
*****************************************************************************/

/****** LTP_CALC ***********************************************************/
        .sect   data_x
/*              -40,    scal,   1,      -81,    l_max,  l_power */
CAL:    .word   -40,    0,      1,      -81,    0,      0

        .sect   data_y
/*              Nc,     bc,     80,     QLB,    l_max,  l_power */
        .word   0,      0,      0,      0,      0,      0

/****** LTP_FILTER *********************************************************/

/*****************************************************************************
        RPE
*****************************************************************************/

/****** W_FILTER ***********************************************************/
        .sect   data_x
/*              count,  0,      0x2000, 4       */
RPF:    .word   -40,    0,      0x1000, 4

        .sect   data_y
/*              -11,    1,      1,      0       */
        .word   -11,    1,      1,      0

/****** RPE_SELECT *********************************************************/
        .sect   data_x
/*              -4,     EM,     0,      temp,   3       */
SEL:    .word   -4,     0,      0,      0,      3

        .sect   data_y
/*              -12,    EM,     Mc,     temp,   3       */
        .word   -12,    0,      0,      0,      3

/****** APCM_QUANT *********************************************************/
        .sect   data_x
/*              -13,    exp,    temp2,  temp3   */
APC:    .word   -13,    0,      0,      0

        .sect   data_y
/*              xmaxc,  mant,   temp1,  temp4   */
        .word   0,      0,      0,      4096

        .sect   data_x
RAN:    .word   -6,     5,      8,      -3,     16

        .sect   data_y
        .word   128,    15,     7,      6,      4

/****** APCM_IQUANT ********************************************************/

/****** RPE_POSIT **********************************************************/

/****** UPSCALE_DP *********************************************************/
        .sect   data_x
```

```
UPD:    .word   -80,    -40

        .sect   data_y
        .word   40,     41

/*************************************************************************
        TABLES
*************************************************************************/
        .sect   data_x
        .org    0x0050

        .sect   data_y
        .org    0x0050

        .sect   data_x
SHIFT:  .word   0x0001, 0x0002, 0x0004, 0x0008
        .word   0x0010, 0x0020, 0x0040, 0x0080
        .word   0x0100, 0x0200, 0x0400, 0x0800
        .word   0x1000, 0x2000, 0x4000, 0x8000

        .sect   data_y
        .word   0x0000, 0x8000, 0x4000, 0x2000
        .word   0x1000, 0x0800, 0x0400, 0x0200
        .word   0x0100, 0x0080, 0x0040, 0x0020
        .word   0x0010, 0x0008, 0x0004, 0x0002

        .sect   data_x
AI:     .word   20480,  20480,  20480,  20480
        .word   13964,  15360,  8534,   9036

        .sect   data_y
BI:     .word   0,      0,      2048,   -2560
        .word   94,     -1792,  -341,   -1144

        .sect   data_x
MICI:   .word   -32,    -32,    -16,    -16
        .word   -8,     -8,     -4,     -4

        .sect   data_y
MACI:   .word   31,     31,     15,     15
        .word   7,      7,      3,      3

        .sect   data_x
DL:     .word   1024,   1024,   1024,   1024
        .word   1024,   1024,   1024,   1024

        .sect   data_y
INVAI:  .word   13107,  13107,  13107,  13107
        .word   19223,  17476,  31454,  29708

        .sect   data_x
DLBI:   .word   6554,   16384,  26214,  32767

        .sect   data_y
QLBI:   .word   3277,   11469,  21299,  32767

        .sect   data_x
HI:
        .word   -134,   -374,   0,      2054,   5741,   8192
        .word   5741,   2054,   0,      -374,   -134,   0

        .sect   data_y
UI:     .word   0,      0,      0,      0,      0,      0
        .word   0,      0,      0,      0,      0,      0

        .sect   data_x
NRFACI: .word   29128,  26215,  23832,  21846
        .word   20165,  18725,  17476,  16384

        .sect   data_y
FACI:   .word   18431,  20479,  22527,  24575
        .word   26623,  28671,  30719,  32767
```

```
/*************************************************************************
        DATA
*************************************************************************/
        .sect   data_x
        .org    0x00a0
IN:
        .sect   data_y
        .org    0x00a0

        .sect   data_x
        .org    0x0150
L_ACF:
        .sect   data_y
        .org    0x0150

        .sect   data_x
        .org    0x0160
R9:

        .sect   data_y
        .org    0x0160

        .sect   data_x
        .org    0x0170
LAR1:   .word   0,      0,      0,      0
        .word   0,      0,      0,      0

        .sect   data_y
        .org    0x0170
        .word   0,      0,      0,      0
        .word   0,      0,      0,      0

        .sect   data_x
        .org    0x0178
LAR2:   .word   0,      0,      0,      0
        .word   0,      0,      0,      0

        .sect   data_y
        .org    0x0178
        .word   0,      0,      0,      0
        .word   0,      0,      0,      0

        .sect   data_x
        .org    0x0180
LAR3:   .word   0,      0,      0,      0
        .word   0,      0,      0,      0

        .sect   data_y
        .org    0x0180
        .word   0,      0,      0,      0
        .word   0,      0,      0,      0

        .sect   data_x
        .org    0x0188
LAR4:   .word   0,      0,      0,      0
        .word   0,      0,      0,      0

        .sect   data_y
        .org    0x0188
        .word   0,      0,      0,      0
        .word   0,      0,      0,      0

        .sect   data_x
        .org    0x019b
WTK:
        .sect   data_y
        .org    0x019b

        .sect   data_x
        .org    0x01a0
CURRENT:
```

```
        .sect   data_y
        .org    0x01a0

        .sect   data_x
        .org    0x01d0
PAST:
        .sect   data_y
        .org    0x01d0

        .sect   data_x
        .org    0x0250
XMI:
        .sect   data_y
        .org    0x0250

        .sect   data_x
        .org    0x0260
OUT:
        .org    0x0268
SEQ1:
        .org    0x0279
SEQ2:
        .org    0x028a
SEQ3:
        .org    0x0296
SEQ4:
        .sect   data_y
        .org    0x0260

        .sect   data_x
        .org    0x02b0
TEMP:
        .sect   data_y
        .org    0x02b0

        .sect   data_x
        .org    0x0500
TEST:
        .sect   data_y
        .org    0x0500


/* gsmenc.asm ver 4.5.2 Dec 1997*/

.include        "oper.asm"
.include        "tables.asm"

          .sect   code
          .org    0x0000

          j       INIT;
          nop

          .sect   code
          .org    0x0064
MAIN:
        ldc     #OUT, i1;

          j       READ_DATA;
          ldc     #@+1, lr0;

          j       PREPROC;
          ldc     #@+1, lr0;

        j       LPC;
          ldc     #@+1, lr0;

          j       STA;
          ldc     #@+1, lr0;

          ldc     #IFI, i2;
          ldc     #-1, i3;

          ldc     #IN, i6;

          ldc     #LAR1, i4;
          ldc     #-13, a0;
          j       I_FILTER;
          ldc     #@+1, lr0;

          ldc     #LAR2, i4;
          ldc     #-14, a0;
          j       I_FILTER;
          ldc     #@+1, lr0;

          ldc     #LAR3, i4;
          ldc     #-13, a0;
          j       I_FILTER;
          ldc     #@+1, lr0;

          ldc     #LAR4, i4;
          ldc     #-120, a0;
          j       I_FILTER;
          ldc     #@+1, lr0;

        ldc     #IN, i4;
          ldc     #STORE, i3;
          stx     i4, (i3);
1:
          j       LTP;
          ldc     #@+1, lr0;

        j       RPE;
        ldc     #@+1, lr0;

        ldc     #COUNT, i2;
          ldy     (i2), a0;
          sub     a0, ONES, a0;
          jn      a0, 1b;
          sty     a0, (i2);
          ldc     #-4, a0;
          sty     a0, (i2);

        j       WRITE_CODE;
        ldc     #@+1, lr0;

          ldc     #COUNT2, i2;
          ldx     (i2), a0;
          sub     a0, ONES, a0;
          jn      a0, MAIN;
          stx     a0, (i2);

          j       EXIT;
          nop

          .sect   code
/* read input */
READ_DATA:
          ldc     #-160, a0;
          ldc     #IN, i6;
          ldc     #0x4800, i0;
1:
          sub     a0, ONES, a0;   ldx     (i0), a1;
          jn      a0, 1b;
          stx     a1, (i6)+1;

          jr
          nop

          .sect   code
/* preprocessing section */
PREPROC:

/* scaling (4.2.1) */
```

```
SCALING:
        ldc     #IN, i6;
        ldc     #-8, a3;
        ldc     #-160, a0;
        ldc     #0x8000, c;
        ldx     (i6)-1, a1;     add     NULL, NULL, a2;

SCALING_1:
        stx     a2, (i6)+2;     and     a1, a3, a2;
        not     a1, a1;
        jn      a1, SCALING_2;
        lsr     a2, a2;
        or      c, a2, a2

SCALING_2:
        jn      a0, SCALING_1;
        sub     a0, ONES, a0;   ldx     (i6)-1, a1;

/* offset compensation (4.2.2) */
OFFSET:
        ldc     #OFF, i2;
        ldc     #IN, i6;
        ldc     #-160, a0;

OFFSET_1:
      /* non-recursive part */
        sub     a0, ONES, a0;   ldx     (i6), a1;       ldy     (i2), d;
        sty     a1, (i2)+1;     sub     a1, d, a2;      /* z1 */

      /* multiplication */
        ldx     (i2)+1, c;      ldy     (i2)+1, d;      /* 32735, lsp */
        mul     c, d;           ldx     (i2), c;        ldy     (i2), d;
        mac     c, d;           sty     a2, (i6);       /* 1, 16384 */
        mv      ph1, a1;        ldx     (i2)+1, NULL;
        ldx     (i2)-1, c;      ldy     (i2)-1, d;      /* 32735, msp */
        mul     c, d;           ldx     (i2)-2, c;      /* 1 */
        mac     c, a1;          ldy     (i6), a3;

      /* execution of a 31 by 16 multiplication */
        sub     NULL, a3, a3;   ldx     (i2)+2, c;      /* 0x8000 */
        mac     c, a3;          ldx     (i2), c;        ldy     (i2), d;
        mv      pl, a1;         ldx     (i2)+1, NULL;   /* 1, 16384 */
        mv      ph1, a2;
        lsl     a1, a1;                                 /* msp */
        mac     c, d;           lsr     a1, a1;         sty     a2, (i2)-2;
        mv      ph1, a2;        sty     a1, (i2)-1;     /* lsp */
        jn      a0, OFFSET_1;
        stx     a2, (i6)+1;

/* pre-emphasis (4.2.3) */
PREEMP:
        ldc     #PRE, i2;
        ldc     #IN, i6;
        ldc     #-160, a0;
        ldc     #1, i3;

PREEMP_1:
        sub     a0, ONES, a0;   ldy     (i2)*, c;       ldx     (i2)*, d;
        mul     c, d;           ldx     (i2)*, c;       ldy     (i2)*, d;
        mac     c, d;           ldx     (i6), a2;
        mv      ph1, a1;        ldx     (i2)-2, NULL;
        sub     a2, a1, a1;     sty     a2, (i2);
        jn      a0, PREEMP_1;
        stx     a1, (i6)+1;     sty     a1, (i6)+1;

