Design and Implementation on DSP of the ETSI GSM Adaptive Multi-Rate Vocoder

by

Javier Martínez Morales

Supervisor:

Joan Serrat Fernández

Contents

Chapter 1

Introduction

1.1 Foreword

With the current focus on high-speed packet data transmission, it is easy to forget that the primary purpose of GSM digital telecommunication systems was speech transmission. The general perception is that the complexity of the overall system is associated with the management of the transmission link. However, there is a great deal of complexity in the compression and decompression of the audio captured by the microphone. Speech must be captured at a high enough sample rate and resolution to allow clear reproduction of the original sound and compressed in such a way as to maintain the fidelity of the audio over a limited bit rate, error-prone wireless transmission channel.

The way in which the human hearing system works allows the coder to create a perceptually similar result at the earpiece of the remote phone. The key principle behind the coders used in the GSM system is the mathematical modeling of the human vocal tract, leading to an efficient compression method for transmitting speech. A vocoder (combination of voice and coder) is used to describe these systems tailored for the compression of speech.

There are four codecs in use within the GSM that perform the compression operation. These are full rate, enhanced full rate (EFR), adaptive multi-rate (AMR) and half-rate speech codecs. The full-rate codec is a fairly computationally-efficient method of transmitting speech, but through the use of more intensive algorithms the quality of the speech can be improved. The full-rate codec was first implemented on the DSPs of the early 1990s and at that time it was not economically viable to use a better quality but more intensive algorithm. In the mid-1990s this was no longer an issue with the availability of higher power DSP cores, and so the EFR codec started to appear in handsets.

Full-rate and EFR codecs allow for good reproduction of speech when all their parameters can be decoded. Due to the redundancy on the transmission channel, many of the raw bits can be in error, but the parameters are still recoverable. However, when the parameters are lost or erroneous, the quality of the received signal decreases rapidly. It is this problem that the AMR codec attempts to resolve. By specifying a set of eight vocoders all sharing common mathematical components and operating at different rates, the amount of redundancy on the channel can be changed. This way, the quality of speech transmission can be slightly degraded by dropping to a lower coding rate, but with an increased confidence of recovering the coding parameters. The result is a better-perceived signal quality in the presence of increased interference on the carrier.

The speech coding functionality is riddled with mathematically intensive processes such as convolution, and as such is best implemented on dedicated digital signal processors with instructions to handle this type of computation. But, even with these specialized DSPs, optimization techniques are still necessary in the implementation of a stable speech codec with real-time performance.

1.2 Project objectives and methodology

This project has a clear goal. The Adaptive Multi-Rate (AMR) vocoder, as defined by ETSI (European Telecommunications Standards Institute) in the GSM 6.90 standard [14], has to be implemented into a specialized digital signal processor. It has to be a "bit-exact" implementation, as defined by ETSI, and it has to be fast enough so as to work in "real-time".

The ADSP-2181 processor, by Analog Devices, is the DSP chosen for the implementation. The use of a specialized DSP is a must. Although it would be possible to implement the vocoder on a general-purpose processor, the clock speed should be orders of magnitude faster to match the same execution speed [9]. Another important decision about the processor is choosing between a floating-point or a fixed-point computational core. Although very fast implementations are available for processors that offer floating-point support within the core [11], these implementations are not bit-exact, meaning that they do not result in the exact mathematical results as the fixed-point reference implementations do. So, being the ADSP-2181 a specialized, fixed-point, digital signal processor, it fulfills all the requirements for the project.

A first, simple, approach would be trying to accomplish the implementation by only using the development tools of the processor manufacturer, in this case Analog Devices. Provided that ETSI includes C-language code together with the AMR vocoder documentation, the Analog Devices' C compiler could be used to get a straight conversion from the C code into the assembling code of the DSP [7]. The problem is the compiler is not remotely good enough for this purpose. As the same manufacturer reckons, the compiler is able to translate simple functions with good results, but using it to translate the huge amount of code of the vocoder would produce unexpected results, illegible code and poor performance. And that supposing that the vocoder worked at all. So we will have to write the whole program in assembling code, from the first to the last function of the AMR vocoder.

We can list then, in a more detailed way, the main objectives of the project as follows:

- The first objective is to study the Adaptive Multi-Rate vocoder standard, provided by ETSI. All the documentation they include has to be studied, but also the theoretical background behind it. That includes the study of voice production, voice coding and the history of the different vocoders that preceded the AMR vocoder. Throughout the implementation process, the translation of that theory to programming code will also have to be understood.

- The second objective is the study of the digital signal processor ADSP-2181, from Analog Devices, to be able to implement the AMR Vocoder into it. Part of this objective will be studying how DSP's are specialized in signal processing algorithms, and how we can take profit of the dedicated instructions to optimize the program.

- The third objective seems to be the most important objective and the main purpose of the project: the bit-exact implementation of the vocoder. Not only has our vocoder to work, but it also has to pass all the official tests provided by ETSI to guarantee that the encoded and decoded signals it produces are, bit by bit, identical to the ones expected.

- The last objective is to obtain a “real-time” implementation, that is, the time the vocoder needs to process the voice signal has to be less than the signal duration itself. Otherwise, we wouldn't get a working stable system. As we have just seen, this objective has had a first implication: the C compiler provided by Analog Devices is not good enough to be used. But there's probably a second implication: even our own-made assembling code will have to follow a strong optimization process.

Now that we have written the main objectives of the project, let's see how they have been developed, beginning with a brief description of how this report will be organized.

1.3 Memory structure

Immediately after this introductory chapter, where we have seen the purpose of the project, in chapter 2, “GSM Adaptive Multi-Rate Vocoder”, the most theoretical chapter, the study of the AMR vocoder is presented. It is one of the objectives of the project, so a detailed description is given, beginning with its “Origins and present situation”. A general view is offered in “Adaptive Multi-Rate (AMR) coding: general description”, and then special focus on the encoding and decoding parts is put, respectively, in “GSM AMR speech encoder” and “GSM AMR speech decoder”. Finally, some additional parts of the vocoder are studied in “Support functions”.

Chapter 3, “Workspace: hardware and software development tools”, will describe the material, documentation, software and, in general, everything that will be necessary to develop the project. The chapter begins with “General description”, where a justification of the workspace needed is presented. It follows with “ADSP-2181 processor from Analog Devices”, an in-depth description of the digital signal processor that will be used to implement the AMR vocoder, with the main characteristics that made it suitable for this purpose. Finally, an important tool among the tools used throughout the project is given special attention in “Running, debugging and analyzing the assembling code: VisualDSP”.

Chapter 4 is titled “Implementation and optimization” and is the most practical of all chapters. The first part, “Methodology”, describes the “day-by-day” realization of the project. Then, “Cases of special interest” shows concrete situations encountered during the implementation that require a detailed explanation, and the solutions applied. Lastly, “Project results” presents the complete work done, with some numeric results that are analyzed altogether with more subjective impressions.

The last chapter, “Conclusions”, tries to wrap up the project discussing about the “Project objectives accomplished” and the “Future perspectives”.

There are three appendixes accompanying the five main chapters. Appendix A, “Speech production and coding” is a theoretical background to the study of the AMR vocoder, giving information about the voice signals, how are they produced and how can they be coded. Appendix B, “Simplified instruction set for the ADSP-2181”, explains the basics of the assembling programming language that will be used in the implementation. Appendix C, “Implementation of the Levinson-Durbin algorithm”, transcribes the complete source code of one of the functions of the vocoder (the well-known Levinson-Durbin algorithm), first the C source code, provided by ETSI as a guidance, and then the assembling code, implemented during the project.

Finally, the bibliography and a quick reference for terms and acronyms used or related to the text are included.

Chapter 2

GSM Adaptive Multi-Rate Vocoder

2.1 Origins and present situation

In this chapter, we will focus on the study of the GSM Adaptive Multi-Rate (AMR) vocoder, which is one of the purposes of this project.

Adaptive Multi-Rate (AMR) is an audio data compression scheme optimized for speech coding. AMR operating at various bit rates is built into every GSM and WCDMA phone, ensuring that content generated by AMR can be played by virtually any wireless phone in the world, including hundreds of millions of new phones every year.

AMR was adopted as the standard speech codec by 3GPP (3rd Generation Partnership Project) in October 1998 and is now widely used in GSM and UMTS [22]. It offers substantial improvement over previous GSM speech codecs in error robustness by adapting speech and channel coding depending on channel conditions.

Initially developed for the GSM system, the single most deployed 2G mobile telecommunication system worldwide, AMR was also standardized by the European Telecommunications Standards Institute (ETSI) in 1999 and adopted by the 3GPP as the mandatory codec for narrow-band telephony in 2.5G/3G wireless systems based on evolved GSM core networks (WCDMA, EDGE, GPRS). It is also the mandatory codec for 3G

(H.324M) terminals supporting video telephony and the default codec for Multimedia Messaging Services (MMS) as defined by the Open Mobile Alliance (OMA).

Due to its great success, its scope went beyond the bounds of speech communication in telephony systems, and we can now find AMR in a number of applications: multimedia services, voice over IP, Wi-Fi telephony, portable audio devices, Internet applications, digital radio broadcasting and many more. A recent remarkable example is that, in 2006, it was included in the PacketCable 2.0 specification, an important international project that seeks for interoperable interface specifications in order to deliver real-time multimedia services over two-way cable networks.

AMR has also become a well-known file format for storing spoken audio using the AMR codec. Many modern mobile telephone handsets allow to store short recordings in the AMR format. The common filename extension is .amr.

We will here study the Adaptive Multi-Rate vocoder as defined by the GSM 6.90 standard. This European standard was produced by ETSI Technical Committee Special Mobile Group (SMG). The complete description of the GSM AMR can therefore be found in the documents provided by ETSI.

2.2 Adaptive Multi-Rate (AMR) coding: general description

The GSM Adaptive Multi-Rate (AMR) vocoder belongs to the Algebraic Code-excited linear prediction (ACELP) vocoders. The bit-rates of the source codec for the adaptive multi-rate are: 4.75, 5.15, 5.90, 6.70, 7.40, 7.95, 10.2 or 12.2 kbit/s. Basic theoretical background concerning voice production, voice coding and the different types of vocoders can be found in the Appendix A, so here we will focus on the specific details concerning the ETSI standard [14].

Figures 2.1 (a) and 2.1 (b) present a reference configuration where the various speech processing functions are identified. In these figures, the relevant documents for each function are also indicated. The audio parts including analogue to digital and digital to analogue conversion are included, to show the complete speech path between the audio input/output in the Mobile Station (MS) and the digital interface of the PSTN (Public Switched Telephone Network). These aspects are only considered to the extent that the performance of the audio parts affect the performance of the speech transcoder.

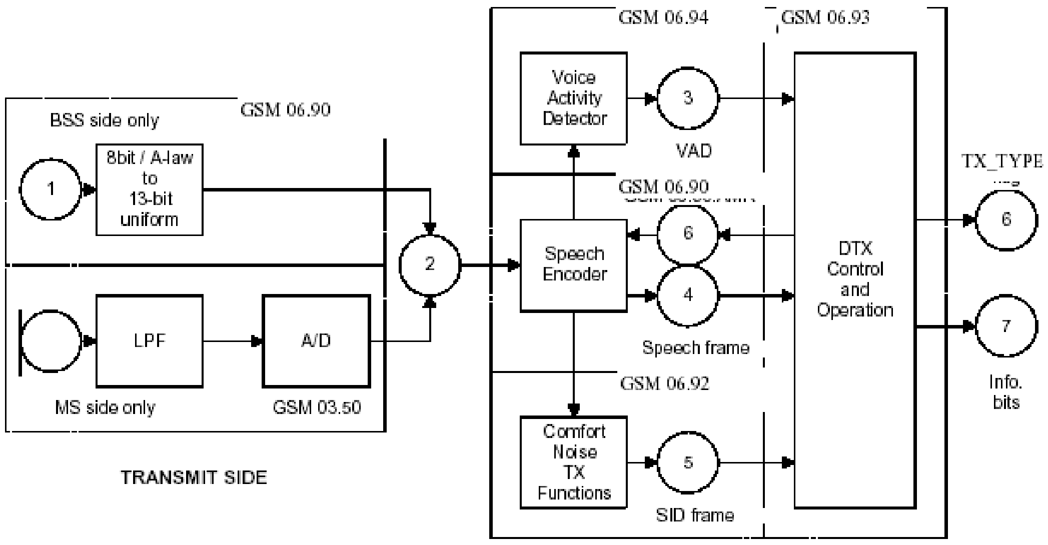


Figure 2.1 (a): Overview of audio processing functions (transmit side)

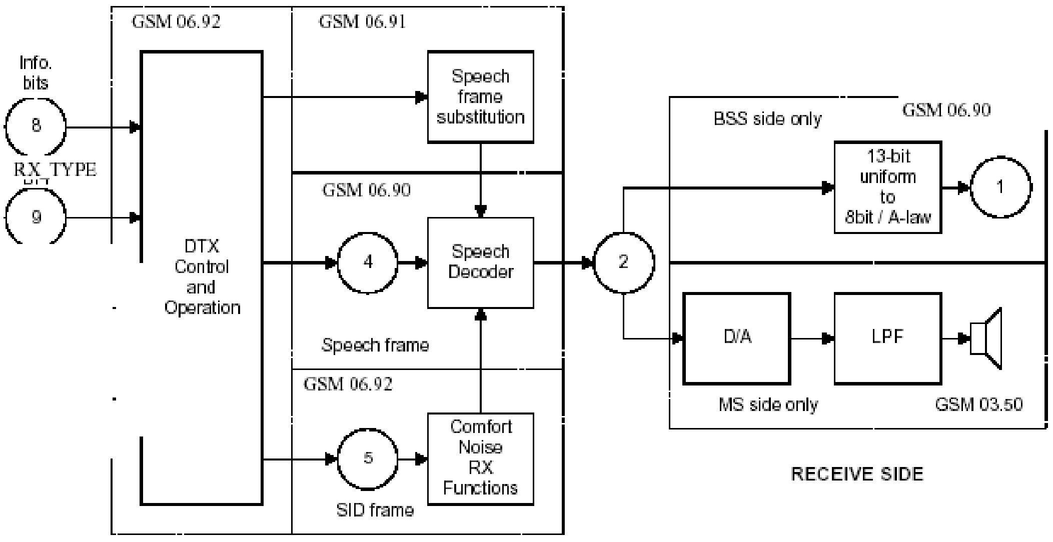


Figure 2.1 (b): Overview of audio processing functions (receive side)

1) 8-bit A-law or μ-law PCM (ITU-T recommendation G.711), 8 000 samples/s;
2) 13-bit uniform PCM, 8 000 samples/s;
3) Voice Activity Detector (VAD) flag;
4) Encoded speech frame, 50 frames/s, number of bits/frame depending on the AMR codec mode;
5) Silence Descriptor (SID) frame;
6) TX_TYPE, 2 bits, indicates whether information bits are available and if they are speech or SID information;
7) Information bits delivered to the radio subsystem;
8) Information bits received from the radio subsystem;
9) RX_TYPE, the type of frame received quantized into three bits

As shown in figure 2.1 (a), the speech encoder takes its input as a 13-bit uniform Pulse Code Modulated (PCM) signal either from the audio part of the Mobile Station or on the network side, from the Public Switched Telephone Network (PSTN) via an 8-bit A-law or μ-law to 13-bit uniform PCM conversion. The encoded speech at the output of the speech encoder is then delivered to the channel coding function.

In the receive direction, the inverse operations take place. GSM 06.90 describes the detailed mapping between input blocks of 160 speech samples in 13-bit uniform PCM format to encoded blocks (in which the number of bits depends on the used codec mode) and from these to output blocks of 160 reconstructed speech samples.

2.3 GSM AMR speech encoder

2.3.1 Principles

The AMR codec uses eight source codecs with bit-rates of 12.2, 10.2, 7.95, 7.40, 6.70, 5.90, 5.15 and 4.75 kbit/s [15]. The codec is based on the code-excited linear predictive (CELP) coding model. A 10th order linear prediction (LP), or short-term, synthesis filter is used which is given by:

$$H(z) = \frac{1}{\hat{A}(z)} = \frac{1}{1 + \sum_{i=1}^{m} \hat{a}_i z^{-i}},$$

where $\hat{a}_i$, $i = 1,...,m$, are the (quantified) linear prediction (LP) parameters, and $m = 10$ is the predictor order. The long-term, or pitch, synthesis filter is given by:

$$\frac{1}{B(z)} = \frac{1}{1 - g_p z^{-T}},$$

where T is the pitch delay and g_p is the pitch gain. The pitch synthesis filter is implemented using the so-called adaptive codebook approach.

The CELP speech synthesis model is shown in figure 2.2. In this model, the excitation signal at the input of the short-term LP synthesis filter is constructed by adding two excitation vectors from adaptive and fixed (innovative) codebooks. The speech is synthesized by feeding the two properly chosen vectors from these codebooks through the short-term synthesis filter.

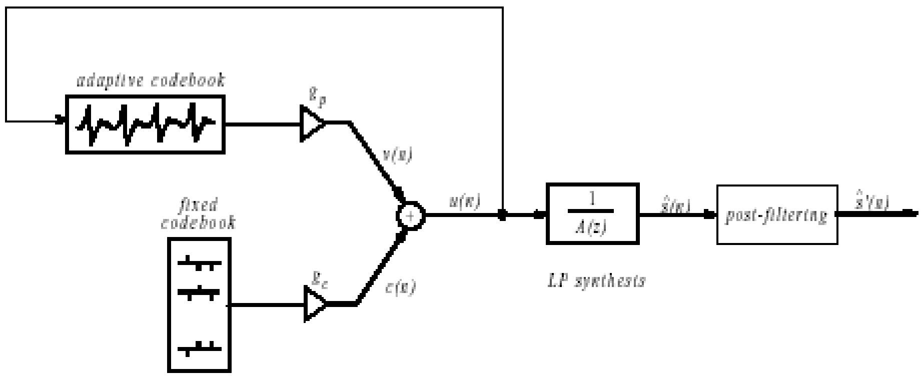


Figure 2.2: Simplified block diagram of the CELP synthesis model

The optimum excitation sequence in a codebook is chosen using an analysis-by-synthesis search procedure in which the error between the original and synthesized speech is minimized according to a perceptually weighted distortion measure. The perceptual weighting filter used in the analysis-by-synthesis search technique is given by:

$$W(z) = \frac{A(z/\gamma_1)}{A(z/\gamma_2)},$$

where *A(z)* is the unquantized LP filter and $0 < \gamma_2 < \gamma_1 \leq 1$ are the perceptual weighting factors. The values $\gamma_1 = 0.9$ (for the 12.2 and 10.2 kbit/s mode) or $\gamma_1 = 0.94$ (for all other modes) and $\gamma_2 = 0.6$ are used. The weighting filter uses the unquantized LP parameters.

The coder operates on speech frames of 20 ms corresponding to 160 samples at the sampling frequency of 8000 sample/s. At each 160 speech samples, the speech signal is analysed to extract the parameters of the CELP model (LP filter coefficients, adaptive and fixed codebooks' indices and gains). These parameters are encoded and transmitted. At the decoder, these parameters are decoded and speech is synthesized by filtering the reconstructed excitation signal through the LP synthesis filter.

The signal flow at the encoder is shown in figure 2.3.

LP analysis is performed twice per frame for the 12.2 kbit/s mode and once for the other modes. For the 12.2 kbit/s mode, the two sets of LP parameters are converted to line spectrum pairs (LSP) and jointly quantized using split matrix quantization (SMQ) with 38 bits. For the other modes, the single set of LP parameters is converted to line spectrum pairs (LSP) and vector quantized using split vector quantization (SVQ). The speech frame is divided into 4 subframes of 5 ms each (40 samples). The adaptive and fixed codebook parameters are transmitted every subframe. The quantized and unquantized LP parameters or their interpolated versions are used depending on the subframe. An open-loop pitch lag is estimated in every other subframe (except for the 5.15 and 4.75 kbit/s modes for which it is done once per frame) based on the perceptually weighted speech signal.

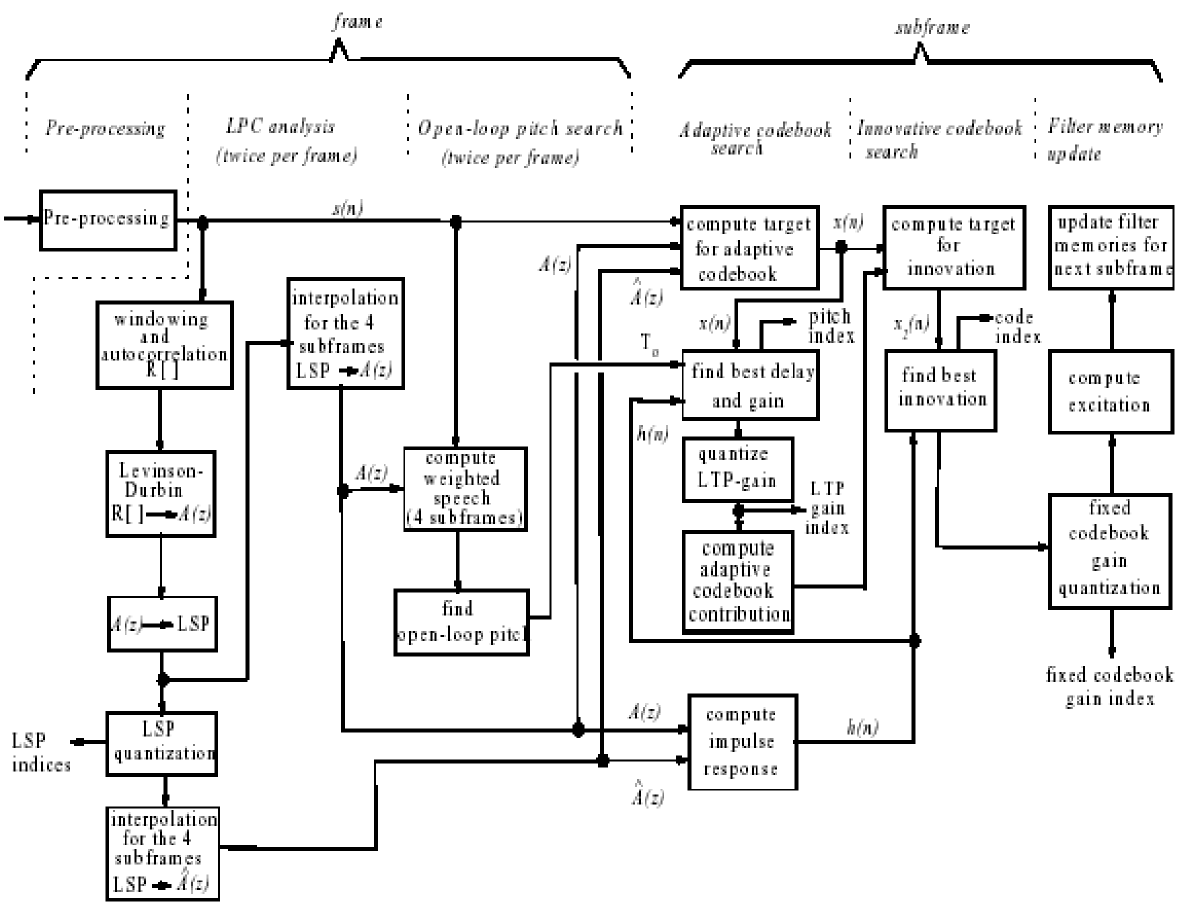

frame
subframe
Pre-processing
LPC analysis (twice per frame)
Open-loop pitch search (twice per frame)
Adaptive codebook search
Innovative codebook search
Filter memory update
Pre-processing
s(n)
windowing and autocorrelation R[ ]
interpolation for the 4 subframes LSP → A(z)
Levinson-Durbin R[ ] → A(z)
compute weighted speech (4 subframes)
A(z) → LSP
find open-loop pitch
LSP quantization
LSP indices
interpolation for the 4 subframes LSP → Â(z)
compute target for adaptive codebook
x(n)
compute target for innovation
update filter memories for next subframe
pitch index
code index
T0
find best delay and gain
find best innovation
compute excitation
quantize LTP-gain
LTP gain index
compute adaptive codebook contribution
fixed codebook gain quantization
fixed codebook gain index
compute impulse response
h(n)
A(z)
Â(z)
x2(n)

Figure 2.3: Block diagram of the GSM adaptive multi-rate encoder

Then the following operations are repeated for each subframe:

- The target signal *x(n)* is computed by filtering the LP residual through the weighted synthesis filter *W(z)H(z)* with the initial states of the filters having been updated by filtering the error between LP residual and excitation (this is equivalent to the common approach of subtracting the zero input response of the weighted synthesis filter from the weighted speech signal).

- The impulse response, *h(n)* of the weighted synthesis filter is computed.

- Closed-loop pitch analysis is then performed (to find the pitch lag and gain), using the target *x(n)* and impulse response *h(n)*, by searching around the open-loop pitch lag. Fractional pitch with 1/6th or 1/3rd of a sample resolution (depending on the mode) is used.

- The target signal *x(n)* is updated by removing the adaptive codebook contribution (filtered adaptive codevector), and this new target, $x_2(n)$, is used in the fixed algebraic codebook search (to find the optimum innovation).

- The gains of the adaptive and fixed codebook are scalar quantified with 4 and 5 bits respectively or vector quantified with 6-7 bits (with moving average (MA) prediction applied to the fixed codebook gain).

- Finally, the filter memories are updated (using the determined excitation signal) for finding the target signal in the next subframe.

The bit allocation of the AMR codec modes is shown in table 2.1. In each 20 ms speech frame, 95, 103, 118, 134, 148, 159, 204 or 244 bits are produced, corresponding to a bit-rate of 4.75, 5.15, 5.90, 6.70, 7.40, 7.95, 10.2 or 12.2 kbit/s. Note that the most significant bits (MSB) are always sent first.

Mode	Parameter	1st subframe	2nd subframe	3rd subframe	4th subframe	total per frame
12.2 kbit/s (GSM EFR)	2 LSP sets					38
	Pitch delay	9	6	9	6	30
	Pitch gain	4	4	4	4	16
	Algebraic code	35	35	35	35	140
	Codebook gain	5	5	5	5	20
	Total					244
10.2 kbit/s	LSP set					26
	Pitch delay	8	5	8	5	26
	Algebraic code	31	31	31	31	124
	Gains	7	7	7	7	28
	Total					204
7.95 kbit/s	LSP sets					27
	Pitch delay	8	6	8	6	28
	Pitch gain	4	4	4	4	16
	Algebraic code	17	17	17	17	68
	Codebook gain	5	5	5	5	20
	Total					159
7.40 kbit/s (DAMPS EFR)	LSP set					26
	Pitch delay	8	5	8	5	26
	Algebraic code	17	17	17	17	68
	Gains	7	7	7	7	28
	Total					148
6.70 kbit/s	LSP set					26
	Pitch delay	8	4	8	4	24
	Algebraic code	14	14	14	14	56
	Gains	7	7	7	7	28
	Total					134
5.90 kbit/s	LSP set					26
	Pitch delay	8	4	8	4	24
	Algebraic code	11	11	11	11	44
	Gains	6	6	6	6	24
	Total					118
5.15 kbit/s	LSP set					23
	Pitch delay	8	4	4	4	20
	Algebraic code	9	9	9	9	36
	Gains	6	6	6	6	24
	Total					103
4.75 kbit/s	LSP set					23
	Pitch delay	8	4	4	4	20
	Algebraic code	9	9	9	9	36
	Gains	8		8		16
	Total					95

Table 2.1: Bit allocation of the AMR coding algorithm for 20ms frame

2.3.2 Pre-processing

Two pre-processing functions are applied prior to the encoding process: high-pass filtering and signal down-scaling.

Down-scaling consists of dividing the input by a factor of 2 to reduce the possibility of overflows in the fixed-point implementation. The high-pass filter serves as a precaution against undesired low frequency components. A filter with a cut off frequency of 80 Hz is used.

2.3.3 Linear prediction analysis and quantization

12.2 kbit/s mode

Short-term prediction, or linear prediction (LP), analysis is performed twice per speech frame using the auto-correlation approach with 30 ms asymmetric windows. No lookahead is used in the auto-correlation computation.

The auto-correlations of windowed speech are converted to the LP coefficients using the Levinson-Durbin algorithm. Then the LP coefficients are transformed to the Line Spectral Pair (LSP) domain for quantization and interpolation purposes. The interpolated quantified and unquantized filter coefficients are converted back to the LP filter coefficients (to construct the synthesis and weighting filters at each subframe).

10.2, 7.95, 7.40, 6.70, 5.90, 5.15, 4.75 kbit/s modes

Short-term prediction, or linear prediction (LP), analysis is performed once per speech frame using the auto-correlation approach with 30 ms asymmetric windows. A look- ahead of 40 samples (5 ms) is used in the auto-correlation computation.

The auto-correlations of windowed speech are converted to the LP coefficients using the Levinson-Durbin algorithm. Then the LP coefficients are transformed to the Line Spectral Pair (LSP) domain for quantization and interpolation purposes. The interpolated quantified and unquantized filter coefficients are converted back to the LP filter coefficients (to construct the synthesis and weighting filters at each subframe).

2.3.4 Open-loop pitch analysis

Open-loop pitch analysis is performed in order to simplify the pitch analysis and confine the closed-loop pitch search to a small number of lags around the open-loop estimated lags.

Open-loop pitch estimation is based on the weighted speech signal s_w *(n)* which is obtained by filtering the input speech signal through the weighting filter

$$W(z) = A(z/\gamma_1)/A(z/\gamma_2).$$

That is, in a subframe of size *L* , the weighted speech is given by:

$$s_w(n) = s(n) + \sum_{i=1}^{10} a_i \gamma_1^i s(n-i) - \sum_{i=1}^{10} a_i \gamma_2^i s_w(n-i), \quad n = 0,\ldots,L-1$$

2.3.5 Impulse response computation

The impulse response, *h(n)* , of the weighted synthesis filter

$$H(z)W(z) = A(z/\gamma_1)/[\hat{A}(z)A(z/\gamma_2)]$$

is computed each subframe. This impulse response is needed for the search of adaptive and fixed codebooks. The impulse response *h(n)* is computed by filtering the vector of coefficients of the filter $A(z/\gamma_1)$ extended by zeros through the two filters $1/\hat{A}(z)$ and $1/A(z/\gamma_2)$.

2.3.6 Target signal computation

The target signal for adaptive codebook search is usually computed by subtracting the zero input response of the weighted synthesis filter from the weighted speech signal $s_w(n)$. This is performed on a subframe basis.

An equivalent procedure for computing the target signal, which is used in this standard, is the filtering of the LP residual signal $res_{LP}(n)$ through the combination of synthesis filter $1 / \hat{A}(z)$ and the weighting filter $A(z/\gamma_1)\ A(z/\gamma_2)$.

After determining the excitation for the subframe, the initial states of these filters are updated by filtering the difference between the LP residual and excitation.

The residual signal $res_{LP}(n)$ which is needed for finding the target vector is also used in the adaptive codebook search to extend the past excitation buffer. This simplifies the adaptive codebook search procedure for delays less than the subframe size of 40 as will be explained in the next clause. The LP residual is given by:

$$res_{LP}(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i s(n-i).$$

2.3.7 Adaptive codebook

Adaptive codebook search is performed on a subframe basis. It consists of performing closed-loop pitch search, and then computing the adaptive codevector by interpolating the past excitation at the selected fractional pitch lag.

The adaptive codebook parameters (or pitch parameters) are the delay and gain of the pitch filter. In the adaptive codebook approach for implementing the pitch filter, the excitation is repeated for delays less than the subframe length. In the search stage, the excitation is extended by the LP residual to simplify the closed-loop search.

The average adaptive codebook gain is calculated if the *LSP_flag* is set and the unquantized adaptive codebook gain exceeds the gain threshold GP_{th}= 0.95 .

The average gain is calculated from the present unquantized gain and the quantized gains of the seven previous subframes. If the average adaptive codebook gain exceeds the GP_{th} , the unquantized gain is limited to the threshold value and the *GpC_flag* is set to indicate the limitation.

2.3.8 Algebraic codebook

The algebraic codebook is searched by minimizing the mean square error between the weighted input speech and the weighted synthesized speech. The target signal used in the closed-loop pitch search is updated by subtracting the adaptive codebook contribution.

The algebraic structure of the codebooks allows for very fast search procedures since the innovation vector contains only a few nonzero pulses.

2.3.9 Quantization of the adaptive and fixed codebook gains

If the *GpC_flag* is set, the limited adaptive codebook gain is used in the gain quantization. The quantization codebook search range is limited to only include adaptive codebook gain values less than GP_{th} . This is performed in the quantization search for all modes.