PREEMP_2:
        jr
        nop

        .sect   code
/* LPC analysis section */
LPC:

/* autocorrelation (4.2.4) */
AUTOCORR:
        ldc     #AUT, i2;
        ldc     #IN, i6;
        ldx     (i6)+1, a1;     add     NULL, NULL, a3;
      ldy     (i2), a0;       abs     a1, a2;         /* -160 */

/* dynamic scaling of the array s[1..159] */
/* search for the maximum */
AUTOCORR_1:
        sub     a2, a3, a1;
        add     a1, ONES, a1;
        jn      a1, AUTOCORR_1_1;
        sub     a0, ONES, a0;   ldx     (i6)+1, a1;
        add     NULL, a2, a3;

AUTOCORR_1_1:
        jn      a0, AUTOCORR_1;
      abs     a1, a2;

/* computation of the scaling factor */
AUTOCORR_2:
        cmpz    a3, a2;
        not     a2, a2;
        jn      a2, AUTOCORR_2_1;
        ldx     (i2)+1, d;      /* 4 */
        j       AUTOCORR_4;
        stx     a2, (i2)+1;     /* scal */

AUTOCORR_2_1:
        l_nor   a0, a1, a3, a2
        sub     d, a2, a2;
        stx     a2, (i2);       /* scal */

/* scaling of the array s[0..159] */
AUTOCORR_3:
        ldc     #SHIFT, i4;
      ldc     #IN, i6;
      add     a2, ONES, a0;   ldy     (i2)+1, a2;     /* 16384 */
      jn      a0, AUTOCORR_4;                         /* jump if scal = 0 */
        stx     a0, (i2);       /* temp */

        ldx     (i2)-2, i5;     cmpz    a0, a3;         /* temp */
      jn      a3, AUTOCORR_3_1;                       /* jump if scal = 1 */
        ldx     (i4)*, NULL;

      ldy     (i4), c;
      mul     c, a2;          not     c, a3;
      jn      a3, AUTOCORR_3_1;                       /* jump of scal != 2 */
      mv      ph, a2;
      sub     NULL, a2, a2;

AUTOCORR_3_1:
        ldy     (i2)+2, a0;     /* -160 */
        stx     a2, (i2);       sub     a0, ONES, a0;   /* temp */
        ldc     #1, i7;

AUTOCORR_3_2:
        ldx     (i2), a2;       ldy     (i6), c;        /* temp */
        s_mur   a1, c, a2, a3
        jn      a0, AUTOCORR_3_2;
        sub     a0, ONES, a0;   stx     a3, (i6)*;      sty     a3, (i6)*;

/* compute l_ACF[..] */
AUTOCORR_4:
        ldc     #L_ACF, i3;
        ldy     (i2)-2, a1;     /* -9 */
        ldy     (i2)+3, a2;     /* -160 */

AUTOCORR_4_1:
        ldc     #IN, i4;
```

```
        ldc     #IN, i6;

        ldx     (i2), a3;       sub     a2, ONES, a2;  /* 0 */
        ldy     (i2), i7;       ldx     (i2), i5;      /* 1, 0 */
        ldx     (i4)*, NULL;    ldy     (i6)*, d;
        ldy     (i2), i5;       sub     a3, ONES, a3;  /* 1 */
        ldx     (i4)*, c;       sub     a1, ONES, a1;
        mul     c, d;           ldx     (i4)*, c;      ldy     (i6)*, d;

AUTOCORR_4_2:
        sub     a2, ONES, a2;   mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;
        sub     a2, ONES, a2;   mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;
        sub     a2, ONES, a2;   mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;
        jn      a2, AUTOCORR_4_2;
        sub     a2, ONES, a2;   mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;

        mv      ph1, a0;        stx     a3, (i2)-3;    /* 0 */
        mv      pl, a3;         sty     a0, (i3);
        lsl     a3, a3;         ldy     (i2)+3, a2;    /* -160 */
        jn      a1, AUTOCORR_4_1;
        stx     a3, (i3)+1;     sub     NULL, NULL, a0;
        stx     a0, (i2)-2;                            /* 0 */

/* rescaling of the array s[0..159] */
AUTOCORR_5:
        ldc     #SHIFT, i4;
        ldc     #IN, i6;
        ldc     #2, i7;
        ldx     (i2), a1;       /* scal */
        add     a1, ONES, a2;   ldx     (i2)-1, i5;    /* scal */
        jn      a2, AUTOCORR_6;
        ldx     (i4)*, NULL;

        ldx     (i4), c;        ldy     (i2), a0;      /* -160 */
        ldx     (i6)+1, d;
        mul     c, d;
        mv      pl, a2;         ldx     (i6)-1, d;

AUTOCORR_5_1:
        sub     a0, ONES, a0;   mul     c, d;   stx     a2, (i6)*;      sty     a2, (i6)*;
        mv      pl, a2;         ldx     (i6)-1, d;
        sub     a0, ONES, a0;   mul     c, d;   stx     a2, (i6)*;      sty     a2, (i6)*;
        mv      pl, a2;         ldx     (i6)-1, d;
        sub     a0, ONES, a0;   mul     c, d;   stx     a2, (i6)*;      sty     a2, (i6)*;
        mv      pl, a2;         ldx     (i6)-1, d;
        sub     a0, ONES, a0;   mul     c, d;   stx     a2, (i6)*;      sty     a2, (i6)*;
        jn      a0, AUTOCORR_5_1;
        mv      pl, a2;         ldx     (i6)-1, d;

AUTOCORR_6:

SCHUR:
        ldc     #SCH, i2;
        ldc     #R9, i3;
        ldc     #L_ACF, i4;
        ldc     #SHIFT, i6;
        ldy     (i4), a2;
        cmpz    a2, a3;
        not     a3, a3;
      jn      a3, SCHUR_3;
        ldx     (i4), a1;

        cmpz    a1, a3;
        not     a3, a3;         ldy     (i2), a0;      /* -9 */
      jn      a3, SCHUR_2;
        sub     a0, ONES, a0;

/* L_ACF[0] = 0 */
SCHUR_1:
        sub     a0, ONES, a0;   sty     a3, (i3)+1;
        sub     a0, ONES, a0;   sty     a3, (i3)+1;
        jn      a0, SCHUR_1;
        sub     a0, ONES, a0;   sty     a3, (i3)+1;
        j       SCHUR_5;
        nop

SCHUR_2:
        jn      a1, SCHUR_2_2;
        ldy     (i2), a0;       add     NULL, NULL, a3; /* -9 */

        l_nor   a0, a2, a1, a3
        stx     a3, (i2);       cmpz    a3, a0;        /* temp */

        jn      a0, SCHUR_2_3;
        ldx     (i2), i7;       ldy     (i2), a0;      /* -9, temp */

        ldy     (i6)*, NULL;    ldx     (i4)+1, d;
        ldx     (i6), c;
        ldc     #2, i5;
        mul     c, d;
        mv      pl, a1;         ldx     (i4)-1, d;

SCHUR_2_1:
        sub     a0, ONES, a0;   mul     c, d;   stx     a1, (i4)*;      sty     a1, (i4)*;
        mv      pl, a1;         ldx     (i4)-1, d;
        sub     a0, ONES, a0;   mul     c, d;   stx     a1, (i4)*;      sty     a1, (i4)*;
        jn      a0, SCHUR_2_1;
        mv      pl, a1;         ldx     (i4)-1, d;
        j       SCHUR_4;
        nop

SCHUR_2_2:
        sub     a0, ONES, a0;   ldx     (i4), a2;
        not     a2, a1;
        jn      a1, SCHUR_2_2_1;
        lsr     a2, a1;
        or      a3, a1, a1;

SCHUR_2_2_1:
        ldc     #0x8000, a3;
        jn      a0, SCHUR_2_2;
        stx     a1, (i4)+1;     sty     a1, (i4)+1;
        j       SCHUR_4;
        nop

SCHUR_2_3:
        sub     a0, ONES, a0;   ldx     (i4), a1;
        jn      a0, SCHUR_2_3;
        sty     a1, (i4)+1;
        j       SCHUR_4;
        nop

SCHUR_3:
        l_nor   a0, a1, a2, a3
        stx     a3, (i2);       cmpz    a3, a1;        /* temp */

        jn      a1, SCHUR_3_2;
        ldx     (i2), i7;       ldy     (i2), a0;      /* temp, -9 */

        ldx     (i6)*, NULL;
        ldx     (i6), c;        ldy     (i2), a0;      /* -9 */

SCHUR_3_1:
        sub     a0, ONES, a0;   ldx     (i4), d;       ldy     (i4), a2;
        not     d, a1;          mul     c, d;
        jn      a1, SCHUR_3_1_1;
        mv      ph, a1;
        sub     a2, ONES, a2;

SCHUR_3_1_1:
        mul     c, a2;
        mv      pl, a2;
        add     a1, a2, a1;
        jn      a0, SCHUR_3_1;
```

```
        stx     a1, (i4)+1;     sty     a1, (i4)+1;
        j       SCHUR_4;
        nop

SCHUR_3_2:
        sub     a0, ONES, a0;   ldy     (i4), a1;
        jn      a0, SCHUR_3_2;
        stx     a1, (i4)+1;

/* compute reflection coefficients */
SCHUR_4:
      ldc     #R9, i3;
        ldy     (i2)+1, a0;     /* -9 */
        sub     a0, ONES, a0;
        stx     a0, (i2);       /* store */

SCHUR_4_1:
        ldc     #L_ACF, i6;

        ldx     (i6)+1, d;
        ldx     (i6)-1, c;

        abs     c, a2;
        sub     d, a2, a3;
        jn      a3, SCHUR_1;
        add     NULL, NULL, a3; ldx     (i2), a0;       /* store */

        s_div   c, d, a0, a1, a2, a3
        jn      c, SCHUR_4_1_1;
      ldx     (i2), a0;       /* store */
        sub     NULL, a3, a3;

SCHUR_4_1_1:
      sty     a3, (i3);       sub     a0, ONES, a0
        not     a0, a1;         stx     a0, (i2);       /* store */
        jn      a1, SCHUR_5;
        nop

      /* the Schur recursion */
        s_mur   a1, c, a3, a2
        s_add   a1, a2, d, a3
        stx     a3, (i6)+1;

SCHUR_4_2:
      sub     a0, ONES, a0;   ldy     (i3), c;
        ldy     (i6)+1, a1;
        s_mur   a2, c, a1, a3
        ldx     (i6)-1, d;
        s_add   a2, a3, d, a1
      stx     a1, (i6)+1;     ldy     (i3), c;
        ldx     (i6)-1, a1;
        s_mur   a2, c, a1, a3
        ldy     (i6), d;
        s_add   a2, a3, d, a1
        jn      a0, SCHUR_4_2
        sty     a1, (i6)+1
        j       SCHUR_4_1;
      ldy     (i3)+1, NULL;   ldx     (i2), a0;       /* store */

SCHUR_5:

/* transformation of reflection coefficients to log-area ratios (4.2.6) */
LARS:
        ldc     #LAR, i2;
        ldc     #R9, i4;
        ldc     #-8, a0;
        ldy     (i4)+1, d;

LARS_1:
        abs     d, a2;          ldx     (i2)+1, a3;     /* 22118 */
        sub     a2, a3, a3;
        jn      a3, LARS_2;
        lsr     a2, a3;         ldx     (i2)-1, a1;     /* 31130 */

        sub     a2, a1, a1;     ldy     (i2)+1, a3;     /* 11059 */
        jn      a1, LARS_2;
        sub     a2, a3, a3;     ldy     (i2)-1, a1;     /* 26112 */

        sub     a2, a1, a3;
        lsl     a3, a3;
        lsl     a3, a3;

LARS_2:
        not     d, a2;
        jn      a2, LARS_3;
        sub     a0, ONES, a0;   ldy     (i4)-1, d;
        sub     NULL, a3, a3;

LARS_3:
        jn      a0, LARS_1;
        sty     a3, (i4)+2;

/* quantisation and coding of the log-area ratios (4.2.7) */
CODE_LAR:
        ldc     #AI, i2;
        ldc     #MICI, i4;
        ldc     #R9, i6;
        ldc     #COD, i7;
        ldc     #-8, a0;
        ldx     (i2), c;        ldy     (i6)+1, d;

CODE_LAR_1:
        sub     a0, ONES, a0;   mul     c, d;
        mv      ph1, a1;        ldy     (i2)+1, a2;
        add     a1, a2, a1;     ldx     (i7), a2;       /* 256 */
        add     a1, a2, a1;     ldy     (i7), c;        /* 512 */
        mul     c, a1;          ldy     (i4), a2;
        mv      ph, a1;

        sub     a2, a1, a3;     ldx     (i2), c;
        jn      a3, CODE_LAR_2;
        add     NULL, a2, a3;   ldx     (i4)+1, a2;

        sub     a1, a2, a3;
        jn      a3, CODE_LAR_2;
        add     NULL, a2, a3;
        add     NULL, a1, a3;

CODE_LAR_2:
        sub     a3, a2, a3;     ldy     (i6)-1, d;
        jn      a0, CODE_LAR_1;
      sty     a3, (i6)+2;     stx     a3, (i1)+1;

CODE_LAR_3:
        jr
        nop

        .sect   code
/* short term analysis filtering section */
STA:

/* decoding of the coded log-area ratios (4.2.8) */
DECODE_LAR:
        ldc     #MICI, i2;
        ldc     #BI, i3;
        ldc     #R9, i4;
        ldc     #INVAI, i5;
        ldc     #-8, a0;

DECODE_LAR_1:
        sub     a0, ONES, a0;   ldx     (i2), a1;       ldy     (i4), a2;
        add     a1, a2, a1;     ldx     (i5), c;        ldy     (i3), a2;
        mul     c, a1;          ldy     (i3)+1, NULL;   lsl     a2, a2;
        mv      pl, a1;         ldy     (i5)+1, c;
```

```
        sub     a1, a2, a2;     ldx     (i2)+1, NULL;
        s_mur   a1, c, a2, a3
        lsl     a3, a3;
        jn      a0, DECODE_LAR_1;
        sty     a3, (i4)+1;

/* interpolation of the LARpp[1..8] to get LARp[1..8] (4.2.9.1) */
INTE_LAR:
        ldc     #R9, i4;
        ldc     #LAR1, i6;
        ldc     #-8, a0;
        ldc     #0x4000, c;
        ldx     (i6)-1, NULL;   add     NULL, NULL, a3;

INTE_LAR_1:
        sub     a0, ONES, a0;   ldy     (i4), a1;
        mul     c, a1;          ldx     (i4), a2;
        mul     c, a2;          mv      ph, a1;
        mv      ph, a2;         stx     a3, (i6)+1;
        add     a1, a2, a3;     ldx     (i4)+1, a2;
        not     a2, a1;
      jn      a1, INTE_LAR_1_1;
        lsr     a2, a2;
        lsl     c, a1;
        or      a1, a2, a2;

INTE_LAR_1_1:
      jn      a0, INTE_LAR_1;
        add     a2, a3, a3;

        ldc     #R9, i4;
        ldc     #-8, a0;
        ldc     #0x8000, c;

INTE_LAR_2:
        sub     a0, ONES, a0;   ldy     (i4), a1;
        ldx     (i4)+1, a2;     sub     NULL, a1, a1;
        mul     c, a1;          sub     NULL, a2, a2;
        mul     c, a2;          mv      ph, a1;
        mv      ph, a2;         stx     a3, (i6)+1;
      jn      a0, INTE_LAR_2;
        add     a1, a2, a3;

        ldc     #R9, i4;
        ldc     #-8, a0;
        ldc     #0x4000, c;

INTE_LAR_3:
        sub     a0, ONES, a0;   ldy     (i4), a1;
        mul     c, a1;          ldx     (i4), a2;
        mul     c, a2;          mv      ph, a1;         stx     a1, (i4);
        mv      ph, a2;         stx     a3, (i6)+1;
        add     a1, a2, a3;     ldy     (i4)+1, a1;
        not     a1, a2;
      jn      a2, INTE_LAR_3_1;
        lsr     a1, a1;
        lsl     c, a2;
        or      a2, a1, a1;

INTE_LAR_3_1:
      jn      a0, INTE_LAR_3;
        add     a1, a3, a3;

        ldc     #R9, i4;
        ldc     #-8, a0;
        ldc     #1, i5;
        ldc     #1, i7;

INTE_LAR_4:
        sub     a0, ONES, a0;   ldy     (i4)*, a3;      stx     a3, (i6)*;
        sub     a0, ONES, a0;   ldy     (i4)*, a3;      stx     a3, (i6)*;
      jn      a0, INTE_LAR_4;
        sub     a0, ONES, a0;   ldy     (i4)*, a3;      stx     a3, (i6)*;

/* computation of the rp[1..8] from the interpolated LARpp[1..8] (4.2.9.2) */
COEFFS:
        ldc     #COE, i2;
        ldc     #LAR1, i4;
        ldc     #-32, a0;
        ldx     (i4)+1, d;

COEFFS_1:
        abs     d, a2;          ldx     (i2)+1, a3;     /* 11059 */
        sub     a2, a3, a3;
        jn      a3, COEFFS_2;
        lsl     a2, a3;         ldx     (i2)-1, a1;     /* 20070 */

        sub     a2, a1, a1;     ldy     (i2)+1, a3;     /* 11059 */
        jn      a1, COEFFS_2;
        add     a2, a3, a3;     ldy     (i2)-1, a1;     /* 26112 */

        lsr     a2, a3;
        lsr     a3, a3;
        add     a3, a1, a3;

COEFFS_2:
        not     d, a2;
        jn      a2, COEFFS_3;
        sub     a0, ONES, a0;   ldx     (i4)-1, d;
        sub     NULL, a3, a3;

COEFFS_3:
        jn      a0, COEFFS_1;
        sty     a3, (i4)+2;

COEFFS_4:
        jr
        nop

/* short term analysis filtering (4.2.10) */
I_FILTER:
        ldc     #UI, i7;
        sub     a0, ONES, a0;   ldx     (i6), a3;
      stx     a0, (i2)+1;     ldy     (i2)+1, a0;     /* temp1, -8 */
      ldx     (i6), a2;

I_FILTER_1:
      sub     a0, ONES, a0;   stx     a2, (i6);
      stx     a3, (i2);       ldy     (i4), c;        /* sav */
      s_mur   a1, c, a2, a3
      ldy     (i7), d;
      s_add   a1, a3, d, a2
      sty     a2, (i2);       /* temp2 */
      ldy     (i4)+1, c;
      ldy     (i7), a2;
      s_mur   a1, c, a2, a3
      ldx     (i6), d;
      s_add   a1, a3, d, a2
      ldx     (i2), a1;       ldy     (i2), a3;       /* sav, temp2 */
        jn      a0, I_FILTER_1;
      sty     a1, (i7)+1;     stx     a3, (i2);

      ldx     (i2)-1, NULL;
        ldx     (i2), a0;       ldy     (i2), i5;       /* temp1, -8 */
        jn      a0, I_FILTER;
        stx     a2, (i6)+1;     ldy     (i4)*, NULL;

I_FILTER_2:
        jr
        nop

        .sect   code
/* long term prediction section */
LTP:
```

```
/* calculation of the LTP parameters (4.2.11) */
LTP_CALC:
        ldc     #CAL, i2;
        ldc     #STORE, i3;
        ldc     #IN, i4;
        ldc     #CURRENT, i6;
        ldx     (i3), i4;

      /* search of the optimum scaling of d[0..39] */
        ldx     (i4)+1, a1;     add     NULL, NULL, a3;
      ldx     (i2)+1, a0;     abs     a1, a2;         /* -40 */

LTP_CALC_1_1:
      sub     a2, a3, a1;     stx     a1, (i6);       sty     a1, (i6);
      sub     a2, a3, a1;
      add     a1, ONES, a1;   ldx     (i6)+1, NULL;
        jn      a1, LTP_CALC_1_2;
        sub     a0, ONES, a0;   ldx     (i4)+1, a1;
        add     NULL, a2, a3;

LTP_CALC_1_2:
      jn      a0, LTP_CALC_1_1;
      abs     a1, a2
      ldx     (i4)-1, NULL;
        stx     i4, (i3);

/* computation of scaling factor */
LTP_CALC_2:
        cmpz    a3, a1;
        not     a1, a1;
      jn      a1, LTP_CALC_2_1;               /* jump if dmax != 0 */
        ldc     #6, d;
        j       LTP_CALC_4;
        stx     d, (i2)+3;      /* scal */

LTP_CALC_2_1:
        l_nor   a0, a1, a3, a2
      sub     a2, d, a0;
      jn      a0, LTP_CALC_2_2;               /* jump if temp < 6 */
      add     NULL, NULL, a1;
        j       LTP_CALC_4;
      stx     a1, (i2)+3;     /* scal */

LTP_CALC_2_2:
      sub     d, a2, a1;
      stx     a1, (i2);       sub     a0, ONES, a0;   /* scal */
      jn      a0, LTP_CALC_3;                 /* jump if scal != 1 */
        ldc     #CURRENT, i4;
        ldc     #CURRENT, i6;
        ldc     #1, i5;
        ldc     #1, i7;
        ldx     (i4)+1, a1;     ldy     (i2)-1, NULL;
        sub     a0, a1, a1;     ldx     (i2)+1, a3;     /* -40 */

        sub     a3, ONES, a3;   sty     a1, (i6)*;      ldx     (i4)*, a1;
        sub     a0, a1, a1;

LTP_CALC_2_3:
        sub     a3, ONES, a3;   sty     a1, (i6)*;      ldx     (i4)*, a1;
        sub     a0, a1, a1;
        sub     a3, ONES, a3;   sty     a1, (i6)*;      ldx     (i4)*, a1;
        sub     a0, a1, a1;
        sub     a3, ONES, a3;   sty     a1, (i6)*;      ldx     (i4)*, a1;
        jn      a3, LTP_CALC_2_3;
        sub     a0, a1, a1;

/* intialisation of a working array wt */
LTP_CALC_3:
        ldc     #SHIFT, i4;
        ldc     #CURRENT, i6;
        ldx     (i2)-1, i5;     /* scal */
        ldx     (i4)*, NULL;
        ldx     (i2)+4, a0;     ldy     (i4), c;        /* -40 */
        ldy     (i6)+1, d;
        mul     c, d;
        mv      ph, a1;         ldy     (i6)-1, d;
        sub     a0, ONES, a0;   sty     a1, (i6)+2;     mul     c, d;
        mv      ph, a1;         ldy     (i6)-1, d;