The fixed codebook gain quantization is performed using MA prediction with fixed coefficients. The 4th order MA prediction is performed on the innovation energy.

After the gain quantization, the buffer with past adaptive codebook gains is updated, regardless of the value of the *GpC_flag*.

2.3.10 Memory update

An update of the states of the synthesis and weighting filters is needed in order to compute the target signal in the next subframe.

2.4 GSM AMR speech decoder

2.4.1 Principles

The signal flow at the decoder is shown in figure 2.4. At the decoder, based on the chosen mode, the transmitted parameters are decoded from the received bitstream at each transmission frame [15]. These parameters are the LSP vectors, the fractional pitch lags, the innovative codevectors, and the pitch and innovative gains. The LSP vectors are converted to the LP filter coefficients and interpolated to obtain LP filters at each subframe. Then, at each 40-sample subframe:

- the excitation is constructed by adding the adaptive and innovative codevectors scaled by their respective gains;
- the speech is reconstructed by filtering the excitation through the LP synthesis filter.

Finally, the reconstructed speech signal is passed through an adaptive postfilter.

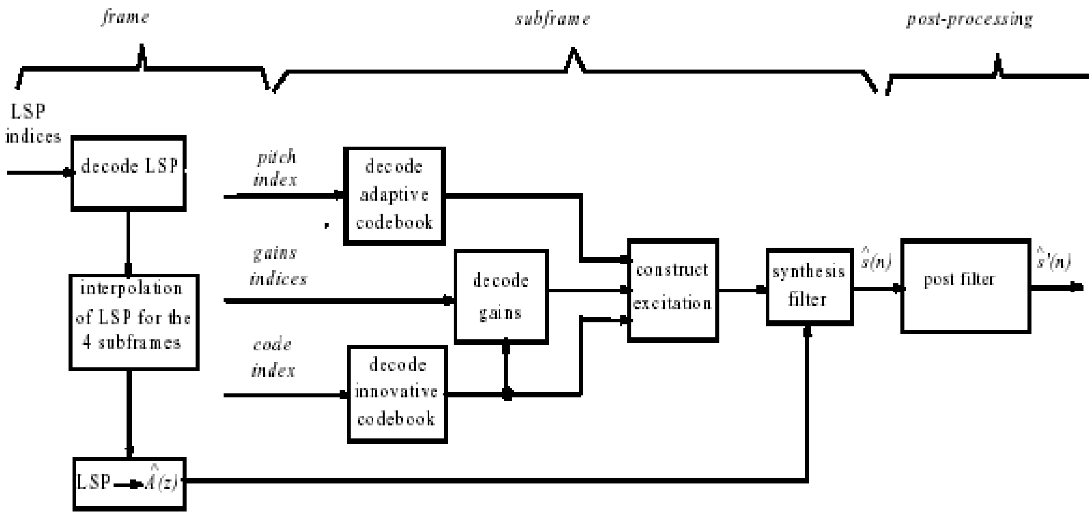


Figure 2.4: Block diagram of the GSM adaptive multi-rate decoder

2.4.2 Decoding and speech synthesis

The decoding process begins with the decoding of LP filter parameters: the received indices of LSP quantization are used to reconstruct the quantified LSP vectors. An interpolation is then performed to obtain 4 interpolated LSP vectors (corresponding to 4 subframes). For each subframe, the interpolated LSP vector is converted to LP filter coefficient domain a_k , which is used for synthesizing the reconstructed speech in the subframe.

The following steps are repeated for each subframe:

1) **Decoding of the adaptive codebook vector:** The received pitch index (adaptive codebook index) is used to find the integer and fractional parts of the pitch lag. The adaptive codebook vector *v(n)* is found by interpolating the past excitation *u(n)* (at the pitch delay).

2) **Decoding of the innovative codebook vector:** The received algebraic codebook index is used to extract the positions and amplitudes (signs) of the excitation pulses and to

find the algebraic codevector *c(n)* . If the integer part of the pitch lag, *T*, is less than the subframe size 40, the pitch sharpening procedure is applied.

3) **Decoding of the adaptive and fixed codebook gains:** In case of scalar quantization of the gains (12.2 kbit/s and 7.95 kbit/s modes) the received indices are used to readily find the quantified adaptive codebook gain and the quantified fixed codebook gain correction factor from the corresponding quantization tables. In case of vector quantization of the gains (all other modes), the received index gives both the quantified adaptive codebook gain and the quantified fixed codebook gain correction factor.

4) **Smoothing of the fixed codebook gain (10.2, 6.70, 5.90, 5.15, 4.75 kbit/s modes):** An adaptive smoothing of the fixed codebook gain is performed to avoid unnatural fluctuations in the energy contour. The smoothing is based on a measure of the stationarity of the short-term spectrum in the **q** domain. The smoothing strength is computed from this measure.

5) **Anti-sparseness processing (7.95, 6.70, 5.90, 5.15, 4.75 kbit/s modes):** An adaptive anti-sparseness postprocessing procedure is applied to the fixed codebook vector *c(n)* in order to reduce perceptual artifacts arising from the sparseness of the algebraic fixed codebook vectors with only a few non-zero samples per subframe. The anti-sparseness processing consists of circular convolution of the fixed codebook vector with an impulse response. Three pre-stored impulse responses are. The selection of the impulse response is performed adaptively from the adaptive and fixed codebook gains.

6) **Computing the reconstructed speech:** The excitation at the input of the synthesis filter is given by:

$$u(n) = \hat{g}_p v(n) + \hat{g}_c c(n).$$

Before the speech synthesis, a post-processing of excitation elements is performed. This means that the total excitation is modified by emphasizing the contribution of the adaptive codebook vector.

Adaptive gain control (AGC) is used to compensate for the gain difference between the non-emphasized excitation and emphasized excitation.

7) **Additional instability protection**: An additional instability protection is implemented in the speech decoder which is monitoring overflows in the synthesis filter. If an overflow has occurred in the synthesis part, the whole adaptive codebook memory is scaled down by a factor of 4, and the synthesis filtering is repeated using this down-scaled memory.

2.4.3 Adaptive post-filtering

The adaptive postfilter is the cascade of two filters: a formant postfilter, and a tilt compensation filter. The postfilter is updated every subframe of 5 ms. The formant postfilter is given by:

$$H_f(z) = \frac{\hat{A}(z/\gamma_n)}{\hat{A}(z/\gamma_d)}$$

where *$\hat{A}(z)$* is the received quantified (and interpolated) LP inverse filter (LP analysis is not performed at the decoder), and the factors γ_n and γ_d control the amount of the formant postfiltering.

Finally, the filter *$H_t(z)$* compensates for the tilt in the formant postfilter *$H_f(z)$* and is given by:

$$H_t(z) = 1 - \mu z^{-1}$$

where μ is a tilt factor.

2.4.4 High-pass filtering and up-scaling

The high-pass filter serves as a precaution against undesired low frequency components. A filter cut-off frequency of 60 Hz is used.

Up-scaling consists of multiplying the post-filtered speech by a factor of 2 to compensate for the down-scaling by 2 which is applied to the input signal.

2.5 Support functions

2.5.1 Discontinuous transmission (DTX)

During a normal phone conversation, the participants alternate so that, on the average, each direction of transmission is occupied about 50 % of the time. Discontinuous transmission (DTX) is a mode of operation where the transmitters are switched on only for those frames which contain useful information. This may be done for the following two purposes:

1) in the MS, battery life will be prolonged or a smaller battery could be used for a given operational duration;
2) the average interference level over the air interface is reduced, leading to better Radio Frequency (RF) spectrum efficiency.

The overall DTX mechanism is implemented in the DTX handlers (Transmit (TX) and Receive (RX)) and requires the following functions [18]:

- a Voice Activity Detector (VAD) on the TX side;
- evaluation of the background acoustic noise on the TX side, in order to transmit characteristic parameters to the RX side;

- generation of comfort noise on the RX side during periods where the radio transmission is turned off.

The transmission of comfort noise information to the RX side is achieved by means of a Silence Descriptor (SID) frame. A SID frame is transmitted at the end of speech bursts and serves as an end of speech marker for the RX side. In order to update the comfort noise characteristics at the RX side, SID frames are transmitted at regular intervals also during speech pauses. This also serves the purpose of improving the measurement of the radio link quality by the radio subsystem (RSS).

The 2 bit field TX_TYPE indicates whether information bits are speech or SID information or if there is no information. The TX_TYPE field is calculated from the VAD flag by the TX DTX handler. When SID information is transmitted the operation of the speech encoder is modified to reduce the remaining computation for that frame.

2.5.2 Voice Activity Detection (VAD)

The input to the VAD is a set of parameters computed by the adaptive multi-rate speech encoder defined in GSM 06.90 [19]. The VAD uses this information to decide whether each 20 ms speech coder frame contains speech or not. Note that the VAD flag is an input to the TX DTX handler and does not need to control the transmitter keying directly.

2.5.3 Comfort noise insertion

When switching the transmission on and off during DTX operation, the effect would be a modulation of the background noise at the receiving end, if no precautions were taken. When transmission is on, the background noise is transmitted together with the speech to the receiving end. As the speech burst ends, the connection is off and the perceived noise would drop to a very low level. This step modulation of noise may be perceived as annoying and reduce the intelligibility of speech, if presented to a listener without modification.

This "noise contrast effect" is reduced in the GSM system by inserting an artificial noise, termed comfort noise, at the receiving end when speech is absent. The comfort noise processes are as follows [17]:

- the evaluation of the acoustic background noise in the transmitter
- the noise parameter encoding (SID frames) and decoding;
- and the generation of comfort noise in the receiver.

2.5.4 Lost speech frame substitution and muting

In the receiver, frames may be lost due to transmission errors or frame stealing. Some actions have to be taken in these cases, both for lost speech frames and for lost SID frames in DTX operation.

In order to mask the effect of an isolated lost frame, the lost speech frame is substituted by a predicted frame based on previous frames. Insertion of silence frames is not allowed. For several subsequent lost frames, a muting technique shall be used to indicate to the listener that transmission has been interrupted. [16]

2.5.5 Adaptive multi-rate codec homing

The GSM adaptive multi-rate speech transcoder, VAD, DTX system and comfort noise parts of the audio processing functions (shown in figures 2.1 (a) and (b)) are defined in bit exact arithmetic. Consequently, they shall react on a given input sequence always with the corresponding bit exact output sequence, provided that the internal state variables are also always exactly in the same state at the beginning of the experiment.

The input test sequences provided in GSM 06.74 shall force the corresponding output test sequences, provided that the tested modules are in their home-state when starting. The modules may be set into their home states by provoking the appropriate homing-functions. This is normally done during reset (initialization of the codec).

Special in band signaling frames (encoder-homing-frame and decoder-homing-frame) have been defined to provoke these homing-functions also in remotely placed modules. This mechanism is specified to support three main areas:

- type approval of mobile terminal equipment;
- type approval of infrastructure equipment;
- remote control and testing for operation and maintenance.

At the end of the first received homing frame, the audio functions that are defined in a bit exact way shall go into their predefined home states. The output corresponding to the first homing frame is dependent on the codec state when the frame was received. Any consecutive homing frames shall produce corresponding homing frames at the output.

Chapter 3

Workspace: hardware and software development tools

3.1 General description

As we mentioned in the introduction chapter, the purpose of the project was the implementation of the AMR vocoder into the ADSP-2181 processor from Analog Devices. Furthermore, we would have to accomplish, if possible, two objectives: it should be a "bit-exact" implementation and it had to run in "real-time". Let's study what kind of material we needed so as to begin with the implementation.

The vocoder description was provided by ETSI specifications. All the related documentation was available in their web site [23]. In the "ETSI EN 301 704" standard and the related documents, a complete description of the vocoder can be found (and we dedicated the whole previous chapter to study it), but, concerning the implementation, the more important document was the "ETSI EN 301 712: ANSI-C code for the AMR speech codec" [20], which provided a complete C language implementation of the vocoder. It had to be the key reference in the process of implementing the vocoder into the ADSP-2181. Finally, if a bit-exact implementation was to be accomplished, some way of verifying that exactitude was necessary, a set of speech samples to test the vocoder in all possible situations, and their coded and decoded results to compare with the obtained with our implementation.

That is exactly what we got with “ETSI EN 301 713: Test sequences for the Adaptive Multi-Rate (AMR) speech codec” [21].

With all the written material, the next step was to get the necessary software tools. C code had been provided by ETSI, so a C compiler was required, as it would be necessary to run the test vectors on the official implementation of the vocoder and extract intermediate results at specific points. Those results could then serve as a reference to compare them with the results obtained from the assembling code. A good debugging environment would also be appreciated, to set breakpoints in the code and be able to examine the values of different variables. The Visual Studio 6.0, from Microsoft Corporation, was chosen. It included Visual C++ (compatible with ANSI C) and also a great deal of debugging tools.

Finally, we needed the digital signal processor itself or, better, a simulation software close enough to it that let us work most of the time fast and easily on the PC. The ADSP-2181 is supported with a complete set of software and hardware development tools. Specifically, the EZ-KIT Lite provided by Analog Devices contained all we needed [7]:

- System Builder—Defines the architecture of the hardware system.
- Assembler—Assembles the source code and data modules (there was no specific source code editor, so a standard editor was used to write the code). It produces object and some other intermediate files, as shown in figure 3.1:

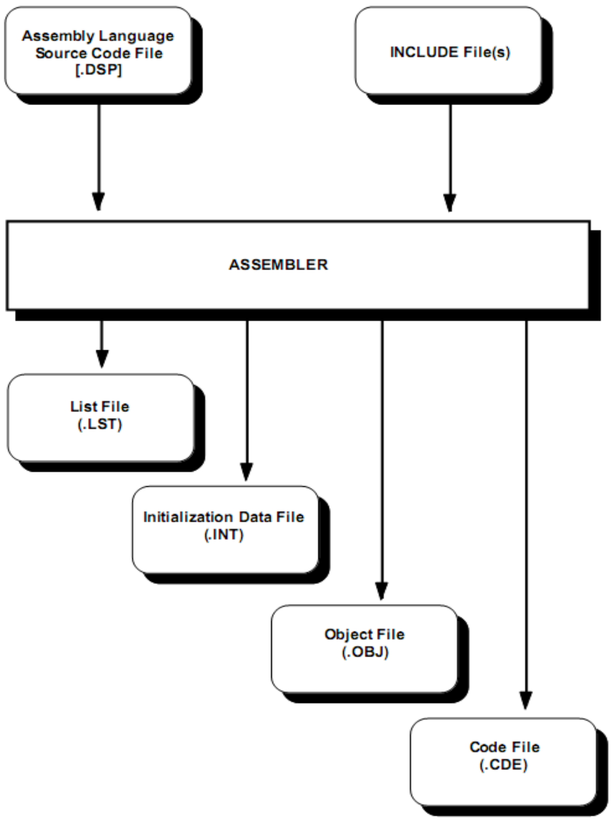


Figure 3.1: Assembler process

- Linker—Links separately assembled modules. It maps the linked code and data output to the target system hardware, as specified by the System Builder output. It is the module that produces the executable, as shown in figure 3.2:

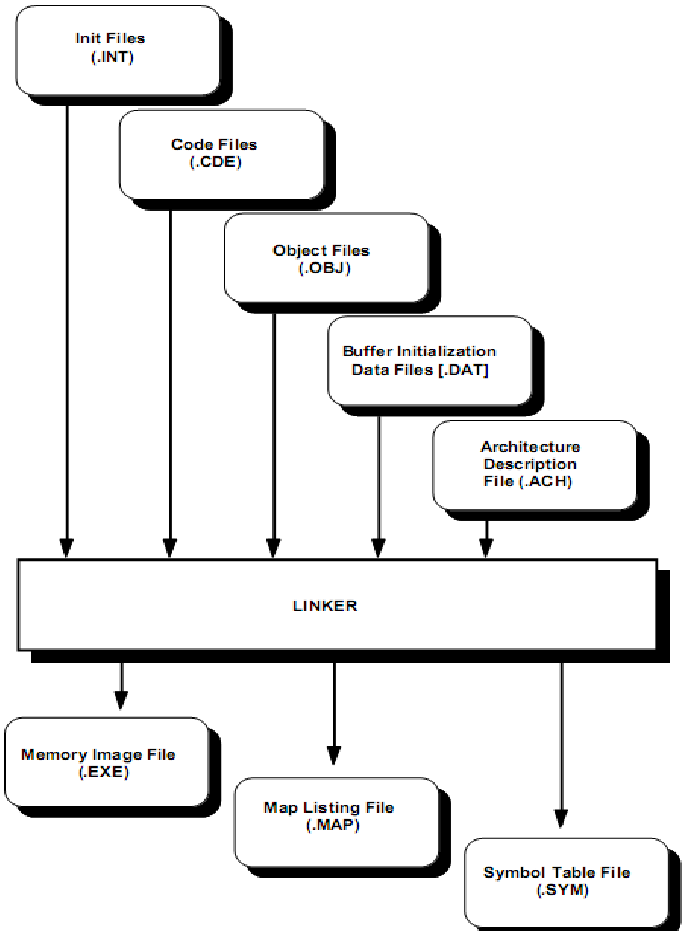


Figure 3.2: Linker process

- VisualDSP—Runs, debugs and analyzes the executable program. Its importance during the development was so high that we will dedicate chapter 3.3 to examine this tool in depth.
- PROM Splitter—This module reads the Linker output and generates PROM programmer compatible files.
- ADSP-2181 EZ-LAB evaluation board—The hardware component, with the ADSP-2181 processor, where the PROM file can be loaded and the program can physically run.