LTP_CALC_3_1:
        sub     a0, ONES, a0;   sty     a1, (i6)+2;     mul     c, d;
        mv      ph, a1;         ldy     (i6)-1, d;
        sub     a0, ONES, a0;   sty     a1, (i6)+2;     mul     c, d;
        mv      ph, a1;         ldy     (i6)-1, d;
        sub     a0, ONES, a0;   sty     a1, (i6)+2;     mul     c, d;
        jn      a0, LTP_CALC_3_1;
        mv      ph, a1;         ldy     (i6)-1, d;

/* search for the maximum cross-correlation and coding of LTP lag */
LTP_CALC_4:
        add     NULL, NULL, a3;
        stx     a3, (i2)-1;     sty     a3, (i2)-1;     /* l_max */
        ldx     (i2)-1, a1;     /* -81 */
        not     a1, a0;
      sty     a0, (i2)-2;     /* 80 */
        sty     a0, (i2)+2;     ldx     (i2)+2, a2;     /* Nc, -40 */

LTP_CALC_4_1:
        ldc     #PAST, i4;
        ldc     #CURRENT, i6;

        ldx     (i2), i7;       ldy     (i2), i5;       /* 1, 80 */
        ldx     (i4)*, NULL;    ldy     (i6)*, c;
        ldx     (i2), i5;       sub     a1, ONES, a1;
        ldx     (i4)*, d;       not     a1, a3;
        sub     a2, ONES, a2;   mul     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;

LTP_CALC_4_2:
        sub     a2, ONES, a2;   mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;
        sub     a2, ONES, a2;   mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;
        sub     a2, ONES, a2;   mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;
        jn      a2, LTP_CALC_4_2;
        sub     a2, ONES, a2;   mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;

        mv      ph1, a0;        sty     a3, (i2)+3;     /* 80 */
        mv      pl, a3;         sty     a0, (i2);       /* l_result */
        lsl     a3, a3;
        stx     a3, (i2)-1;     ldy     (i2)-1, a0;     /* l_result */
        ldx     (i2)+1, a2;     ldy     (i2)+1, d;      /* l_max */

        l_sub   a2, a3, d, a0
        not     a3, a3;
        jn      a3, LTP_CALC_4_3;
        ldx     (i2)-5, a0;     ldy     (i2)-5, a2;     /* l_result */
        sub     NULL, a1, a3;
        sty     a3, (i2)+4;     /* Nc */
        stx     a0, (i2)-4;     sty     a2, (i2)-4;     /* l_max */

LTP_CALC_4_3:
        jn      a1, LTP_CALC_4_1;
        ldx     (i2)+2, a2;     /* -40 */

        ldx     (i2)-2, NULL;
        ldc     #120, a0;
        ldy     (i2)+1, a1;     /* Nc */
        sub     a0, a1, a0;
      stx     a0, (i1)+1;

/* rescaling l_max */
LTP_CALC_5:
        ldc     #SHIFT, i4;
        ldc     #6, d;
      ldx     (i2), a0;       /* scal */
```

```
        sub     d, a0, a1;
        add     a1, ONES, a2;
        jn      a2, LTP_CALC_6;
      stx     a1, (i2);       /* scal */

      ldx     (i2)+3, i5;     /* scal */
      ldx     (i4)*, NULL;
      ldy     (i4), c;        ldx     (i2), a0;       /* l_max */
      ldy     (i2), a1;       not     a0, a2;         /* l_max */

      mul     c, a0;          not     c, a3;
      jn      a2, LTP_CALC_5_1;
      mv      ph, a2;
      sub     a1, ONES, a1;
      sty     a1, (i2);                               /* l_max */

LTP_CALC_5_1:
      mul     c, a1;          ldx     (i2), c;        /* l_max */
      jn      a3, LTP_CALC_5_2;
      stx     a2, (i2);       add     NULL, NULL, a1; /* l_max */
      sub     NULL, ONES, a1;

LTP_CALC_5_2:
      mac     c, a1;          sub     NULL, ONES, a2; ldx     (i2), c;        /* l_max */
      mac     c, a2;
      mv      ph, a0;         ldy     (i2), c;        /* l_max */
      mv      pl, a3;         sty     a0, (i2);       /* l_max */
      mul     c, a1;          ldy     (i2), c;        /* l_max */
      mac     c, a2;
      mv      pl, a1;
      stx     a3, (i2)-3;     sty     a1, (i2)-3;     /* l_max */

/* initialisation of working array wt[0..39] */
LTP_CALC_6:
        ldc     #PAST, i4;
        ldc     #TEMP, i6;
        ldc     #1, i7;
        ldc     #8192, c;
        ldx     (i2)-1, NULL;
        ldx     (i2), a0;       ldy     (i2), i5;       /* -40, Nc */
        ldx     (i4)*, NULL;
        ldx     (i4)+1, d;
        mul     c, d;
        mv      ph, a3;         ldx     (i4)+1, d;
        sub     a0, ONES, a0;   mul     c, d;   stx     a3, (i6)*;      sty     a3, (i6)*;
        mv      ph, a3;         ldx     (i4)+1, d;

LTP_CALC_6_1:
        sub     a0, ONES, a0;   mul     c, d;   stx     a3, (i6)*;      sty     a3, (i6)*;
        mv      ph, a3;         ldx     (i4)+1, d;
        sub     a0, ONES, a0;   mul     c, d;   stx     a3, (i6)*;      sty     a3, (i6)*;
        mv      ph, a3;         ldx     (i4)+1, d;
        sub     a0, ONES, a0;   mul     c, d;   stx     a3, (i6)*;      sty     a3, (i6)*;
        jn      a0, LTP_CALC_6_1;
        mv      ph, a3;         ldx     (i4)+1, d;

/* compute the power of the reconstructed short term residual signal dp[] */
LTP_CALC_7:
        ldc     #TEMP, i4;
        ldc     #1, i5;
        ldx     (i2)+5, a0;     /* -40 */
        sub     a0, ONES, a0;   ldx     (i4)*, c;       ldy     (i4)*, d;
        sub     a0, ONES, a0;   mul     c, d;           ldx     (i4)*, c;       ldy     (i4)*, d;

LTP_CALC_7_1:
        sub     a0, ONES, a0;   mac     c, d;           ldx     (i4)*, c;       ldy     (i4)*, d;
        sub     a0, ONES, a0;   mac     c, d;           ldx     (i4)*, c;       ldy     (i4)*, d;
        jn      a0, LTP_CALC_7_1;
        sub     a0, ONES, a0;   mac     c, d;           ldx     (i4)*, c;       ldy     (i4)*, d;

        mv      pl, a0;
        lsl     a0, a0;
        mv      ph1, a1;
        stx     a0, (i2)-1;     sty     a1, (i2)-1;     /* l_result */

/* normalisation of l_max and l_power */
LTP_CALC_8:
        ldx     (i2), a0;       sub     NULL, ONES, a1; /* l_max */
        ldy     (i2), a2;       add     NULL, NULL, a3; /* l_max */
        l_sub   a0, a1, a2, a3
          not     a1, a0;
          jn      a0, LTP_CALC_8_1;
          add     NULL, NULL, a0;

          ldx     (i2)-3, NULL;
          j       LTP_CALC_12;
        sty     a0, (i2);       stx     a0, (i1)+1;     /* bc */

LTP_CALC_8_1:
          ldx     (i2)+1, a0;     ldy     (i2)+1, a1;     /* l_max */
          ldx     (i2), a2;       ldy     (i2), a3;       /* l_result */
          l_sub   a0, a2, a1, a3
          jn      a2, LTP_CALC_9;
          ldc     #3, a0;

          ldx     (i2)-4, NULL;
          j       LTP_CALC_12;
        sty     a0, (i2);       stx     a0, (i1)+1;     /* bc */

LTP_CALC_9:
          ldx     (i2)-4, a2;     ldy     (i2)-4, a1;     /* l_result */
          cmpz    a1, a0;
          jn      a0, LTP_CALC_10;
          nop

          l_nor   a0, a2, a1, a3
          stx     a3, (i2)+4;     cmpz    a3, a1;         /* scal */

          jn      a1, LTP_CALC_11;
          ldc     #SHIFT, i4;

          ldx     (i2)-4, a0;     ldy     (i2)-4, a1;     /* l_power */
          ldx     (i2)+4, i5;     /* scal */
          ldx     (i4)*, NULL;
          ldx     (i4), c;        not     a0, a2;
          jn      a2, LTP_CALC_9_1;
          mul     c, a0;
          sub     a1, ONES, a1;

LTP_CALC_9_1:
          mv      ph, a0;         mul     c, a1;
          mv      pl, a1;
          add     a0, a1, a1;
          sty     a1, (i2)-1;     /* l_power */

          ldx     (i2), a2;       ldy     (i2), a3;       /* l_max */
          not     a2, a0;
          jn      a0, LTP_CALC_9_1_1;
          mul     c, a2;
          sub     a3, ONES, a3;

LTP_CALC_9_1_1:
          mv      ph, a2;         mul     c, a3;
          mv      pl, a3;
          add     a2, a3, a3;
          j       LTP_CALC_11;
          sty     a3, (i2)+1;     /* l_max */

LTP_CALC_10:
          jn      a2, LTP_CALC_10_1;
          nop

          l_nor   a0, a1, a2, a3
          stx     a3, (i2)+4;     cmpz    a3, a0;         /* scal */
```

```
        jn      a0, LTP_CALC_10_2;
        ldc     #SHIFT, i4;

        ldx     (i2)-4, a0;     /* l_power */
        ldx     (i2)+4, i5;     /* scal */
        ldx     (i4)*, NULL;
        ldx     (i4), c;

        mul     c, a0;
        mv      pl, a0;
        sty     a0, (i2)-1;     /* l_power */

        ldx     (i2), a1;       /* l_max */
        mul     c, a1;
        mv      pl, a1;
        j       LTP_CALC_11;
        sty     a1, (i2)+1;     /* l_max */

LTP_CALC_10_1:
        lsr     a2, a2;         ldx     (i2)+4, NULL;
        sty     a2, (i2)-1;     /* l_power */
        ldx     (i2), a3;       /* l_max */
        lsr     a3, a3;
        j       LTP_CALC_11;
        sty     a3, (i2)+1;     /* l_power */

LTP_CALC_10_2:
        ldx     (i2), a0;       /* l_power */
        sty     a0, (i2)-1;     /* l_power */
        ldx     (i2), a1;       /* l_max */
        sty     a1, (i2)+1;     /* l_max */

/* coding of LTP gain */
LTP_CALC_11:
        ldc     #DLBI, i4;
        ldc     #-2, a0;
        ldy     (i2)-1, c;      ldx     (i4)+1, d;      /* l_power */
        ldy     (i2)-3, a3;                             /* l_max */
        mul     c, d;           ldx     (i4)+1, d;

LTP_CALC_11_1:
        mv      ph1, a1;
        sub     a1, a3, a2;
        jn      a2, LTP_CALC_11_2;
        ldc     #2, a2;

        add     a0, a2, a2;
      j       LTP_CALC_12;
      sty     a2, (i2);       stx     a2, (i1)+1;   /* bc */