3.2 ADSP-2181 processor from Analog Devices

3.2.1 General description

The ADSP-2181 is a programmable single-chip microprocessor optimized for digital signal processing (DSP) and other high-speed numeric processing applications. It belongs to the ADSP-2100 processors family from Analog Devices. It combines the ADSP-2100 family base architecture (three computational units, data address generators and a program sequencer) with two serial ports, a 16-bit internal DMA port, a byte DMA port, a programmable timer, Flag I/O, extensive interrupt capabilities, and on-chip program and data memory.

The ADSP-2181 integrates 80K bytes of on-chip memory configured as 16K words (24-bit) of program RAM, and 16K words (16-bit) of data RAM.

Fabricated in a high speed, double metal, low power, CMOS process, the ADSP-2181 operates with a 25 ns instruction cycle time. Every instruction can execute in a single processor cycle [5].

3.2.2 Architecture

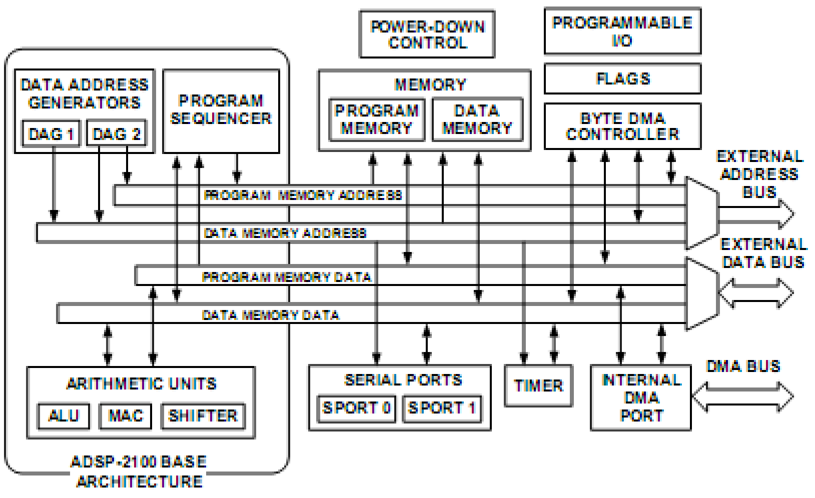


Figure 3.3: ADSP-2181 architecture

- Computational Units—Every processor in the ADSP-2100 family contains three independent, full-function computational units: an arithmetic/logic unit (ALU), a multiplier/accumulator (MAC) and a barrel shifter. The computational units process 16-bit data directly and also provide hardware support for multi precision computations.

- Data Address Generators & Program Sequencer—Two dedicated address generators and a program sequencer supply addresses for on-chip or external memory access. The sequencer supports single-cycle conditional branching and executes program loops with zero overhead. Dual data address generators allow the processor to generate simultaneous addresses for dual operand fetches. Together the sequencer and data address generators keep the computational units continuously working, maximizing throughput.

- Memory—The ADSP-2100 family uses a modified Harvard architecture in which data memory stores data, and program memory stores both instructions and data. All ADSP-2100 family processors contain on-chip RAM that comprises a portion of the program memory space and data memory space. The speed of the on-chip memory allows the processor to fetch two operands (one from data memory and one from program memory) and an instruction (from program memory) in a single cycle.

- Buses—The processor has five internal buses. The program memory address (PMA) and data memory address (DMA) buses are used internally for the addresses associated with program and data memory. The program memory data (PMD) and data memory data (DMD) buses are used for the data associated with the memory spaces. The buses are multiplexed into a single external address bus and a single external data bus; the BMS, DMS and PMS signals select the different address spaces. The R bus transfers intermediate results directly between the various computational units.

- Serial Ports—The serial ports (SPORTs) provide a complete serial interface with hardware companding for data compression and expansion. Both μ-law and A-law companding are supported. The SPORTs interface easily and directly to a wide variety of popular serial devices. Each SPORT can generate a programmable internal clock or accept an external clock. SPORT0 includes a multichannel option.

- Timer—A programmable timer/counter with 8-bit prescaler provides periodic interrupt generation.

- DMA Ports—The ADSP-2181's Internal DMA Port (IDMA) and Byte DMA Port (BDMA) provide efficient data transfers to and from internal memory. The IDMA port has a 16-bit multiplexed address and data bus and supports 24-bit program memory. The IDMA port is completely asynchronous and can be written to while the ADSP-2181 is operating at full speed. The byte memory DMA port allows boot loading and storing of program instructions and data.

The ADSP-2181 architecture exhibits a high degree of parallelism, tailored to DSP requirements. In a single cycle, it can:

- Generate the next program address.
- Fetch the next instruction.
- Perform one or two data moves.
- Update one or two data address pointers.
- Perform a computation.
- Receive and/or transmit data via the serial ports.
- Receive and/or transmit via the DMA ports.

3.2.3 Instruction Set

The assembly language of the ADSP-2181 is often referred as simply "ADI". It's a language that, unlike many other assembly languages, uses an algebraic syntax for readability and ease of coding. Because some assembly code will be shown in chapter 4, a brief description of the assembly language of the ADSP-2181 will be included in the Appendix B. Here, some features will be highlighted because they had an impact over the development of the project, or at least made it a little easier the always difficult task of programming in assembly language.

The instruction set is quite appropriate to the computation-intensive algorithms of the AMR vocoder. For example, sustained single-cycle multiplication/accumulation operations are possible. The instruction set provides full control of the processors' three computational units: the ALU, MAC and Shifter. Arithmetic instructions can process single-precision 16-bit operands directly and provisions for multi-precision operations are available. A 40-bit accumulator provides eight bits of protection against overflow in successive additions to ensure that no loss of data or range occurs; 256 overflows would have to occur before any data is lost [6].

There is no performance penalty for the high-level syntax of ADSP-2181 language: each program statement assembles into one 24-bit instruction which executes in a single cycle. There are no multi-cycle instructions in the instruction set.

In addition to JUMP and CALL, the instruction set's control instructions support conditional execution of most calculations and a DO UNTIL looping instruction. Return from interrupt (RTI) and return from subroutine (RTS) are also provided.

As a consequence of the high degree of parallelism of the ADSP-2181 architecture, seen in the previous section, and the 24-bit instruction words, the instruction set allows for single-cycle execution of any of the following combinations [5]:

- any ALU, MAC or Shifter operation (conditional or non-conditional)
- any register-to-register move
- any data memory read or write
- a computation with any data register to data register move

- a computation with any memory read or write
- a computation with a read from two memories

Lastly, but equally remarkable, ADI allows maximum flexibility moving data. It provides moves from any register to any other register, and from most registers to/from memory.

3.3 Running, debugging and analyzing the assembling code: VisualDSP

The implementation and testing of the vocoder required an absolute control over the data at every step of the execution process. Firstly, we needed a tool to execute the program without uploading it into the DSP itself, that is, in order to accelerate the testing process and being more productive we needed a PC simulator. Secondly, without the proper tool to debug the code it would have been impossible to accomplish the ”bit-exact” goal. If the program does not function correctly, the ability to step through code or to run to a predetermined line and halt are very useful features. Each time the program halts, you can examine registers and memory to determine the source of errors. From the typical programming errors to the, more difficult to detect, overflow situations, they all needed a step-by-step code revision to find the exact line of the assembly program where the error was generated. Lastly, the “real-time” implementation goal required the proper tool to analyze the processor use and performance. All three requirements were met by the tool we are now presenting: the VisualDSP debugger.

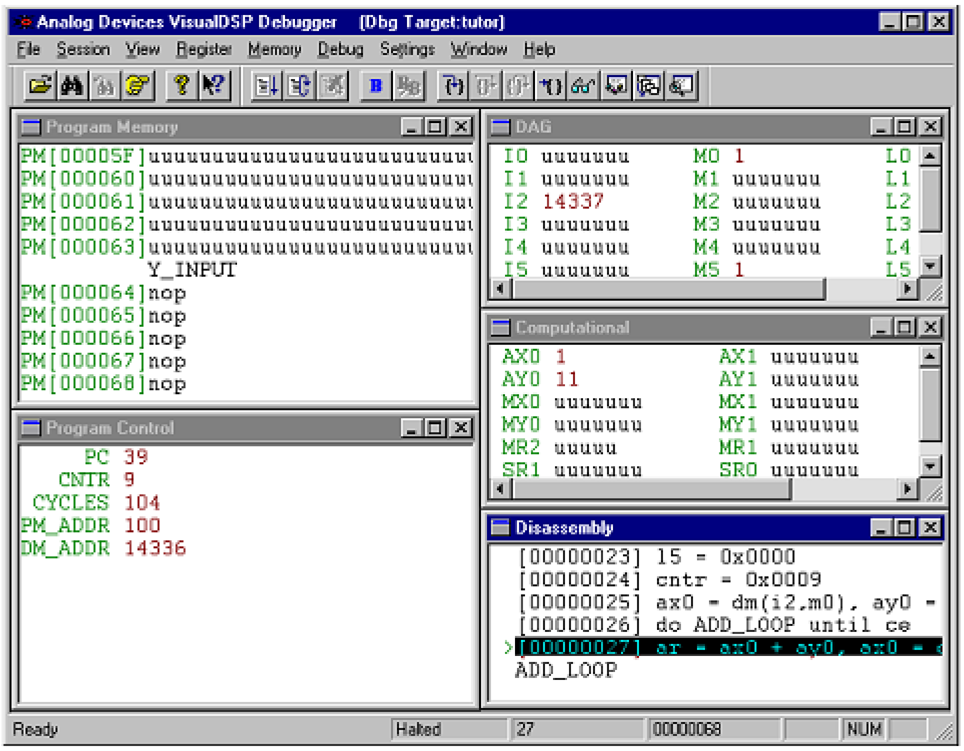
The VisualDSP debugger (which we'll often call “Simulator”) gives several options for running and observing program execution:

- Examining and changing the contents of memory and register windows
- Setting breakpoints to control program execution
- Defining watch points to capture program activity

- Tracing functions in the program to determine which paths are taken during specific activities
- Profiling the application's performance within defined memory ranges
- Streaming data from the test vector into the program to watch the results

3.3.1 VisualDSP interface

Figure 3.4 shows the distribution of VisualDSP debugger windows that was used most of the time during the development process.


Analog Devices VisualDSP Debugger [Dbg Target:tutor]
File Session View Register Memory Debug Settings Window Help
Program Memory
Program Control
DAG
Computational
Disassembly

```
PM[00005F]uuuuuuuuuuuuuuuuuuuuuuuuuuu
PM[000060]uuuuuuuuuuuuuuuuuuuuuuuuuuu
PM[000061]uuuuuuuuuuuuuuuuuuuuuuuuuuu
PM[000062]uuuuuuuuuuuuuuuuuuuuuuuuuuu
PM[000063]uuuuuuuuuuuuuuuuuuuuuuuuuuu
          Y_INPUT
PM[000064]nop
PM[000065]nop
PM[000066]nop
PM[000067]nop
PM[000068]nop
```

```
      PC  39
    CNTR  9
  CYCLES  104
 PM_ADDR  100
 DM_ADDR  14336
```

```
 I0  uuuuuuu      M0  1          L0
 I1  uuuuuuu      M1  uuuuuuu    L1
 I2  14337        M2  uuuuuuu    L2
 I3  uuuuuuu      M3  uuuuuuu    L3
 I4  uuuuuuu      M4  uuuuuuu    L4
 I5  uuuuuuu      M5  1          L5
```

```
AX0  1              AX1  uuuuuuu
AY0  11             AY1  uuuuuuu
MX0  uuuuuuu        MX1  uuuuuuu
MY0  uuuuuuu        MY1  uuuuuuu
MR2  uuuuu          MR1  uuuuuuu
SR1  uuuuuuu        SR0  uuuuuuu
```

```
  [00000023] 15 = 0x0000
  [00000024] cntr = 0x0009
  [00000025] ax0 = dm(i2,m0), ay0 =
  [00000026] do ADD_LOOP until ce
 >[00000027] ar = ax0 + ay0, ax0 =
 ADD_LOOP
```

```
Ready        Halted   27   00000068     NUM
```

Figure 3.4: Common VisualDSP interface

The "Disassembly" window shows the code of the program with the instruction currently executing highlighted.

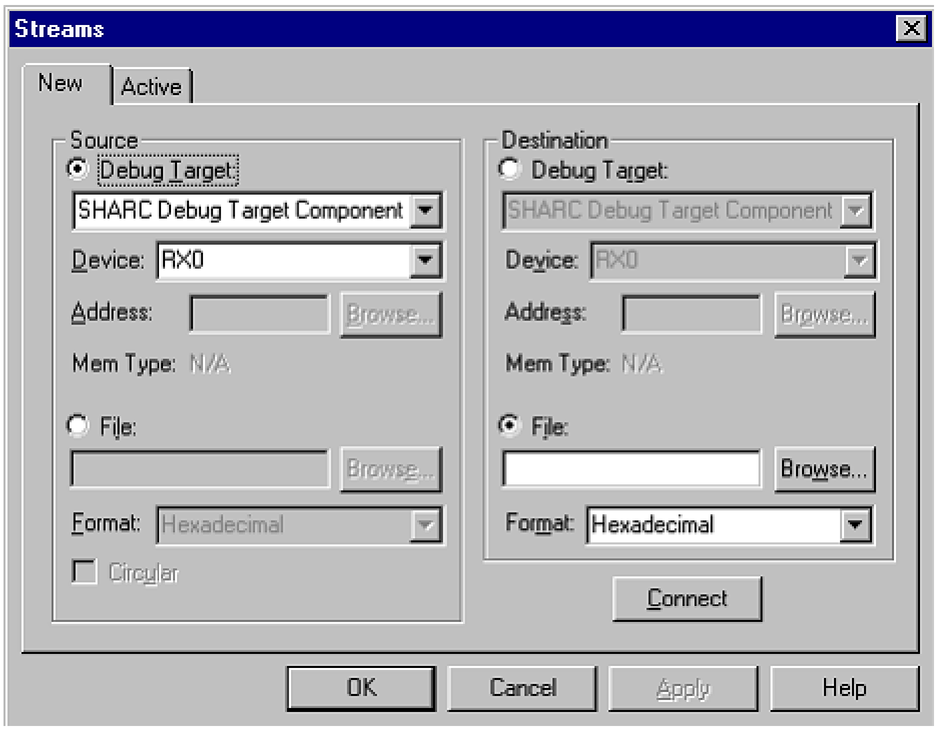
"Computational" and "DAG" are used to check registers status, which was very useful to trace the data changes involved in every action.

Another important window is “Program Control”. It shows some general indicators that help keep track of execution loops, for example.

Lastly, another not so often used, but also useful, windows include: “Program Memory” and “Data Memory” (to check at a very low level the content of any address in program or data memory) and “Call Stack” (to follow the execution path when using sub-routines).

3.3.2 I/O streaming

Most of the time, it was necessary to use data input (often one of the test vectors provided by ETSI) to follow the data processing throughout the program. It was then appropriate to use the “streaming” option of the debugger. As shown in the figure 3.5, there is an option to choose sources and a destinations for the data entering and exiting the processor. They can be physical devices or, as in our case, data files.


Streams

New | Active

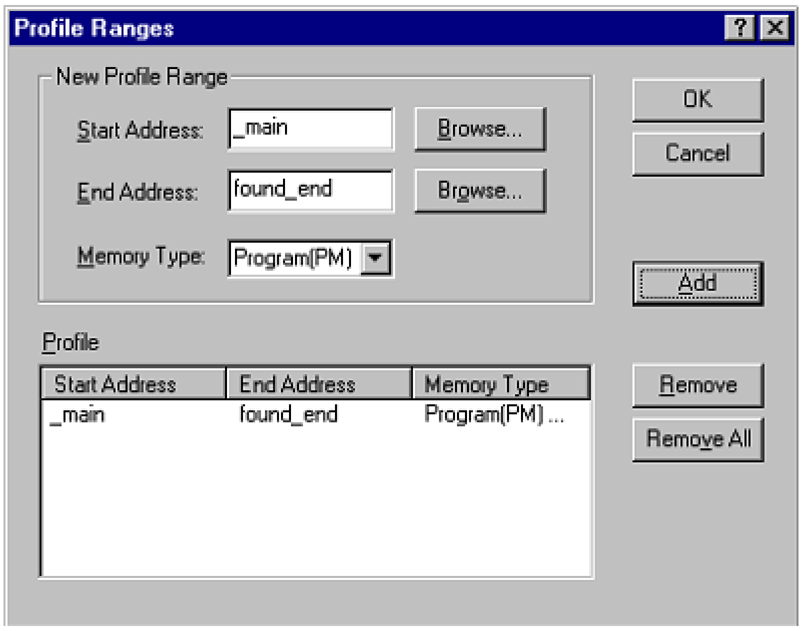
Source
[✓ Debug Target:]
SHARC Debug Target Component
Device: RX0
Address:
Mem Type: N/A
File:
Format: Hexadecimal
Circular

Destination
Debug Target:
SHARC Debug Target Component
Device: RX0
Address:
Mem Type: N/A
[✓ File:]
Format: Hexadecimal

Figure 3.5: Stream configuration

3.3.3 Profiling

To analyze the executable program runtime behavior, the “profiling” function was used. This feature allows to select areas of code and obtain its performance on program execution, as shown in figure 3.6.



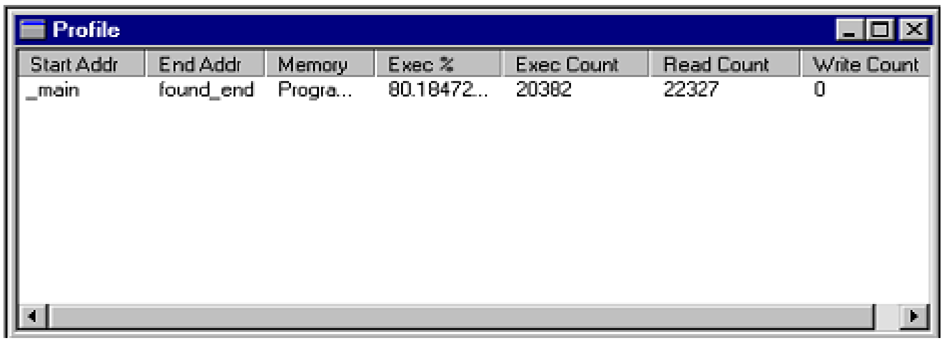
Start Address: _main
End Address: found_end
Memory Type: Program(PM)

Start Address	End Address	Memory Type
_main	found_end	Program(PM) ...

Figure 3.6: Profile definition

We were especially interested in seeing the processor cycles consumption of the vocoder, divided into its different modules. That would make it possible to evaluate the optimization level of the assembly code, and if we would be able to accomplish one of our main objectives: the real-time operation of the system.

Once a profile have been defined, the program (or the part of it we are interested in) can be run and the performance results are shown, as in figure 3.7.



Start Addr	End Addr	Memory	Exec %	Exec Count	Read Count	Write Count
_main	found_end	Progra...	80.18472...	20382	22327	0

Figure 3.7: Profiling results

The “Exec Count” parameter shows the number of executions of any instruction belonging to the profile, and the “Exec %” is the percentage over the total number of instructions executed. As we learned previously in this chapter, the processor executes any instruction in one cycle, so these parameters tell us the number of processor cycles used in any part of the program that we want to evaluate. We will come back to the performance issue in the next chapter, when we study some numeric results of the vocoder implementation.

Chapter 4

Implementation and optimization

4.1 Methodology

4.1.1 Preparation

A simple first view to the C source code provided by ETSI [20] made it clear it was not going to be an easy, nor short, development. The vocoder was implemented into 113 ".c" code files (with their respective ".h" header files) and 22 ".tab" files for the numeric tables. Further examining the ".c" files, they contained 27636 lines of source code. It was clear that different phases had to be established: concentrating into smaller objectives would make it easier to organize and evaluate the progress of the project.

But something had to be done first. The code provided by ETSI had to work, that is, compiled, linked and executed in a PC. As it has been commented before, it would serve as a reference to verify the implementation in ADI: coding a speech sample with the "C vocoder" would give us, at any point of the process, the "officially" expected result to be compared with the one obtained with our "ADI vocoder" at the same point.

Being standard ANSI-C, it shouldn't be difficult to make the code work in our Visual C++ environment. And it wasn't. With a small change in the file "typedefs.h", to specify the platform we were working with, we got executable files, both for the encoder and the decoder. We tried then to encode and decode a real speech sample.

The file “spch_dos.inp” is provided by ETSI together with the source code. It had to be passed through the encoder, with the file “allmodes.txt” specifying the AMR mode for each frame (in this case it switches between all possible modes) and the coded file should be identical to the “spch_dos.cod”, also provided. We performed the test and the result was the expected. Then we tried to pass the just encoded sequence to the decoder and compared it with the “spch_dos.out” file provided. Again, they were identical.

We also wanted to perform this first test to get an idea (to some extent) of the degree of optimization it was going to be necessary in the project. The time the encoder needed to process the speech frames was an important indicator (the decoder would be much faster). With the Pentium III processor we were using, and the standard voice test sample “spch_dos.inp”, it took a little more than 10 sec. to complete the encoding of the 8.5 sec. sample (425 speech frames, at 20 msec. per frame). The PC processor couldn't make it in real time. It confirmed the demanding algorithm we were dealing with, and also confirmed that an optimization process was going to be needed when we implemented it into the ADSP-2181 [9].

With all the previous considerations, we could make a decision about the methodology we were going to follow.

4.1.2 Procedure

- To divide the large implementation into smaller, more affordable, modules, we decided to follow the logical flow of the functions of the program. Beginning with the first function the speech sequence would “encounter” when entering the vocoder, that is, the “Main” function of the encoder (“Speech_Encode_Frame”), we would follow the same path as the speech frames, implementing the functions in the same order we found them. That way, any moment we could test the work done, from the beginning to the last point implemented. Of course, a first division was clear between the encoder and the decoder, but then we had to take our way down into lower levels. Table 4.1 shows the code hierarchy (for clarity, only the encoder is shown, with functions up to level 4, basic routines avoided).

Following the table, the order would be to begin with “Speech_Encode_Frame”, then to “Pre_Process”, “cod_amr”, “Copy”... and so on.

Speech_Encode_Frame	Pre_Process		
	cod_amr	Copy	
		Vad1	filter_bank
			vad_decision
		Tx_dtx_handler	
		Lpc	Autocorr
			Lag_window
			Levinson
		Lsp	Az_lsp
			Q_plsf_5
			Int_lpc_1and3_2
			Int_lpc_1and3
			Q_plsf_3
			Int_lpc_1to3_2
			Int_lpc_1to3
			Copy
		dtx_buffer	Copy
			Log2
		dtx_enc	Lsp_lsf
			Reorder_lsf
			Lsf_lsp
		Set_zero	
		lsp_reset	Copy
			Q_plsf_reset
		Cl_ltp_reset	Pitch_fr_reset
		check_lsp	
		pre_big	Weight_Ai
			Residu
			Syn_filt
		ol_ltp	Pitch_ol
			Pitch_ol_wgh
		vad_pitch_detection	
		subframePreProc	Weight_Ai
			Syn_filt
			Residu
			Copy
		cl_ltp	Pitch_fr
			Pred_lt_3or6
			Convolve
			G_pitch
			check_gp_clipping
			q_gain_pitch
		Cbsearch	
		GainQuant	
		update_gp_clipping	Copy
		subframePostProc	Syn_filt
		Pred_lt_3or6	
		Convolve	
	Prm2bits	Int2bin	

Table 4.1: Code hierarchy (partial)

- When a function called a subroutine, even if the subroutine didn't exist yet, the parameters where placed in the proper registers or memory positions (where eventually the subroutine would look for them). The CALL instruction itself was written, but left disabled in the code if the subroutine didn't exist.

- The implementation of each function was made literally in a first approach, that is, totally equivalent to the C counterpart instruction by instruction. Then, it was tested (as we'll see in the next point). When the test was successful, an optimization was performed, rearranging the instructions into multifunction instructions and using other techniques that have been seen in chapter 3 and will be shown in more detail in chapter 4.2. Finally, a new test for the optimized version and the function could be considered done.

- The testing process depended on the function itself. For basic functions, using the VisualDSP debugging tools was enough. Setting a breakpoint at the beginning of the function and then stepping over every instruction, manually defining the data in the registers and memory, allowed to verify the correct behavior of the function.

  When more complex functions had to be tested, or critical points of the vocoder process were reached, the whole test sample “spch_dos.inp” was used. Using the original C code and the Visual Studio debugging tools, the test sample was passed through the vocoder, and sequences were taken at the beginning and ending of the function we wanted to check. If repeating the same process, but with our own vocoder, produced the same sequences, then we considered the function tested. It's important to remember at this point that only a test sequence was used, a standard voice sample. Working with that sample didn't guarantee that the whole set of test vectors, provided by ETSI to test a bit-exact implementation [21], would pass. Furthermore, provided that the test vectors are prepared to test non-standard situations, like overflow errors, that doesn't normally appear during a voice conversation, it was very likely that some of them would fail through our functions at this point. We counted on it, but using the whole set of test vectors with individual functions would have made the process much slower. We should deal with them later. But assuring a correct implementation with normal speech samples made us confident that we would get, if not bit-exact, at least a working vocoder.

- Regarding the optimization: without a reference, it was difficult to know the degree of optimization required. In other words, there are always ways to save one processor cycle, but they have a cost to pay, be it in hours spent by the developer, be it in processor's memory used (more on this in chapter 4.2). As we had a clear objective to accomplish, the system functioning in real-time, some calculations were made to get an idea of the real meaning of that objective [8].

  First of all, from the specifications we know that the ADSP-2181 processor operates with a 25 ns instruction cycle time, for all the instructions (even multioperation instructions). We will try to convert that value to a more often used parameter when referring a processor's performance: MIPS. The number of MIPS a processor can provide is the number of Millions of Instructions Per Second. In our case, that's equivalent to the millions of cycles per second, so:

$$(25\,ns/cycle)^{-1}=(25\times10^{-9}\,s/cycle)^{-1}=4\times10^{7}\,cycle/s$$

$$MIPS=\frac{(4\times10^{7}\,cycle/s)}{10^{6}}=40$$

  So 40 MIPS is the maximum power we can get from the processor, but how many of those MIPS are we using into a specific function? We usually work with complete frames or subframes into a function but, to answer the question, it would be easier if we could work with individual samples (meaning by *sample* here each one of the 16-bit words that form the frame; each frame containing 160 samples). The reason is that, in the end, the algorithm operates sample by sample, and the instructions that a function uses for each sample are easier to count.

To work with samples, the only information we need is that they are taken at a frequency of 8 KHz. In one second, we have to process 8000 samples. In that same second, one million cycles dedicated will count as 1 MIP, so:

$$\frac{(10^6\, cycle / s)}{(8000\, sample / s)} = 125\, cycle / sample$$

We could then implement and optimize our code taking into account that every 125 instructions dedicated to one sample would cost 1 MIP of our processor. That knowledge was useful to detect, simply looking at the code, the functions we had to put a special interest in. And, although we are not going to give any more details on how this affected to specific functions, we'll come back to the MIPS issue soon, when we analyze the final results of the project.

4.2 Cases of special interest

It would make little sense describing function by function all the implementation process. It is not possible due to space limitation, nor is it the purpose of this report. As an extract of the whole process, a complete example is shown in Appendix C. We have chosen one function, the Levinson-Durbin algorithm, that is very representative of the complete system. The ANSI-C code is first shown, and then our implementation of the same code into ADI, the assembling language of the DSP. Of course, all the functions of the AMR vocoder are available, in case someone was interested. We would be willing to provide them upon previous request.

It is very interesting, though, to comment on some particular cases that we encountered during the development of the project and that we found had special relevance in the implementation. It's our selection of cases of interest. It can be useful to understand the typical problems that were found and the solutions we applied.

4.2.1 Sum of products

Perhaps the single most common operation in DSP algorithms is the sum of products. It consists in fetching two operands (such as a coefficient and a sample point) and then multiplying the operands and summing the result with previous products.

As we have seen, the ADSP-2181 can execute both data fetches and the multiplication/ accumulation in a single-cycle. Besides, it doesn't introduce extra cycles managing loops, so an operation of this type can execute with sustained single-cycle throughput. Typically, it can be written with just two program lines:

```
                DO sample_loop UNTIL CE;
sample_loop:            MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M5);
```

4.2.2 Common mistake: testing a memory value

If we wanted to check if the value of *sample_var* is greater than zero (GT) to perform some action, the next code would be INCORRECT:

```
                AR = DM(sample_var);
                IF GT JUMP ...
```

The reason is that a data movement doesn't alter the status flags, only operations do. So the CORRECT version of the code would be:

```
                AR = DM(sample_var);
                AR = PASS AR;
                IF GT JUMP ...
```

The operation *PASS* doesn't do anything, but activates the status flags.

4.2.3 Testing 32-bit values

To produce the correct flags when testing a 32-bit value, for example stored in the registers MR1 (higher 16 bits) and MR0 (lower 16 bits), we begin testing the higher bits. If they are different from zero, they provide all the information we need. In case they are equal to zero (EQ) the absolute value of the lower part (because the lower 16 bits don't store sign information) is tested.

```
                AR = PASS MR1;
                IF EQ AR = ABS MR0;
```

4.2.4 Saving one cycle per iteration in long loops

There's a way to save a cycle in a loop by performing the last iteration outside the loop. For example, imagine we have to sum the components of some vector, *sample_vector*, of N components:

```
                I0 = ^sample_vector;
                CNTR = N;
                M1 = 1;
                DO loop UNTIL CE;
                        AY0 = DM(I0,M1);
loop:                   AR = AR + AY0;
```

The loop uses two cycles, one to fetch the value and the other to add the result, so we have used 2N processor's cycles (plus 3 extra initialization cycles outside the loop). Let's take a look at the next variation:

```
                I0 = ^sample_vector;
                AY0 = DM(I0,M1);
                CNTR = N - 1;
                M1 = 1;
                DO loop UNTIL CE;
```

```
loop:                AR = AR + AY0,AY0 = DM(I0,M1);

                /* now the extra iteration, outside the loop */
                AR = AR + AY0;
```

By taking the first memory fetch and the last addition out of the loop, we can use inside a multi-operation that adds the value and fetches the next value in the same cycle. The total processor's cycles used has been reduced to N (instead of 2N) and we have only increased the extra cycles to 5 (instead of 3).

4.2.5 Common mistake: initializing variables in subroutines

If some variable needs to be initialized at the beginning of a subroutine, for example the variable *lsp_flag = 0,* it is not enough to use the following ADI code:

```
                .VAR/DM lsp_flag;
                .INIT lsp_flag: 0;.
```

The reason is that we wanted the variable to be re-initialized every time the subroutine was called, while in ADI it is initialized only once, at the beginning of the program, keeping its value between calls. The solution would be to include the necessary instructions at the beginning of the subroutine to assign the value to the variable every time is called.

4.2.6 Analog Devices bug! Use of constants in ALU operations

Even if the use of any constant value is allowed as a valid parameter for an ALU instruction, as can be checked in Appendix B, we discovered that it produced a wrong compiled code when the constant was equal to zero. Very oddly, instead of zero, the current value of register *AY0* was used.

It would be long to explain why such a case could happen, but it did, and the error was very difficult to detect. Analog Devices were informed of it, agreed it was an error of their compiler and thanked us for the information.