LTP_CALC_11_2:
        jn      a0, LTP_CALC_11_1;
        sub     a0, ONES, a0;   mul     c, d;   ldx     (i4)+1, d;

        add     a0, a2, a2;
      sty     a2, (i2);       stx     a2, (i1)+1;   /* bc */

LTP_CALC_12:

/* long term analysis filtering (4.2.12) */
/* decoding of the coded LTP gain */
LTP_FILTER:
        ldc     #CAL, i2;
        ldc     #QLBI, i4;
        ldc     #PAST, i6;
        ldy     (i2)+1, i7;     ldx     (i2)+1, a0;     /* Nc, -40 */
        ldx     (i6)*, NULL;    ldy     (i2)+2, i5;     /* bc */
        ldx     (i4)*, NULL;
        ldy     (i4), c;
        ldc     #CURRENT, i4;
        sty     c, (i2);        /* QLB */
```

```
/* calculating the array e[0..39] and the array dpp[0..39] */
LTP_FILTER_1:
      sub     a0, ONES, a0;   ldx     (i4), d;
        ldy     (i2), c;        ldx     (i6)+1, a1;
        s_mur   a2, c, a1, a3
      stx     a3, (i4);
      s_sub   a2, d, a3, a1
        jn      a0, LTP_FILTER_1;
      sty     a1, (i4)+1;

LTP_FILTER_2:
        jr
        nop

        .sect   code
/* RPE encoding section */
RPE:

        .sect   code
/* weighting filter (4.2.13) */
W_FILTER:
        ldc     #RPF, i2;
        ldc     #TEMP, i3;
        ldc     #WTK, i6;
        ldx     (i2)+1, a0;     ldy     (i2)+1, a1;     /* -40, -11 */

/* compute the signal x[0..39] */
W_FILTER_1:
        ldc     #HI, i4;
        ldc     #WTK, i6;

        ldx     (i2), a3;       sub     a0, ONES, a0;   /* 0 */
        ldx     (i2), i7;       ldy     (i2), i5;       /* 0, 1 */
        ldy     (i6)*, NULL;    sub     a3, ONES, a3;
        stx     a3, (i2)+1;     ldy     (i2)+1, i7;     /* 0, 1 */
        ldx     (i2)+1, c;      ldy     (i2)+1, d;      /* 0x1000, 1 */
        sub     a1, ONES, a1;   mul     c, d;           ldx     (i4)*, c;       ldy     (i6)*, d;
        sub     a1, ONES, a1;   mac     c, d;           ldx     (i4)*, c;       ldy     (i6)*, d;

W_FILTER_2:
        sub     a1, ONES, a1;   mac     c, d;           ldx     (i4)*, c;       ldy     (i6)*, d;
        sub     a1, ONES, a1;   mac     c, d;           ldx     (i4)*, c;       ldy     (i6)*, d;
        sub     a1, ONES, a1;   mac     c, d;           ldx     (i4)*, c;       ldy     (i6)*, d;
        sub     a1, ONES, a1;   mac     c, d;           ldx     (i4)*, c;       ldy     (i6)*, d;
        jn      a1, W_FILTER_2;
        sub     a1, ONES, a1;   mac     c, d;           ldx     (i4)*, c;       ldy     (i6)*, d;

        mv      ph1, a3;        ldx     (i2)-3, c;      /* 4 */
        mv      pl, a1;
        lsl     a1, a1;

      not     a1, a2;
        jn      a2, W_FILTER_3;
      mul     c, a1;          add     a3, NULL, a2;
        sub     a3, ONES, a3;

W_FILTER_3:
        mv      ph, a1;         mul     c, a3;
      add     a2, NULL, a3;
      mv      pl, a2;
      add     a1, a2, a2;     ldy     (i2)+1, a1;     /* -11 */
      xor     a2, a3, a3;
      not     a3, a3;
      jn      a3, W_FILTER_4;
      nop
      jn      a2, W_FILTER_4;
      lsr     ONES, a2;
      not     a2, a2;

W_FILTER_4:
      jn      a0, W_FILTER_1;
```

```
        sty     a2, (i3)+1;     add     NULL, NULL, a2;
          stx     a2, (i2);                               /* 0 */

/* RPE grid selection (4.2.14) */
RPE_SELECT:
          ldc     #SEL, i2;
          ldc     #TEMP, i4;
          ldc     #2, i5;
          ldc     #CURRENT, i6;
          ldc     #1, i7;
          ldc     #0x4000, c;
          ldc     #-40, a0;

          ldy     (i4)+1, d;
          mul     c, d;
          mv      ph, a3;           ldy     (i4)-1, d;
          sub     a0, ONES, a0;     mul     c, d;   stx     a3, (i4)*;      sty     a3, (i6)*
          mv      ph, a3;           ldy     (i4)-1, d;

RPE_SELECT_1:
          sub     a0, ONES, a0;     mul     c, d;   stx     a3, (i4)*;      sty     a3, (i6)*
          mv      ph, a3;           ldy     (i4)-1, d;
          sub     a0, ONES, a0;     mul     c, d;   stx     a3, (i4)*;      sty     a3, (i6)*
          mv      ph, a3;           ldy     (i4)-1, d;
          sub     a0, ONES, a0;     mul     c, d;   stx     a3, (i4)*;      sty     a3, (i6)*
          jn      a0, RPE_SELECT_1;
          mv      ph, a3;           ldy     (i4)-1, d;

RPE_SELECT_2:
          add     NULL, NULL, a1; ldx     (i2)+1, a0;     /* -4 */
          stx     a1, (i2)+1;       sty     a1, (i2)+1;     /* EM */
          sty     a1, (i2);                                 /* Mc */

RPE_SELECT_2_1:
          ldc     #TEMP, i4;
          ldc     #CURRENT, i6;

          sub     a0, ONES, a0;     ldx     (i2), i5;       /* 0 */
          ldx     (i2)-2, i7;       ldy     (i4)*, NULL;    /* 0 */
          ldx     (i6)*, NULL;      ldy     (i2)+4, a3;     /* -12 */
          ldx     (i2)-3, i5;       ldy     (i2)-3, i7;     /* 3, 3 */
          ldx     (i4)*, c;         ldy     (i6)*, d;
          sub     a3, ONES, a3;     mul     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;

RPE_SELECT_2_2:
          sub     a3, ONES, a3;     mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;
          sub     a3, ONES, a3;     mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;
          sub     a3, ONES, a3;     mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;
          jn      a3, RPE_SELECT_2_2;
          sub     a3, ONES, a3;     mac     c, d;   ldx     (i4)*, c;       ldy     (i6)*, d;

          mv      ph1, a1;          ldx     (i2), a3;       /* EM */
          mv      pl, a2;           ldy     (i2)+2, d;      /* EM */
          lsl     a2, a2;           sty     a1, (i2);       /* temp */
          stx     a2, (i2)-1;                               /* temp */
          l_sub   a3, a2, d, a1
          not     a2, a2;           ldx     (i2)+1, a1;     /* 0 */
          jn      a2, RPE_SELECT_2_3;
          ldx     (i2)-1, a2;       ldy     (i2)-1, a3;     /* temp */
          sty     a1, (i2)-1;                               /* Mc */
          stx     a2, (i2)+1;       sty     a3, (i2)+1;     /* EM */

RPE_SELECT_2_3:
          sub     a1, ONES, a1;
          jn      a0, RPE_SELECT_2_1;
          stx     a1, (i2);         add     NULL, NULL, a2; /* 0 */

          stx     a2, (i2);         ldy     (i2), a1;       /* 0, Mc */

/* down sampling by a factor of 3 to get the selected xM[0..12] RPE sequence */
RPE_SELECT_3:
          ldc     #XMI, i4;
          ldc     #TEMP, i6;

          ldy     (i2)-2, i7;                               /* Mc */
          ldx     (i6)*, NULL;      ldy     (i2)+2, a0;     /* -12 */
        ldy     (i6)+3, a3;       stx     a1, (i1)+1;
          ldc     #1, i5;
          ldc     #3, i7;
          sub     a0, ONES, a0;   ldy     (i6)*, a3;      stx     a3, (i4)*;

RPE_SELECT_3_1:
          sub     a0, ONES, a0;   ldy     (i6)*, a3;      stx     a3, (i4)*;
          sub     a0, ONES, a0;   ldy     (i6)*, a3;      stx     a3, (i4)*;
          sub     a0, ONES, a0;   ldy     (i6)*, a3;      stx     a3, (i4)*;
          jn      a0, RPE_SELECT_3_1;
          sub     a0, ONES, a0;   ldy     (i6)*, a3;      stx     a3, (i4)*;

/* APCM quantisation of the selected RPE sequence (4.2.15) */
APCM_QUANT:
          ldc     #APC, i2;
          ldc     #RAN, i3;
          ldc     #XMI, i6;

        /* find the maximum absolute value xmax of xM[0..12] */
        ldx     (i6)+1, a1;     add     NULL, NULL, a3;
        ldx     (i2)+1, a0;     /* -13 */
        s_abs     a1, a2

APCM_QUANT_1:
        sub     a2, a3, a1;
          add     a1, ONES, a1;
          jn      a1, APCM_QUANT_1_1;
          sub     a0, ONES, a0;   ldx     (i6)+1, a1;
          add     NULL, a2, a3;

APCM_QUANT_1_1:
        s_abs     a1, a2
          jn      a0, APCM_QUANT_1;
        nop

/* quantising and coding of xmax to get *xmaxc_p */
APCM_QUANT_2:
          add     NULL, NULL, a1; ldx     (i3), a2;       /* -6 */
          stx     a1, (i2)-1;     sub     a2, ONES, a2;   /* exp */
          ldy     (i3)+1, c;                              /* 128 */
          mul     c, a3;          sty     a3, (i2)+1;     /* xmaxc */
          mv      ph, a3;

APCM_QUANT_2_1:
          not     a3, a0;
          sub     a0, ONES, a0;
          jn      a0, APCM_QUANT_2_2;
          lsr     a3, a3;
          sub     NULL, ONES, a1;

APCM_QUANT_2_2:
          cmpz    a1, a0;
          not     a0, a0;
          jn      a0, APCM_QUANT_2_3;
          ldx     (i2), a0;                               /* exp */
          sub     a0, ONES, a0;
          stx     a0, (i2);                               /* exp */

APCM_QUANT_2_3:
          jn      a2, APCM_QUANT_2_1;
          sub     a2, ONES, a2;

          ldc     #SHIFT, i4;
          ldx     (i3), a1;                               /* 5 */
          add     a1, a0, a1;     ldx     (i2)+1, NULL;
          stx     a1, (i2);       lsl     a0, a3;         /* temp2 */
          ldx     (i2)-2, i5;     lsl     a3, a3;         /* temp2 */
          ldx     (i4)*, NULL;    ldy     (i2), d;        /* xmaxc */
```

```
        ldy     (i4), c;
        mul     c, d;           lsl     a3, a1;
        mv      ph, a2;
        add     a1, a2, a1;
      sty     a1, (i2)+1;     stx     a1, (i1)+1;     /* xmaxc */