4.2.7 Reutilization of memory for different buffers

Several functions of the vocoder used huge temporal buffers. Implementing them as local to the specific function would have supposed a great amount of memory used, but there was a solution to optimize that. Instead of using local buffers, some global buffers where declared. Then they could be used in different functions to store temporal buffers.

To make it easier the legibility of the code, local names can be used. Let's assume that we have a global buffer declared as *temp_dm*. A function that wanted to use it to store two local buffers, *filter_param* and *speech*, could define them easily this way:

```
#define  filter_param        temp_dm
#define  speech              temp_dm + 40
```

And then use them as if they were local:

```
I0 = ^speech;

AR = DM(I0,M1);
...
```

But some precautions must be taken: using it only locally (can't store anything useful if the execution exits the function, either because it ends or because it calls a subroutine) and not exceeding their maximum length.

4.2.8 Multiplication of fractional 1.15 numbers

When we have fractional values of 16 bits in format 1.15 (they have an absolute value less than 1), some steps must be performed when multiplying them.

If we multiply two 1.15 numbers, a 2.30 number results in the MAC unit of the processor. The value of the highest bit is irrelevant (as the result of the multiplication doesn't exceed 1 in absolute value, the two bits of the integer part are either “00” or “11”), so we can left-shift the result and we will get a 1.31 value. Then we simply take the 16 most significant bits and we will have a 1.15 number again.

4.3 Project results

Applying the method described in chapter 4.1, using some of the techniques described in chapter 4.2, and dedicating tons of hours, one day came when the AMR vocoder had been totally implemented into the ADSP-2181.

As a systematic testing method had been applied, we didn't get any surprise and it was rather easy to check that the encoder and the decoder worked with the file “spch_dos.inp” provided by ETSI. The coded file was identical to the “spch_dos.cod” provided, and the decoded file was identical to “spch_dos.out”.

Another very important test had to be done. It had been periodically done throughout the process, with good results, but the results that counted were the last ones. We are referring to the MIPS performance of the vocoder, and if it would be able to work in real-time. Specifically, and provided that we are treating independently encoding and decoding, we were interested in the MIPS consumption of the encoder, that was way more complex (and slow) than the decoder. Using the profiling options in the VisualDSP simulator, the following results were obtained for each mode, shown in table 4.2. A standard voice sequence (vector “t08”) was used as an input, trying to get results as close to real use as possible.

MODE	*BITRATE (Kbps)*	*ENCODER MIPS*
MR475	4.75	23
MR515	5.15	18.7
MR59	5.90	23
MR67	6.70	29
MR74	7.40	26.5
MR795	7.95	28.1
MR102	10.2	27.2
MR122	12.2	28.7

Table 4.2: MIPS consumption for each mode

The first thing to note, of course, is that the MIPS used are less than the 40 MIPS the ADSP-2181 is capable of. We have even obtained an important margin, that will be useful to include into the ADSP, together with the AMR encoder, some possibly necessary functions (analog to digital speech conversion, for example). The system could work in real-time. And the same applies to the decoder by a greater margin. As a side note, unfortunately, we wouldn't be able to build a whole dual-channel system, with the encoder, the decoder and the rest of the external modules working in one ADSP-2181. But that was not part of the objectives of the project.

Interestingly enough, the MIPS consumption is not proportional (or at least related in some way) to the bitrate used. The reason is that all the input samples have to be processed in any of the modes, no matter the output frame length. Nevertheless, internally each mode acts as a different vocoder, with its specific functions, and they are more or less complex independently one from each other.

Trying to find the most cost-expensive function, different profiling tests were made, evaluating the MIPS of each of the main functions. Results are presented in table 4.3. The numbers were obtained with the same test sequence “t08” and mode MR122.

FUNCTION	*MIPS*	*DESCRIPTION*
Pre_Process	0.42	Preprocessing of input speech
Lpc	0.84	LP coefficients calculation
Lsp	5.25	Conversion from LP coefficients to LSPs
ol_ltp	3.4	Compute the open loop pitch lag
subframePreProc	2.3	Subframe preprocessing
cl_ltp	2.6	Close-loop fractional pitch search
Cbsearch	11.8	Innovative codebook search
GainQuant	0.2	Quantization of gains
subframePostProc	0.6	Subframe postprocessing

Table 4.3: MIPS consumption of the main functions

The most processor-consuming function, by a difference, is “Cbsearch”. The codebook search, as we studied in chapter 2, is the key technique in the AMR vocoder. It provides the most important reduction in the length of the encoded bits, reducing the transmission of the speech frame to the index and gain of the codevector into the codebook. But, as we have just seen, this great reduction comes with a great computational cost.

So the vocoder seemed to work, but we wanted to “hear it”. Using the ADSP-2181 EZ-LAB evaluation board, a microphone, a speaker, and some audio functions (not made by the author), including analogue to digital (13-bit uniform PCM) and digital to analogue conversion, we were able to encode our own voice and then recover it through the decoder, using different modes. The audio quality was quite good, obviously better with the 12.2 Kbit/s mode than with the 4.75 Kbit/s mode (the two extreme modes), but acceptable in any case. It was one of the most satisfying moments of the project.

But the job wasn't finished yet. Remember the bit-exact implementation and the test vectors? As expected, some of them didn't pass the test. Exactly 176 out of 360 test vectors failed, most of them corresponding to the encoder. It's a very significant information about the ETSI standard of quality: an apparently working vocoder couldn't pass almost half of their tests [21].

The process of solving these particular cases wasn't as hard as it may seem, though. Using the debugging tools, at the same time with the C code and the ADI code, it was easy to identify the exact point where they diverged and correct the mistake. Most of the times it was due to an overflow situation: an addition of very high (or low) values through many samples produced a saturation of the corresponding value that was treated differently by the ADSP accumulator than by the ETSI implementation. Once detected and corrected, the amount of test vectors failing was greatly reduced (which is expected, because the same vectors are repeated for every mode).

It has to be noticed that a lot of time was spent trying to find why some of the tests failed with no logical explanation, until we found out that the C code also failed with them! The reason was that the C code we were working with for the last months (v7.3.0) was not the last available. Apparently, ETSI had also detected the error and had updated the code. Once the latest version (v7.4.0) was downloaded, the changes it included were implemented into our ADI code and the problem was solved.

It took long, but finally, with only about 10 or 12 small corrections, all the test vectors worked, and our vocoder could be considered a bit-exact implementation of the ETSI standard.

Chapter 5

Conclusions

5.1 Project objectives accomplished

We begun chapter 1 enumerating the objectives we would try to accomplish in the project. It is now time to review them and see if we have succeeded.

The first objective was to study the Adaptive Multi-Rate vocoder. Prior to the implementation, all the theory about it was studied (it was presented in chapter 2, and is complemented in Appendix A), and during the project we have been able to identify the parts of the vocoder in their "physical" translation into programming code. Of course, the magnitude of the project didn't allow to study every specific parameter of every filter, but it did allowed a general understanding of the system.

The second objective was to study digital signal processors, specially the ADSP-2181, and how they are specialized in dealing with the computing-expensive algorithms that signal processing requires. In this case, the theory was also studied, but the importance of the experience obtained was even higher. We had to deal with the optimization problem and took profit of every dedicated instruction of the processor, as shown in chapter 4.

Maybe, the main objective was the implementation itself: we had to get the vocoder working and we wanted that it passed the strong restrictions of the ETSI standard for a "bit-exact" implementation. It took very long, as we have seen, but the vocoder finally passed all the test vectors. This guarantees that it meets all the official requirements of the mobile industry and that it is ready to be used in any mobile device.

The last objective could be considered an “extra”. Real-time implementation was desirable, but the processor chosen was not specially fast. If the program hadn't run fast enough, it could have been solved choosing a more powerful processor. Anyway, we obtained a rather optimized implementation: it could run in real-time, considering the two parts, encoder and decoder, as different programs (as they are presented by ETSI). If a commercial implementation required the whole system, plus some external functions, running in real-time in the ADSP-2181, a different solution would be necessary. But we'll discuss about that in “Future perspectives”.

As a conclusion, having reviewed the results of the project, we think the main objectives were accomplished successfully.

5.2 Future perspectives

The main objectives of the project, as we have just seen, were accomplished. But that doesn't mean that no further improvement could be done. For example, special dedication has been put into optimizing the code, but the optimization was aimed at reducing the MIPS of the program, making it faster. Nevertheless, there's another important resource that could be optimized: memory used.

Although memory is becoming cheaper and cheaper and it seems as if, in general, not much effort is put by software companies into controlling the amount of memory their programs use, in the end it's also a limited resource and should be optimized. A few ideas to reduce memory are now going to be presented, that could be applied to further optimize the assembling code of this project:

- PM (Program Memory) of the ADSP-2181 is used mainly to store the program instructions, but it has often been used to store data, sometimes big tables that consumed a lot of memory. That was done to take profit of the architecture of the DSP, that allows reading data from Program Memory and Data Memory in one single cycle. When data from two vectors are used in a computation, it is more efficient to store one in PM and the other in DM and access them at the same time. But Program Memory is 24-bit wide, while our data was always 16-bit wide, so 8 bits were wasted with every entry. Some method to take profit of those 8 extra bit could be thought, like storing two tables in the same portion of memory: one of them using the normal 16 bits and the other dividing every 16-bit word into two 8-bit words that could use the free space. Of course, the access to them would be computationally more complicated.

- Portions of code that are identical could be written only once and used as small functions. For example, common operations used in the algorithm include shifting with saturation or 32-by-16-bit multiplications. They are implemented into several lines of code, so writing them as small functions would save some memory.

- "Pre_proc.dsp" and "Post_proc.dsp", while not identical, are very similar. Again, their common parts could be written only once, so the use of memory wasn't duplicated.

- The mechanism of saturation was implemented in almost every operation performed, but maybe it's not always necessary. That is, the test vectors that guarantee for a bit-exact implementation maybe never produce saturation in some of those operations (because mathematically it's impossible, for example). So the extra instructions written to take care of the saturation are never used and could be removed with no effect in the results. In this case, though, the time employed in finding them could be long, and maybe not worth it.

Apart from the improvement in memory used, another future perspective could be the implementation of the dual-channel [13]. As we have repeated a couple of times before, the ETSI standard implements the encoder and the decoder independently. In real use, though, they should normally work together, with extra modules like analog to/from digital speech conversion, echo cancellation or many more. It would be an interesting future work to join all those programs into one and implement it in the ADSP. The problem could be, in that case, that the ADSP-2181 wasn't powerful enough to support all those MIPS. An upgrade to a newest processor, like Analog Devices' ADSP-2184, very similar to 2181 but capable of 80 MIPS, could probably be the solution.

Appendix A

Speech production and coding

A.1 Speech production

Speech is used to communicate information from a speaker to a listener. The speaker must produce a speech signal in the form of a sound pressure wave that travels from the speaker's mouth to a listener's ears. Not only the speech production, but also the hearing process is an integral part of the so-called speech chain. Actually, the human auditory system also plays an important role in the production of the speech as a feedback.

The speech waveform (see Figure 3.1) is an acoustic sound pressure wave that originates from voluntary movements of anatomical structures which make up the human speech production system.

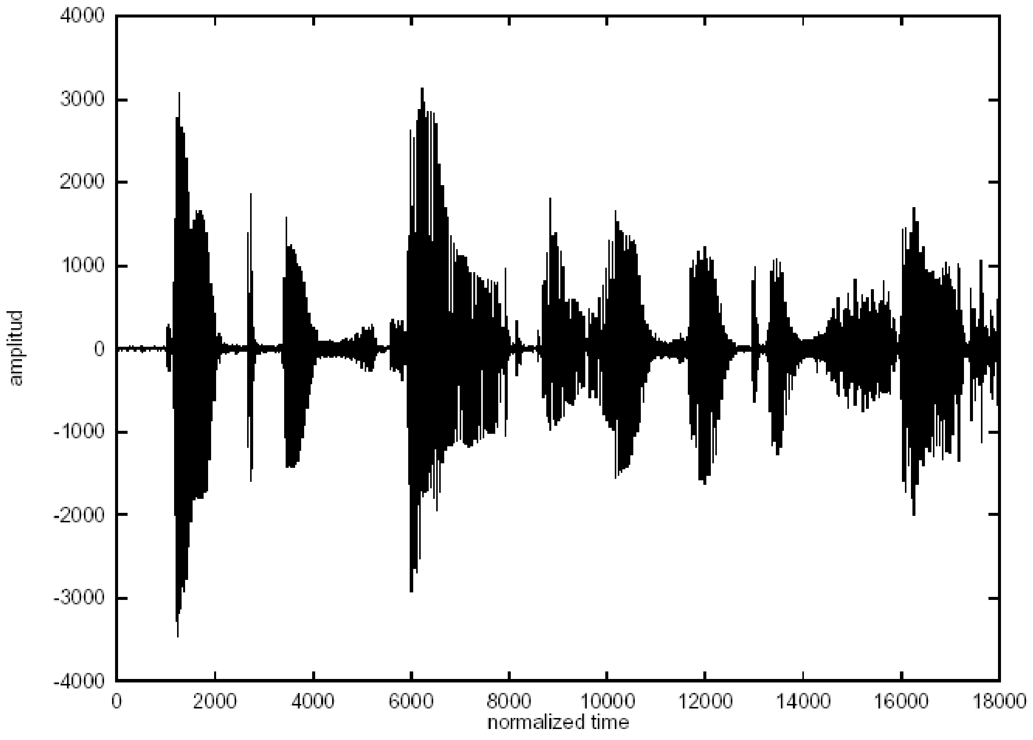


Figure 3.1: example of a speech signal

The gross components of the system are the lungs, trachea (windpipe), larynx (organ of voice production, where the vocal cords or folds are), pharyngeal cavity (throat), oral cavity (mouth) and nasal cavity (nose) [3]. In technical discussions, the pharyngeal and oral cavities are usually grouped into one unit referred to as the vocal tract and the nasal cavity is often called the nasal tract. These three main cavities of the speech production system comprise the main acoustic filter. The filter is excited by the organs below it and is loaded at its main output by a radiation impedance due to the lips. The function of the larynx is to provide a periodic excitation to the system for speech sounds that are known as voiced sounds.

Roughly speaking, the periodic vibration of the vocal folds is responsible for this voicing. The articulators are used to change the properties of the system, its form of excitation and its output loading over time. These articulators are human tissues and/or muscles which are moved from one position to another to produce the desired speech sounds.

One of the principal features of any speech sound is the manner of excitation. There are two elemental excitation types: voiced and unvoiced (there are other types of excitation, which are really just combinations of voiced, unvoiced and silence).

Voiced sounds are produced by forcing air through the glottis or an opening between the vocal folds. The tension of the vocal cords is adjusted so that they vibrate in oscillatory fashion at the pitch frequency, referred to as F0 (see Figure 3.2 (a)). Unvoiced sounds are generated by forming a constriction at some point along the vocal tract and forcing air through the constriction to produce turbulence (see Figure 3.2 (b)). As it was pointed out before, a sound may be simultaneously voiced and unvoiced (e.g. the sound corresponding to the letter z).

The spectral characteristics of the speech wave are time-varying (or nonstationary), since the physical system changes rapidly over time (but not instantaneously due to the requirement of finite movement of the articulators to produce each sound). As a result, speech can be divided into sound segments that possess similar acoustic properties over short periods of time. In Figure 3.3 (a), a voiced sound, there is evidence of periodic excitation, while in Figure 3.3 (b), an unvoiced sound, there is no such harmonic structure present.

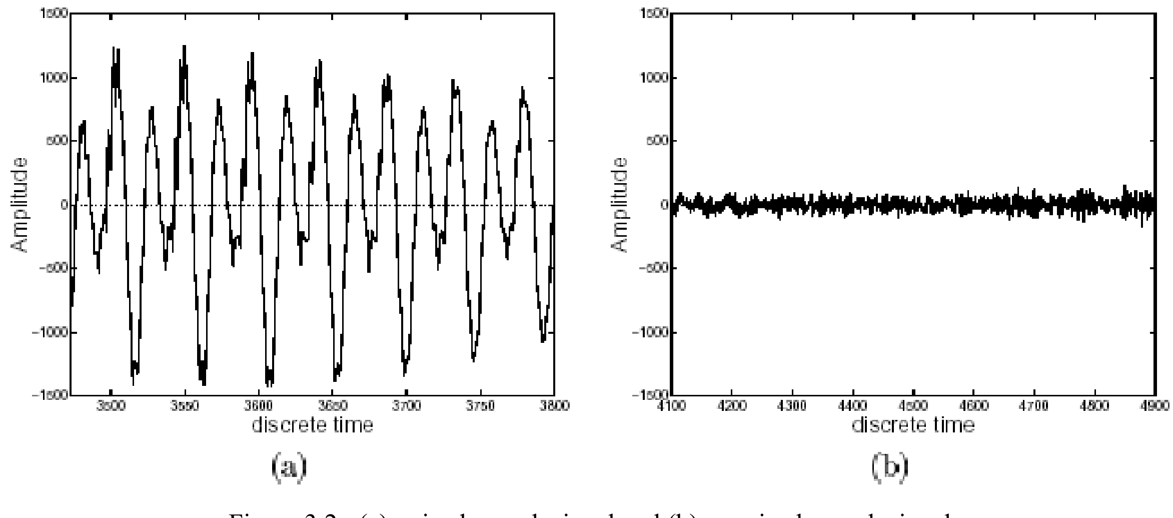


Figure 3.2: (a) voiced speech signal and (b) unvoiced speech signal

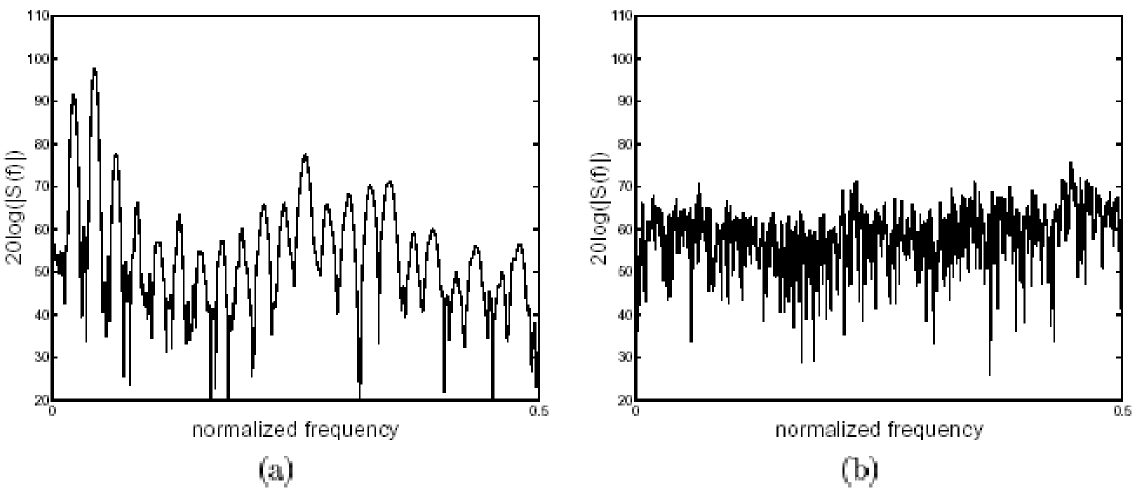


Figure 3.3: spectrum of (a) a voiced speech signal and (b) an unvoiced speech signal in frequency domain

It can also be noted in each case (and especially for the voiced sound) that there are well-defined regions of emphasis (resonances) and deemphasis (antiresonances) in the spectrum. These resonances are a consequence of the articulators having formed various acoustical cavities and subcavities out of the vocal tract cavities. So the locations of these resonances in the frequency domain depend upon the shape and physical dimensions of the vocal tract. Conversely, each vocal tract shape is characterized by a set of resonant frequencies.

From a system modeling point of view, the articulators determine the properties of the speech system filter. Since these resonances tend to shape the overall spectrum, speech scientists refer to them as formants and they are referred to as: F1, F2, F3, ...

A basic model for the speech production can be seen in Figure 3.4, composed by the two main different types of excitation (voiced, as a periodic signal, and unvoiced, as noise) and the vocal tract.

Nevertheless, the fact of modeling the source as a switch to voiced or unvoiced speech production, is a serious limitation for several reasons. First, human speech production does not require voicing to turn off immediately prior to an unvoiced phoneme. In addition, several phonemes, such as voiced fricatives (/z/, /v/), possess two sources of excitation, vocal fold movement and a major constriction resulting in both voiced and unvoiced forms of excitation. Moreover, since the model is based on acoustic tube theory assuming short-term stationarity, it lacks the ability to characterize rapidly changing excitation properties such as those found in plosive sounds like /t/ and /b/.

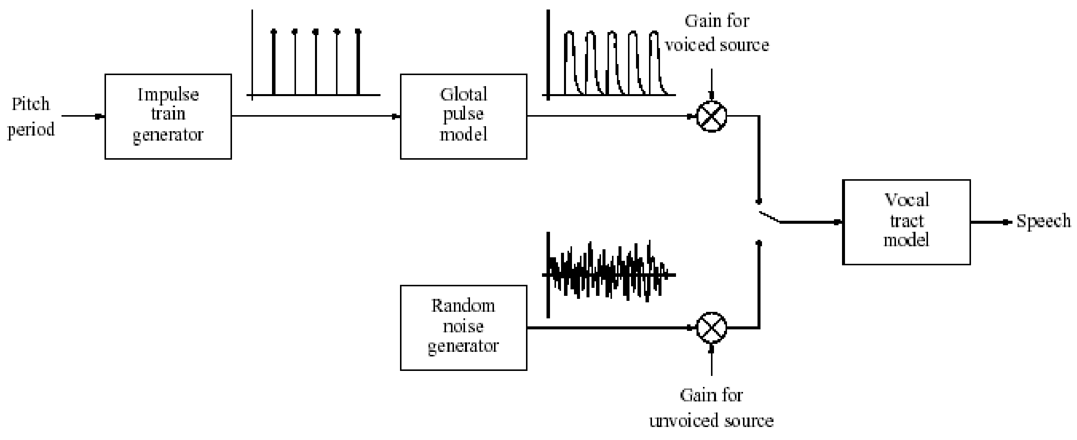


Figure 3.4: Basic model representing speech production

A.2 Speech coding

Coding algorithms seek to minimize the bit rate in the digital representation of a signal without an objectionable loss of signal quality in the process. High quality is attained at low bit rates by exploiting signal redundancy as well as the knowledge that certain types of coding distortion are imperceptible because they are masked by the signal. New models based on signal redundancy and distortion masking are becoming increasingly more sophisticated, leading to continuing improvements in the quality of low bit rate signals.

Speech coding techniques can be broadly divided into two classes: waveform coders, that aim at reproducing the speech waveform as faithfully as possible, and voice coders (vocoders), that use some parameters determined by a speech analysis of the original signal to represent it. The waveform coders are able to produce high-quality speech at high enough bit rates; vocoders produce intelligible speech at much lower bit rates, but the level of speech quality is also lower.

The general model for vocoders follows the speech production system studied in the previous section. The vocal tract is modeled as an all-pole filter. For voiced speech, the excitation is a periodic impulse train with period equal to the pitch period of the speech. For unvoiced speech, the excitation is a white noise sequence. In addition, there can be an estimated gain parameter included in the model. Basically, the different vocoders estimate the model parameters from frames of speech (speech analysis), encode and transmit the parameters to the receiver on a frame-by-frame basis, and reconstruct the speech signal from the model (speech synthesis) at the receiver. Different types of vocoders include: channel vocoders, cepstral vocoders, phase vocoders, formant vocoders and linear predictive coders. The last of these are the most widely used in practice today, and the model studied in this project belongs to this class.

The objective of the linear prediction analysis (LP analysis) in speech processing is to estimate the parameters of the all-pole model of the vocal tract for a given signal. If these parameters have been determined for a given frame of speech to be transmitted or stored, this is called linear predictive coding.

Several methods have been devised for generating the excitation sequence for speech synthesis. Next we describe several LPC-type speech analysis and synthesis schemes that differ primarily in the type of excitation signal that is generated for speech synthesis [1].

A.2.1 LPC-10 algorithm

The earliest LPC-based vocoders followed exactly the voice representation model studied in section 3.1. The all-pole filter directly used a pitch-pulse excitation for synthesis of voiced speech, and a noise source excitation for unvoiced. Such a vocoder first caught the public's attention in the 1970's. The algorithm is usually called LPC-10 in reference to the fact that 10 coefficients are typically employed.

A.2.2 Residual excited linear prediction vocoder

Speech quality in LPC can be improved at the expense of a higher bit rate by computing and transmitting a residual error.

Once the LPC model and excitation parameters are estimated from a frame of speech, the speech is synthesized at the transmitter and subtracted from the original speech signal to form a residual error. The residual error is quantized, coded, and transmitted to the receiver along with the model parameters. At the receiver the signal is synthesized by adding the residual error signal to the signal generated from the model. Thus the addition of the residual error improves the quality of the synthesized speech.

Another approach that produces a residual error is shown in figure 3.5. In this case, the original speech signal is passed through the inverse (all-zero) filter to generate the prediction residual signal. Then this residual will be transmitted, after being properly processed and encoded so as to reduce the bit rate. Once in the receiver, the residual signal will be passed through the all-pole filter to recover the original speech. We note that this method does not require pitch information and voicing information. The residual error signal provides the excitation to the all-pole LPC model. This LPC vocoder is called a residual excited linear prediction (RELP) vocoder.

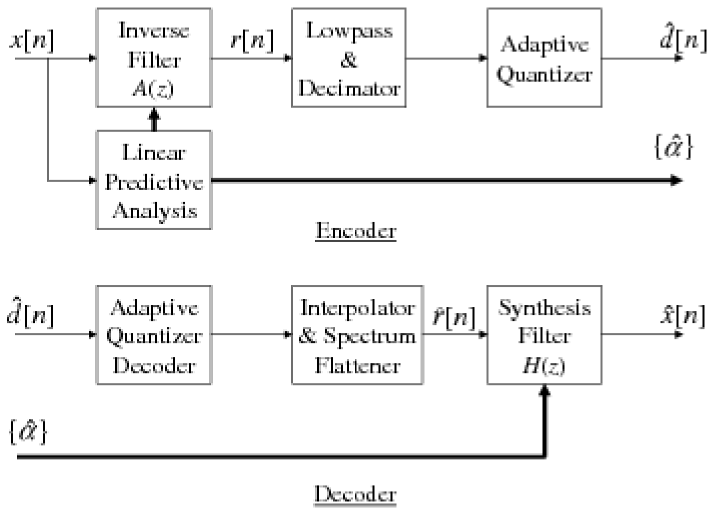


Figure 3.5: RELP encoder and decoder

A.2.3 Multipulse LPC vocoder

One of the shortcomings of RELP is that the regeneration scheme results in a crude approximation of the high frequencies. Multipulse LPC is an analysis-by-synthesis method that results in a better excitation signal for the LPC vocal system filter.

The LPC filter coefficients are determined from the speech signal samples by the conventional methods. The output of the all-pole filter is the synthetic speech, which is subtracted from the original speech signal to form the residual error sequence. This error sequence is then passed through a perceptual error weighting filter that is used to control the noise spectrum weighting.

The multipulse excitation consists of a short sequence of pulses (discrete-time pulses) whose amplitudes and locations are chosen to minimize the energy of the weighted error signal. For simplicity, the amplitudes and locations of the impulses are obtained sequentially by minimizing the error energy for one pulse at a time.

A.2.4 Code-excited linear prediction vocoder

Code Excited Linear Prediction (CELP) is a speech coding algorithm originally proposed by M.R. Schroeder and B.S. Atal in 1985. At the time, it provided significantly better quality than existing low bit-rate algorithms, such as RELP and LPC vocoders.

Figure 3.6 describes a generic CELP decoder. The excitation is produced by summing the contributions from an adaptive (aka pitch) codebook and a fixed (aka innovation or stochastic) codebook. The fixed codebook is a vector quantization dictionary that is hard-coded into the codec. This codebook can be algebraic (ACELP) or be stored explicitly. The entries in the adaptive codebook consist of delayed versions of the excitation. This makes it possible to efficiently code periodic signals, such as voiced sounds.

The filter that shapes the excitation has an all-pole model of the form 1 / A(z), where A(z) is called the prediction filter and is obtained using linear prediction (Levinson-Durbin algorithm). An all-pole filter is used because it is a good representation of the human vocal tract and because it is easy to compute.

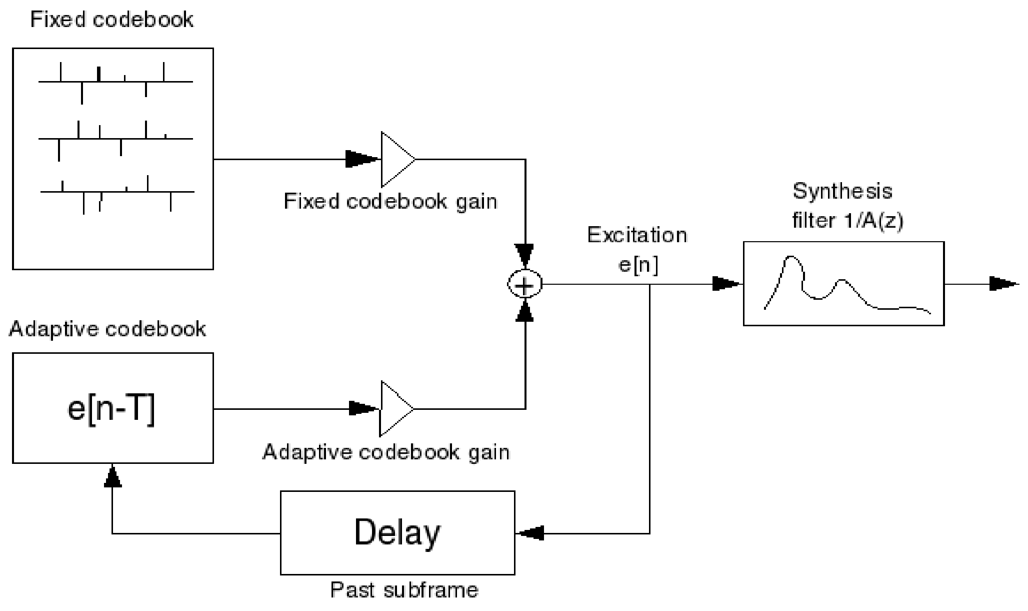


Figure 3.6: CELP decoder

The main principle behind CELP is called Analysis-by-Synthesis (AbS) and means that the encoding (analysis) is performed by perceptually optimizing the decoded (synthesis) signal in a closed loop. In theory, the best CELP stream would be produced by trying all possible bit combinations and selecting the one that produces the best-sounding decoded signal. This is obviously not possible in practice for two reasons: the required complexity is beyond any currently available hardware and the "best sounding" selection criterion implies a human listener.