/* quantising and coding of the xM[0..12] RPE sequence to get the xMc[0..12] */
/* compute exponent and mantissa decoded version of *xmaxc_p */
APCM_QUANT_3:
        ldy     (i3)+1, a0;                             /* 15 */
        sub     a1, a0, a0;     ldx     (i3), c;        /* 8 */
        add     a0, ONES, a0;
        jn      a0, APCM_QUANT_3_1;
        add     NULL, NULL, a3;
        lsr     a1, a0;
        lsr     a0, a0;
        lsr     a0, a0;
        add     a0, ONES, a3;

APCM_QUANT_3_1:
        mul     c, a3;
        mv      p1, a2;
        sub     a1, a2, a1;     stx     a3, (i2);       /* exp */
        sty     a1, (i2);                               /* mant */

/* normalise mantissa 0 <= mant <= 7 */
APCM_QUANT_4:
        cmpz    a1, a2;         ldx     (i3)+1, a3;     /* 8 */
        not     a2, a2;         ldx     (i3)-1, a0;     /* -3 */
        jn      a2, APCM_QUANT_4_1;
        add     NULL, NULL, a2; ldy     (i2), a1;       /* mant */
        ldc     #-4, a3;
        stx     a3, (i2);                               /* exp */
        ldc     #15, a1;
        j       APCM_QUANT_4_4;
        sty     a1, (i2);                               /* mant */

APCM_QUANT_4_1:
        sub     a1, a3, a1;
        jn      a1, APCM_QUANT_4_2;
        sub     a0, ONES, a0;
        sub     NULL, ONES, a2;

APCM_QUANT_4_2:
        cmpz    a2, a1;
        not     a1, a1;
        jn      a1, APCM_QUANT_4_3;
        ldy     (i2), a1;                               /* mant */
        lsl     a1, a1;
        sub     a1, ONES, a1;
        sty     a1, (i2);                               /* mant */
        ldx     (i2), a1;                               /* exp */
        add     a1, ONES, a1;
        stx     a1, (i2);                               /* exp */

APCM_QUANT_4_3:
        jn      a0, APCM_QUANT_4_1;
        ldy     (i2), a1;                               /* mant */

APCM_QUANT_4_4:
        ldx     (i3)+1, a0;                             /* 8 */
        sub     a1, a0, a1;
        sty     a1, (i2);                               /* mant */

/* direct computation of xMc[0..12] using table NRFAC */
APCM_QUANT_5:
        ldc     #SHIFT, i4;
        ldc     #NRFACI, i6;
        ldy     (i2)+1, i7;     ldx     (i2)+1, a1;     /* mant, exp */
        ldy     (i3)+1, a0;                             /* 6 */
        sub     a0, a1, a0;     ldx     (i6)*, NULL;
        sty     a0, (i2);       ldx     (i6), c;        /* temp1 */
        stx     c, (i2);                                /* temp2 */
        ldy     (i2)-2, i5;                             /* temp1 */
        ldy     (i4)*, NULL;    ldx     (i2)+2, a0;     /* -13 */
        ldx     (i4), c;
        sty     c, (i2);                                /* temp1 */

        ldc     #XMI, i6;

APCM_QUANT_5_1:
        sub     a0, ONES, a0;   ldx     (i6), d;
        mul     c, d;           ldx     (i2), c;        /* temp2 */
        mv      p1, a1;
        mul     c, a1;          ldx     (i3), c;        /* 4096 */
        mv      ph1, a1;
        mul     c, a1;          ldy     (i3), a2;       /* 4 */
        mv      ph, a1;
        add     a1, a2, a1;     ldy     (i2), c;        /* temp1 */
        jn      a0, APCM_QUANT_5_1;
      sty     a1, (i6)+1;     stx     a1, (i1)+1;

/* APCM inverse quantisation (4.2.16) */
APCM_IQUANT:
        ldc     #APC, i2;
        ldc     #FAC1, i4;
        ldc     #SHIFT, i6;
        ldc     #6, a3;

        ldx     (i2)+1, a0;                             /* -13 */
        ldy     (i2)+1, i5;     ldx     (i2)+1, a2;     /* mant, exp */
        ldx     (i4)*, NULL;    sub     a3, a2, a2;
        ldy     (i4), a1;       add     a2, ONES, a3;
        sty     a1, (i2)+1;     add     NULL, NULL, a1; /* temp1 */
        jn      a3, APCM_IQUANT_1;
        stx     a1, (i2);       abs     a2, a1;         /* temp3 */
        stx     a3, (i2);
        ldx     (i2), i7;                               /* temp3 */
        ldx     (i6)*, NULL;
        ldx     (i6), d;
        stx     d, (i2);                                /* temp3 */

APCM_IQUANT_1:
        ldc     #XMI, i4;
        ldc     #SHIFT, i6;
        ldc     #7, d;

        ldx     (i2)-1, NULL;
        stx     a1, (i2);                               /* temp2 */
        ldx     (i2), i7;
        ldx     (i6)*, NULL;
        stx     a3, (i2)+1;     ldy     (i4)+1, a1;     /* temp2 */

APCM_IQUANT_1_1:
        lsl     a1, a1;         ldy     (i2)-1, c;      /* temp4 */
        sub     a1, d, a1;
        mul     c, a1;          ldy     (i2)+1, c;      /* temp1 */
        mv      p1, a1;
        s_mur   a2, c, a1, a3
        ldx     (i2)-1, a2;     sub     a0, ONES, a0;   /* temp3 */
        add     a3, a2, a3;     ldx     (i2)+1, a1;     /* temp2 */
        ldx     (i6), c;
        mul     c, a3;          ldy     (i6), c;
        jn      a1, APCM_IQUANT_1_2;
        mv      p1, a2;         ldy     (i4)-1, a1;
        mul     c, a3;          not     c, a2;
        jn      a2, APCM_IQUANT_1_2;
        mv      ph, a2;
        not     a3, a2;
        jn      a2, APCM_IQUANT_1_2;
        lsr     a3, a2;
        or      c, a2, a2;

APCM_IQUANT_1_2:
```

```
        jn      a0, APCM_IQUANT_1_1;
        stx     a2, (i4)+2;

/* RPE grid positioning (4.2.17) */
RPE_POSIT:
          ldc     #SEL, i2;
          ldc     #CURRENT, i4;
          ldc     #-39, a0;
          add     NULL, NULL, a3; ldx     (i2)+2, NULL;

RPE_POSIT_1:
          sub     a0, ONES, a0;   sty     a3, (i4)+1;
          sub     a0, ONES, a0;   sty     a3, (i4)+1;
          sub     a0, ONES, a0;   sty     a3, (i4)+1;
          sub     a0, ONES, a0;   sty     a3, (i4)+1;
          jn      a0, RPE_POSIT_1;
          sub     a0, ONES, a0;   sty     a3, (i4)+1;

RPE_POSIT_2:
          ldc     #CURRENT, i4;
          ldc     #XMI, i6;
          ldc     #1, i7;
          ldc     #-13, a0;

          ldy     (i2), i5;                               /* Mc */
          sub     a0, ONES, a0;   ldy     (i4)*, NULL;    ldx     (i6)*, a3;
          ldc     #3, i5;
          sub     a0, ONES, a0;   sty     a3, (i4)*;      ldx     (i6)*, a3;

RPE_POSIT_2_1:
          sub     a0, ONES, a0;   sty     a3, (i4)*;      ldx     (i6)*, a3;
          sub     a0, ONES, a0;   sty     a3, (i4)*;      ldx     (i6)*, a3;
          sub     a0, ONES, a0;   sty     a3, (i4)*;      ldx     (i6)*, a3;
          jn      a0, RPE_POSIT_2_1;
          sub     a0, ONES, a0;   sty     a3, (i4)*;      ldx     (i6)*, a3;

/* update of the reconstructed short term residual signal dp[-120..-1] (4.2.18) */
UPDATE_DP:
          ldc     #UPD, i2;
          ldc     #PAST, i4;
          ldc     #PAST, i6;
          ldx     (i2)+1, a0;     ldy     (i2)+1, i7;     /* -80, 40 */
          ldy     (i6)*, NULL;    ldx     (i2), a3;       /* -40 */

UPDATE_DP_1:
          sub     a0, ONES, a0;   ldx     (i6)+1, a1;
          stx     a1, (i4)+1;
          sub     a0, ONES, a0;   ldx     (i6)+1, a1;
          jn      a0, UPDATE_DP_1;
          stx     a1, (i4)+1;

UPDATE_DP_2:
          ldc     #CURRENT, i6;

UPDATE_DP_2_1:
        sub     a3, ONES, a3;   ldx     (i6), d;        ldy     (i6), a2;
        s_add   a0, a2, d, a1
        jn      a3, UPDATE_DP_2_1;
        stx     a1, (i4)+1;     ldy     (i6)+1, NULL;

UPDATE_DP_3:
          jr
          nop

          .sect   code
/* write output */
WRITE_CODE:
          ldc     #-76, a0;
          ldc     #OUT, i6;
          ldc     #0x4801, i1;

1:
          sub     a0, ONES, a0;   ldx     (i6)+1, a1;
          jn      a0, 1b;
          stx     a1, (i1);

          jr
          nop

          .sect   code
INIT:
          j       MAIN;
          nop

          .sect   code
          .org    0xface
/* exit */
EXIT:
_face:
          j       _face;
          nop

          .end


/* gsmdec.asm ver 3.1.2 Dec 1997*/

.include        "oper.asm"
.include        "tables.asm"

          .sect   code
          .org    0x0000

          j       INIT;
          nop

          .sect   code
          .org    0x0064
MAIN:
          j       READ_DATA;
          ldc     #@+1, lr0;

        ldc     #SEQ1, i4;
          ldc     #STORE, i3;
          stx     i4, (i3);
        ldc     #IN, i4;
        ldc     #STORE2, i3;
        stx     i4, (i3);

1:
        j       RPE;
        ldc     #@+1, lr0;

        j       LTP;
        ldc     #@+1, lr0;

        ldc     #COUNT, i2;
          ldy     (i2), a0;
          sub     a0, ONES, a0;
          jn      a0, 1b;
          sty     a0, (i2);
          ldc     #-4, a0;
          sty     a0, (i2);

        j       STS;
          ldc     #@+1, lr0;

          ldc     #IFI, i2;
          ldc     #-1, i3;
          ldc     #IN, i6;

          ldc     #LAR1, i4;
          ldc     #-13, a0;
```

```
        j       S_FILTER;
          ldc     #@+1, lr0;

          ldc     #LAR2, i4;
          ldc     #-14, a0;
        j       S_FILTER;
          ldc     #@+1, lr0;

          ldc     #LAR3, i4;
          ldc     #-13, a0;
        j       S_FILTER;
          ldc     #@+1, lr0;

          ldc     #LAR4, i4;
          ldc     #-120, a0;
        j       S_FILTER;
          ldc     #@+1, lr0;

        j       POSTPROC;
        ldc     #@+1, lr0;

        j       WRITE_CODE;
        ldc     #@+1, lr0;

          ldc     #COUNT2, i2;
          ldx     (i2), a0;
          sub     a0, ONES, a0;
          jn      a0, MAIN;
          stx     a0, (i2);

          j       EXIT;
          nop

          .sect   code
/* read input */
READ_DATA:
        ldc     #-76, a0;
        ldc     #OUT, i6;
          ldc     #0x4800, i0;

1:
          sub     a0, ONES, a0;   ldx     (i0), a1;
          jn      a0, 1b;
        stx     a1, (i6)+1;     sty     a1, (i6)+1;

          jr
          nop

          .sect   code
/* RPE decoding section */
RPE:
        ldc     #STORE, i3;
        ldc     #SEQ1, i4;
        ldx     (i3), i4;
        ldc     #CAL, i5;
        ldc     #SEL, i6;
        ldc     #APC, i7;
        ldx     (i4)+1, a1;     ldy     (i6)+2, NULL;
        ldx     (i5)+1, NULL;
        stx     a1, (i5);       ldy     (i4)+1, a1;     /* scal */
        sty     a1, (i5);       ldx     (i4)+1, a1;     /* bc */
        sty     a1, (i6);       ldx     (i4)+1, a1;     /* Mc */
        sty     a1, (i7);       ldx     (i7), a0;       /* xmaxc */

        ldc     #XMI, i6;

1:
        sub     a0, ONES, a0;   ldx     (i4)+1, a1;
        jn      a0, 1b;
        sty     a1, (i6)+1;

        ldx     (i4), NULL;
        stx     i4, (i3);

/* get mant and exp of xmax, modified from (4.2.15) according to (4.3.1) */
RPE_DECODE:
        ldc     #APC, i2;
        ldc     #RAN, i3;
        ldx     (i3)+1, NULL;   ldy     (i2)+1, a1;     /* xmaxc */

/* compute exponent and mantissa decoded version of xmaxc */
RPE_DECODE_1:
        ldy     (i3)+1, a0;                             /* 15 */
          sub     a1, a0, a0;     ldx     (i3), c;      /* 8 */
        add     a0, ONES, a0;
        jn      a0, RPE_DECODE_1_1;
          add     NULL, NULL, a3;
          lsr     a1, a0;
          lsr     a0, a0;
          lsr     a0, a0;
          add     a0, ONES, a3;

RPE_DECODE_1_1:
          mul     c, a3;
          mv      p1, a2;
          sub     a1, a2, a1;     stx     a3, (i2);       /* exp */
          sty     a1, (i2);                               /* mant */

/* normalise manttissa 0 <= mant <= 7 */
RPE_DECODE_2:
          cmpz    a1, a2;         ldx     (i3)+1, a3;     /* 8 */
          not     a2, a2;         ldx     (i3)-1, a0;     /* -3 */
        jn      a2, RPE_DECODE_2_1;
          add     NULL, NULL, a2; ldy     (i2), a1;       /* mant */
          ldc     #-4, a3;
          stx     a3, (i2);                               /* exp */
          ldc     #15, a1;
        j       RPE_DECODE_2_4;
          sty     a1, (i2);                               /* mant */

RPE_DECODE_2_1:
          sub     a1, a3, a1;
        jn      a1, RPE_DECODE_2_2;
          sub     a0, ONES, a0;
          sub     NULL, ONES, a2;

RPE_DECODE_2_2:
          cmpz    a2, a1;
          not     a1, a1;
        jn      a1, RPE_DECODE_2_3;
          ldy     (i2), a1;                               /* mant */
          lsl     a1, a1;
          sub     a1, ONES, a1;
          sty     a1, (i2);                               /* mant */
          ldx     (i2), a1;                               /* exp */
          add     a1, ONES, a1;
          stx     a1, (i2);                               /* exp */

RPE_DECODE_2_3:
        jn      a0, RPE_DECODE_2_1;
          ldy     (i2), a1;                               /* mant */

RPE_DECODE_2_4:
          ldx     (i3)+1, a0;                             /* 8 */
          sub     a1, a0, a1;
          sty     a1, (i2);                               /* mant */

/* APCM inverse quantisation (4.2.16) */
APCM_IQUANT:
        ldc     #APC, i2;
        ldc     #FACI, i4;
        ldc     #SHIFT, i6;
        ldc     #6, a3;
```

```
        ldx     (i2)+1, a0;                             /* -13 */
        ldy     (i2)+1, i5;     ldx     (i2)+1, a2;     /* mant, exp */
        ldx     (i4)*, NULL;    sub     a3, a2, a2;
        ldy     (i4), a1;       add     a2, ONES, a3;
        sty     a1, (i2)+1;     add     NULL, NULL, a1; /* temp1 */
        jn      a3, APCM_IQUANT_1;
        stx     a1, (i2);       abs     a2, a1;         /* temp3 */
        stx     a3, (i2);
        ldx     (i2), i7;                               /* temp3 */
        ldx     (i6)*, NULL;
        ldx     (i6), d;
        stx     d, (i2);                                /* temp3 */

APCM_IQUANT_1:
        ldc     #XMI, i4;
        ldc     #SHIFT, i6;
        ldc     #7, d;

        ldx     (i2)-1, NULL;
        stx     a1, (i2);                               /* temp2 */
        ldx     (i2), i7;
        ldx     (i6)*, NULL;
        stx     a3, (i2)+1;     ldy     (i4)+1, a1;     /* temp2 */

APCM_IQUANT_1_1:
        lsl     a1, a1;         ldy     (i2)-1, c;      /* temp4 */
        sub     a1, d, a1;
        mul     c, a1;          ldy     (i2)+1, c;      /* temp1 */
        mv      p1, a1;
        s_mur   a2, c, a1, a3
        ldx     (i2)-1, a2;     sub     a0, ONES, a0;   /* temp3 */
        add     a3, a2, a3;     ldx     (i2)+1, a1;     /* temp2 */
        ldx     (i6), c;
        mul     c, a3;          ldy     (i6), c;
        jn      a1, APCM_IQUANT_1_2;
        mv      p1, a2;         ldy     (i4)-1, a1;
        mul     c, a3;          not     c, a2;
        jn      a2, APCM_IQUANT_1_2;
        mv      ph, a2;
        not     a3, a2;
        jn      a2, APCM_IQUANT_1_2;
        lsr     a3, a2;
        or      c, a2, a2;

APCM_IQUANT_1_2:
        jn      a0, APCM_IQUANT_1_1;
        stx     a2, (i4)+2;

/* RPE grid positioning (4.2.17) */
RPE_POSIT:
        ldc     #SEL, i2;
        ldc     #CURRENT, i4;
        ldc     #-39, a0;
        add     NULL, NULL, a3; ldx     (i2)+2, NULL;

RPE_POSIT_1:
        sub     a0, ONES, a0;   sty     a3, (i4)+1;
        sub     a0, ONES, a0;   sty     a3, (i4)+1;
        sub     a0, ONES, a0;   sty     a3, (i4)+1;
        sub     a0, ONES, a0;   sty     a3, (i4)+1;
        jn      a0, RPE_POSIT_1;
        sub     a0, ONES, a0;   sty     a3, (i4)+1;

RPE_POSIT_2:
        ldc     #CURRENT, i4;
        ldc     #XMI, i6;
        ldc     #1, i7;
        ldc     #-13, a0;

        ldy     (i2), i5;                               /* Mc */
        sub     a0, ONES, a0;   ldy     (i4)*, NULL;    ldx     (i6)*, a3;
        ldc     #3, i5;
        sub     a0, ONES, a0;   sty     a3, (i4)*;      ldx     (i6)*, a3;
RPE_POSIT_2_1:
        sub     a0, ONES, a0;   sty     a3, (i4)*;      ldx     (i6)*, a3;
        sub     a0, ONES, a0;   sty     a3, (i4)*;      ldx     (i6)*, a3;
        sub     a0, ONES, a0;   sty     a3, (i4)*;      ldx     (i6)*, a3;
        jn      a0, RPE_POSIT_2_1;
        sub     a0, ONES, a0;   sty     a3, (i4)*;      ldx     (i6)*, a3;

RPE_POSIT_2_2:
        jr
        nop

        .sect   code
/* LTP synthesis section */
LTP:

/* long term synthesis filtering  (4.3.2) */
/* check limits of Nr */
LTP_SYNTH:
        ldc     #CAL, i2;
        ldc     #QLBI, i4;
        ldc     #PAST, i6;
        ldc     #40, a0;
        ldc     #120, a1;
        ldx     (i2)+1, NULL;
        ldx     (i2)-1, a3;             /* scal */

        sub     a1, a3, a2;
        jn      a2, LTP_SYNTH_1;
        nop

        sub     a3, a0, a2;
        jn      a2, LTP_SYNTH_1;
        nop

        sub     a1, a3, a1;
        sty     a1, (i2);               /* Nc */

/* decoding of the coded LTP gain bcr*/
LTP_SYNTH_1:
        ldy     (i2)+1, i7;     ldx     (i2)+1, a0;     /* Nc, -40 */
        ldx     (i6)*, NULL;    ldy     (i2)+2, i5;     /* bc */
        ldx     (i4)*, NULL;
        ldy     (i4), c;
        ldc     #STORE2, i3;
        ldc     #CURRENT, i4;
        ldc     #IN, i5;
        sty     c, (i2);        ldx     (i3), i5;       /* QLB */

/* computation of the reconstructed short term residual signal drp[0..39] */
LTP_SYNTH_1_1:
        sub     a0, ONES, a0;   ldy     (i4), d;
        ldy     (i2), c;        ldx     (i6)+1, a1;
        s_mur   a2, c, a1, a3
        s_add   a2, a3, d, a1
        stx     a1, (i4)+1;
        jn      a0, LTP_SYNTH_1_1;
        stx     a1, (i5)+1;     sty     a1, (i5)+1;
        stx     i5, (i3);

/* update the reconstructed short term residual signal drp[1..120] */
LTP_SYNTH_2:
        ldc     #UPD, i2;
        ldc     #PAST, i4;
        ldc     #PAST, i6;
        ldx     (i2)+1, a0;     ldy     (i2)+1, i7;     /* -80, 40 */
        ldy     (i6)*, NULL;    ldx     (i2), a3;       /* -40 */

LTP_SYNTH_2_1:
        sub     a0, ONES, a0;   ldx     (i6)+1, a1;
        stx     a1, (i4)+1;
```

```
          sub     a0, ONES, a0;   ldx     (i6)+1, a1;
        jn      a0, LTP_SYNTH_2_1;
          stx     a1, (i4)+1;

LTP_SYNTH_2_2:
          ldc     #CURRENT, i6;

LTP_SYNTH_2_2_1:
        sub     a3, ONES, a3;   ldx     (i6)+1, a1;
        jn      a3, LTP_SYNTH_2_2_1;
        stx     a1, (i4)+1;

LTP_SYNTH_2_3:
          jr
          nop

          .sect   code
/* short term synthesis filtering section */
STS:

/* decoding of the coded log-area ratios (4.2.8) */
DECODE_LAR:
          ldc     #MICI, i2;
          ldc     #BI, i3;
        ldc     #OUT, i4;
          ldc     #INVAI, i5;
          ldc     #-8, a0;

DECODE_LAR_1:
          sub     a0, ONES, a0;   ldx     (i2), a1;       ldy     (i4), a2;
          add     a1, a2, a1;     ldx     (i5), c;        ldy     (i3), a2;
          mul     c, a1;          ldy     (i3)+1, NULL;   lsl     a2, a2;
          mv      p1, a1;         ldy     (i5)+1, c;
          sub     a1, a2, a2;     ldx     (i2)+1, NULL;
          s_mur   a1, c, a2, a3
          lsl     a3, a3;
          jn      a0, DECODE_LAR_1;
        sty     a3, (i4)+1;

        ldc     #-8, a0;
        ldc     #R9, i6;
        ldx     (i4)-1, NULL;

DECODE_LAR_2:
        sub     a0, ONES, a0;   ldy     (i4)-1, a1;
        jn      a0, DECODE_LAR_2;
        sty     a1, (i6)+1;

/* interpolation of the LARpp[1..8] to get LARp[1..8] (4.2.9.1) */
INTE_LAR:
          ldc     #R9, i4;
          ldc     #LAR1, i6;
          ldc     #-8, a0;
          ldc     #0x4000, c;
          ldx     (i6)-1, NULL;   add     NULL, NULL, a3;

INTE_LAR_1:
          sub     a0, ONES, a0;   ldy     (i4), a1;
          mul     c, a1;          ldx     (i4), a2;
          mul     c, a2;          mv      ph, a1;
          mv      ph, a2;         stx     a3, (i6)+1;
          add     a1, a2, a3;     ldx     (i4)+1, a2;
          not     a2, a1;
          jn      a1, INTE_LAR1_1;
          lsr     a2, a2;
          lsl     c, a1;
          or      a1, a2, a2;

INTE_LAR1_1:
        jn      a0, INTE_LAR_1;
          add     a2, a3, a3;

          ldc     #R9, i4;
          ldc     #-8, a0;
          ldc     #0x8000, c;

INTE_LAR_2:
          sub     a0, ONES, a0;   ldy     (i4), a1;
          ldx     (i4)+1, a2;     sub     NULL, a1, a1;
          mul     c, a1;          sub     NULL, a2, a2;
          mul     c, a2;          mv      ph, a1;
          mv      ph, a2;         stx     a3, (i6)+1;
        jn      a0, INTE_LAR_2;
          add     a1, a2, a3;

          ldc     #R9, i4;
          ldc     #-8, a0;
          ldc     #0x4000, c;

INTE_LAR_3:
          sub     a0, ONES, a0;   ldy     (i4), a1;
          mul     c, a1;          ldx     (i4), a2;
          mul     c, a2;          mv      ph, a1;         stx     a1, (i4);
          mv      ph, a2;         stx     a3, (i6)+1;
          add     a1, a2, a3;     ldy     (i4)+1, a1;
          not     a1, a2;
          jn      a2, INTE_LAR_3_1;
          lsr     a1, a1;
          lsl     c, a2;
          or      a2, a1, a1;

INTE_LAR_3_1:
          jn      a0, INTE_LAR_3;
          add     a1, a3, a3;

          ldc     #R9, i4;
          ldc     #-8, a0;
          ldc     #1, i5;
          ldc     #1, i7;

INTE_LAR_4:
          sub     a0, ONES, a0;   ldy     (i4)*, a3;      stx     a3, (i6)*;
          sub     a0, ONES, a0;   ldy     (i4)*, a3;      stx     a3, (i6)*;
          jn      a0, INTE_LAR_4;
          sub     a0, ONES, a0;   ldy     (i4)*, a3;      stx     a3, (i6)*;

/* computation of the rp[1..8] from the interpolated LARpp[1..8] (4.2.9.2) */
COEFFS:
          ldc     #COE, i2;
          ldc     #LAR1, i4;
          ldc     #-32, a0;
          ldx     (i4)+1, d;

COEFFS_1:
          abs     d, a2;          ldx     (i2)+1, a3;     /* 11059 */
          sub     a2, a3, a3;
          jn      a3, COEFFS_2;
          lsl     a2, a3;         ldx     (i2)-1, a1;     /* 20070 */

          sub     a2, a1, a1;     ldy     (i2)+1, a3;     /* 11059 */
          jn      a1, COEFFS_2;
          add     a2, a3, a3;     ldy     (i2)-1, a1;     /* 26112 */

          lsr     a2, a3;
          lsr     a3, a3;
          add     a3, a1, a3;

COEFFS_2:
          not     d, a2;
          jn      a2, COEFFS_3;
          sub     a0, ONES, a0;   ldx     (i4)-1, d;
          sub     NULL, a3, a3;

COEFFS_3:
```

```
        jn      a0, COEFFS_1;
        sty     a3, (i4)+2;

COEFFS_4:
        jr
        nop

/* short term synthesis filtering (4.3.4) */
S_FILTER:
        ldc     #UI, i7;
        sub     a0, ONES, a0;   ldx     (i6), a3;
        stx     a0, (i2)+1;     ldy     (i2)+1, a0;     /* temp1, -8 */
        stx     a3, (i2);       ldy     (i7)+1, NULL;   /* sri */

S_FILTER_1:
        sub     a0, ONES, a0;   ldy     (i7), a3;
        ldy     (i4), c;
        s_mur   a1, c, a3, a2
        ldx     (i2), d;        /* sri */
        s_sub   a1, d, a2, a3
        stx     a3, (i2);       ldy     (i4)+1, c;      /* sri */
        s_mur   a1, c, a3, a2
        ldy     (i7)-1, d;
        s_add   a1, a2, d, a3
        jn      a0, S_FILTER_1;
        sty     a3, (i7)+2;

        ldx     (i2)-1, a3;     ldy     (i7)-1, NULL;   /* sri */
        sty     a3, (i7);
        ldx     (i2), a0;       ldy     (i2), i5;       /* temp1, -8 */
        jn      a0, S_FILTER;
        stx     a3, (i6)+1;     ldy     (i4)*, NULL;

S_FILTER_2:
        jr
        nop

        .sect   code
/* postprocessing section */
POSTPROC:

/* deemphasis (4.3.5) */
DEEMP:
        ldc     #PRE, i2;
        ldc     #IN, i6;
        ldc     #-160, a0;
        ldc     #1, i3;

DEEMP_1:
        sub     a0, ONES, a0;   ldy     (i2)*, c;       ldx     (i2)*, d;
        mul     c, d;           ldx     (i2)*, c;       ldy     (i2)*, d;
        mac     c, d;           ldx     (i6), d;
        mv      ph1, a1;        ldx     (i2)-2, NULL;
        s_add   a2, a1, d, a3
        jn      a0, DEEMP_1;
        stx     a3, (i6)+1;     sty     a3, (i2);

/* upscale output signal */
UP_SCALE:
        ldc     #IN, i6;
        ldc     #-160, a0;

UP_SCALE_1:
        sub     a0, ONES, a0;   ldx     (i6), a1;
        ldx     (i6), d;
        s_add   a2, a1, d, a3
        jn      a0, UP_SCALE_1;
        stx     a3, (i6)+1;

/* truncating of output signal */
TRUNCATE:
        ldc     #IN, i6;
        ldc     #-160, a0;
        ldc     #-8, a3;

TRUNCATE_1:
        sub     a0, ONES, a0;   ldx     (i6), a1;
        and     a1, a3, a1;
        jn      a0, TRUNCATE_1;
        stx     a1, (i6)+1;

TRUNCATE_2:
        jr
        nop

        .sect   code
/* write output */
WRITE_CODE:
        ldc     #-160, a0;
        ldc     #IN, i6;
        ldc     #0x4801, i1;

1:
        sub     a0, ONES, a0;   ldx     (i6)+1, a1;
        jn      a0, 1b;
        stx     a1, (i1);

        jr
        nop

        .sect   code
INIT:
        j       MAIN;
        nop

        .sect   code
        .org    0xface
/* exit */
EXIT:
_face:
        j       _face;
        nop

        .end
```

Appendix I Results and Tables of Implementation of GSM Full-Rate GEPARD Code

Table I.1 Parameters of GSM full-rate encoder.

Input	sop[160]	input data
	z1	stored value of vector z1
	l_z2	stored value of vector l_z2
	mp	stored value of vector mp
	last_LARpp[8]	stored value of vector last_LARpp
	u[8]	stored value of vector u[8]
	dp[121]	stored value of vector dp
Output	LARc[8], Nc[4], bc[4], Mc[4], xmaxc[4], xMc[52]	coded parameters

Table I.2 Parameters of GSM full-rate decoder.

Input	LARc[8], Nc[4], bc[4], Mc[4], xmaxc[4], xMc[52]	coded parameters
	last_LARpp[8]	stored value of vector last_LARpp
	v[9]	stored value of vector u[8]
	msr	stored value of vector mp
Output	sop[160]	output data vector

Table I.3 Subblocks of the GSM full-rate encoder.

Section	Input Variables	Output Variables	Used Variables	Approximate Number of Cycles
Preprocessing				
scaling (4.2.1)	sop[160]	so[160]		1000
offset (4.2.2)	so[160]	sof[160]		3000
preemp (4.2.3)	sof[160]	s[160]		1000
LPC Analysis				
autocorr (4.2.4)	s[160]	l_ACF[9], s[160]		5000
schur (4.2.5)	l_ACF[9]	r[8]		1500
lars (4.2.6)	r[8]	LAR[8]		50
code_lar (4.2.7)	LAR[8]	LARc[8]	A[8], B[8], MAC[8], MIC[8]	50
Short Term Analysis Filtering				
decode_lar (4.2.8)	LARc[8]	LARpp[8]	INVA[8], MIC[8], B[8]	150
inte_lar (4.2.9.1)	LARpp[8]	LARp[32]		200
coeffs (4.2.9.2)	LARp[32]	rp[32]		300
i_filter (4.2.10)	s[160], rp[32]	d[160]		46000
Long Term Prediction (LTP)				
ltp_calc (4.2.11)	d[40]	d[40]	DBL[4]	10000
ltp_filter (4.2.12)	d[40]	e[40]	QLB[4]	700
RPE Encoding				
w_filter (4.2.13)	e[40]	x[40]	H[11]	1500
rpe_select (4.2.14)	x[40]	xM[13]	NRFAC[8]	400
apcm_quant (4.2.15)	xM[13]	xMc[13]	FAC[8]	300
apcm_iquant (4.2.16)	xMc[13]	xMp[13]		400
rpe_posit (4.2.17)	xMp[13]	ep[40]		100
update_dp (4.2.18)	ep[40]	dp[121]		500

Table I.4 Subblocks of the GSM full-rate decoder.

Section	Input Variables	Output Variables	Used Variables	Approximate Number of Cycles
RPE Decoding				
rpe_decode (4.3.1)	xmaxc	exp_p, mant_p		50
apcm_iquant (4.2.16)	xMc[13]	xMp[13]		300
rpe_posit (4.2.17)	xMp[13]	erp[40]		100
LTP Synthesis				
ltp_synth (4.3.2)	erp[40]	drp[40]	QBL[4]	1200
Short Term Synthesis Filtering				
decode_lar (4.2.8)	LARc[8]	LARpp[8]	INVA[8], MIC[8], B[8]	150
inte_lar (4.2.9.1)	LARpp[8]	LARp[32]		200
coeffs (4.2.9.2)	LARp[32]	rrp[32]		300
s_filter (4.3.4)	drp[160], rrp[32]	d[160]		44000
Postprocessing				
deemph (4.3.5)	sr[160]	sro[160]		2000
upscale (4.3.6)	sro[160]	srop[160]		2000
truncate (4.3.7)	srop[160]	srop[160]		600