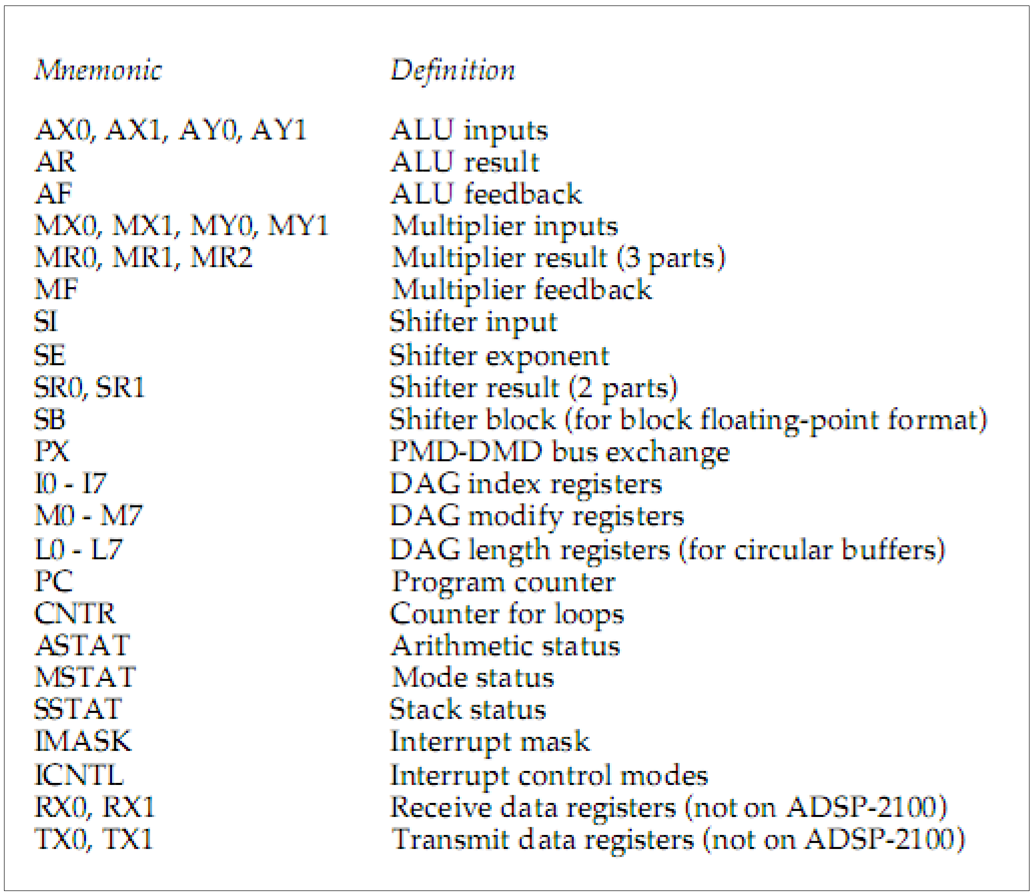
In order to achieve real-time encoding using limited computing resources, the CELP search is broken down into smaller, more manageable, sequential searches using a simple perceptual weighting function. Typically, the encoding is performed in the following order:

- LPC coefficients are computed and quantized, usually as LSPs
- The adaptive (pitch) codebook is searched and its contribution removed
- The fixed (innovation) codebook is searched

Appendix B

Simplified instruction set for the ADSP-2181

B.1 Assembly language overview



Mnemonic	*Definition*
AX0, AX1, AY0, AY1	ALU inputs
AR	ALU result
AF	ALU feedback
MX0, MX1, MY0, MY1	Multiplier inputs
MR0, MR1, MR2	Multiplier result (3 parts)
MF	Multiplier feedback
SI	Shifter input
SE	Shifter exponent
SR0, SR1	Shifter result (2 parts)
SB	Shifter block (for block floating-point format)
PX	PMD-DMD bus exchange
I0 - I7	DAG index registers
M0 - M7	DAG modify registers
L0 - L7	DAG length registers (for circular buffers)
PC	Program counter
CNTR	Counter for loops
ASTAT	Arithmetic status
MSTAT	Mode status
SSTAT	Stack status
IMASK	Interrupt mask
ICNTL	Interrupt control modes
RX0, RX1	Receive data registers (not on ADSP-2100)
TX0, TX1	Transmit data registers (not on ADSP-2100)

Table B1: Registers of the ADSP-2100 family

The ADSP-2100 family's assembly language uses an algebraic syntax for ease of coding and readability. The sources and destinations of computations and data movements are written explicitly in each assembly statement, eliminating cryptic assembler mnemonics. Each assembly statement, however, corresponds to a single 24-bit instruction, executable in one cycle. Register mnemonics, listed in table B1, are concise and easy to remember.

B.2 ALU, MAC & Shifter instructions

This group of instructions performs computations. All of these instructions can be executed conditionally except the ALU division instructions and the Shifter SHIFT IMMEDIATE instructions.

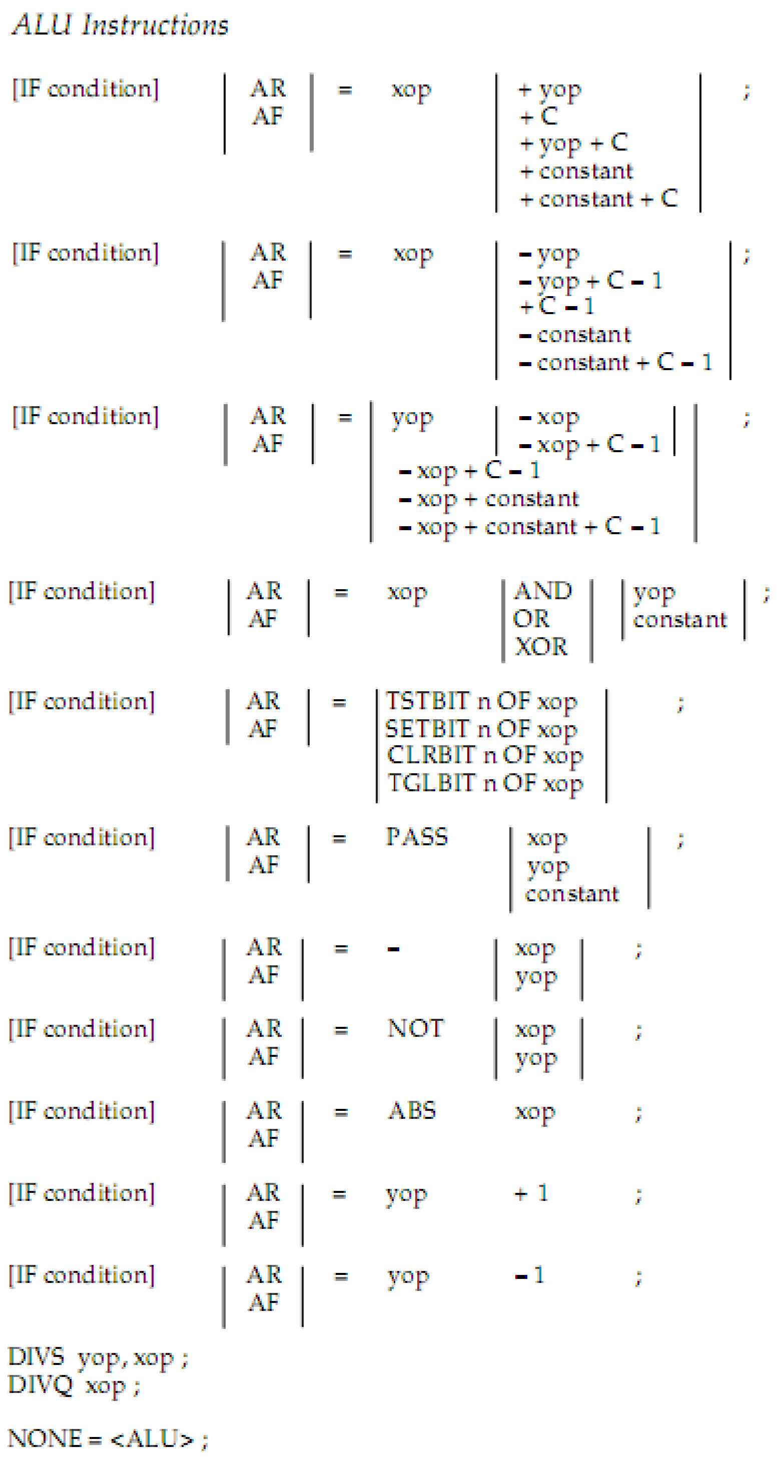
B.2.1 ALU Group

Here is an example of one ALU instruction, Add/Add with Carry:

IF AC AR=AX0+AY0+C;

The (optional) conditional expression, IF AC, tests the ALU Carry bit (AC); if there is a carry from the previous instruction, this instruction executes, otherwise a NOP occurs and execution continues with the next instruction. The algebraic expression AR=AX0+AY0+C means that the ALU result register (AR) gets the value of the ALU X input and Y input registers plus the value of the carry-in bit.

Table B2 gives a summary list of all ALU instructions. In this list, condition stands for all the possible conditions that can be tested and *xop* and *yop* stand for the registers that can be specified as input for the ALU. The conditional clause is optional and is enclosed in square brackets to show this. A complete list of conditions is given in Table B8.



ALU Instructions

[IF condition]	AR AF	=	xop	+ yop + C + yop + C + constant + constant + C		;
[IF condition]	AR AF	=	xop	− yop − yop + C − 1 + C − 1 − constant − constant + C − 1		;
[IF condition]	AR AF	=	yop − xop yop − xop + C − 1 − xop + C − 1 − xop + constant − xop + constant + C − 1			;
[IF condition]	AR AF	=	xop	AND OR XOR	yop constant	;
[IF condition]	AR AF	=	TSTBIT n OF xop SETBIT n OF xop CLRBIT n OF xop TGLBIT n OF xop			;
[IF condition]	AR AF	=	PASS	xop yop constant		;
[IF condition]	AR AF	=	−	xop yop		;
[IF condition]	AR AF	=	NOT	xop yop		;
[IF condition]	AR AF	=	ABS	xop		;
[IF condition]	AR AF	=	yop	+ 1		;
[IF condition]	AR AF	=	yop	− 1		;

DIVS yop, xop ;
DIVQ xop ;

NONE = <ALU> ;

Table B2: ALU instructions

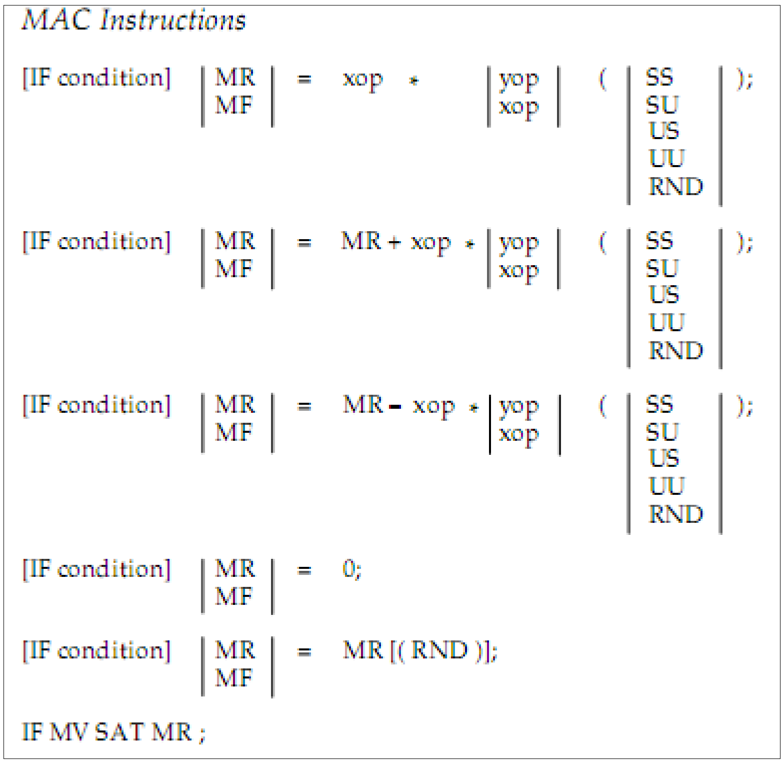
B.2.2 MAC Group

Here is an example of one of the MAC instructions, Multiply/Accumulate:

IF NOT MV MR=MR+MX0*MY0(UU);

The conditional expression, IF NOT MV, tests the MAC overflow bit. If the condition is not true, a NOP is executed. The expression MR=MR+MX0*MY0 is the multiply/accumulate operation: the multiplier result register (MR) gets the value of itself plus the product of the X and Y input registers selected. The modifier in parentheses (UU) treats the operands as unsigned. There can be only one such modifier selected from the available set. (SS) means both are signed, while (US) and (SU) mean that either the first or second operand is signed; (RND) means to round the (implicitly signed) result.

Table B3 gives a summary list of all MAC instructions. In this list, condition stands for all the possible conditions that can be tested and xop and yop stand for the registers that can be specified as input for the MAC.



MAC Instructions

[IF condition]	MR MF	=	xop *	yop xop	(	SS SU US UU RND	);
[IF condition]	MR MF	=	MR + xop *	yop xop	(	SS SU US UU RND	);
[IF condition]	MR MF	=	MR – xop *	yop xop	(	SS SU US UU RND	);
[IF condition]	MR MF	=	0;				
[IF condition]	MR MF	=	MR [(RND)];				
IF MV SAT MR ;							

Table B3: MAC instructions

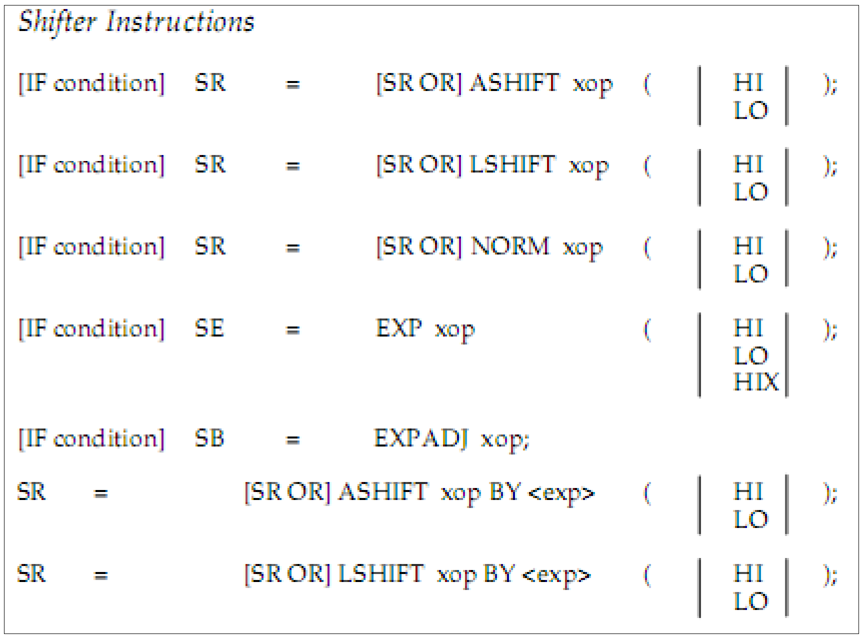
B.2.3 Shifter Group

Here is an example of one of the Shifter instructions, Normalize:

IF NOT CE SR= SR OR NORM SI (HI);

The conditional expression, IF NOT CE, tests the “not counter expired” condition. If the condition is false, a NOP is executed. The destination of all shifting operations is the Shifter Result register, SR. (The destination of exponent detection instructions is SE or SB, as shown below.) In this example, SI, the Shifter Input register, is the operand. The amount and direction of the shift is controlled by the signed value in the SE register in all shift operations except an immediate shift. Positive values cause left shifts; negative values cause right shifts.

The “SR OR” modifier (which is optional) logically ORs the result with the current contents of the SR register; this allows you to construct a 32-bit value in SR from two 16-bit pieces. “NORM” is the operator and “(HI)” is the modifier that determines whether the shift is relative to the HI or LO (16-bit) half of SR. If “SR OR” is omitted, the result is passed directly into SR. Table B4 gives a summary list of all Shifter instructions.



Shifter Instructions

[IF condition]	SR	=	[SR OR] ASHIFT xop	(	HI LO	);
[IF condition]	SR	=	[SR OR] LSHIFT xop	(	HI LO	);
[IF condition]	SR	=	[SR OR] NORM xop	(	HI LO	);
[IF condition]	SE	=	EXP xop	(	HI LO HIX	);
[IF condition]	SB	=	EXPADJ xop;			
SR	=		[SR OR] ASHIFT xop BY <exp>	(	HI LO	);
SR	=		[SR OR] LSHIFT xop BY <exp>	(	HI LO	);

Table B4: Shifter instructions

B.3 MOVE: read and write

MOVE instructions, shown in Table B5, move data to and from data registers and external memory. Registers are divided into two groups, referred to as reg which includes almost all registers and dreg, or data registers, which is a subset. Only the program counter (PC) and the ALU and MAC feedback registers (AF and MF) are not accessible. Table B6 shows which registers belong to these groups.

```
MOVE Instructions

reg     =     reg ;

reg     =     DM (<address>);

dreg    =     DM ( | I0 | , | M0 | );
                   | I1 | , | M1 |
                   | I2 | , | M2 |
                   | I3 | , | M3 |
                   ---------------
                   | I4 | , | M4 |
                   | I5 | , | M5 |
                   | I6 | , | M6 |
                   | I7 | , | M7 |

DM ( | I0 | , | M0 | )   =   | dreg   |   ;
     | I1 | , | M1 |         | <data> |
     | I2 | , | M2 |
     | I3 | , | M3 |
     ---------------
     | I4 | , | M4 |
     | I5 | , | M5 |
     | I6 | , | M6 |
     | I7 | , | M7 |

DM (<address>) =       reg;

reg     =     <data> ;

dreg    =     PM ( | I4 | , | M4 | );
                   | I5 | , | M5 |
                   | I6 | , | M6 |
                   | I7 | , | M7 |

PM ( | I4 | , | M4 | )   =   dreg;
     | I5 | , | M5 |
     | I6 | , | M6 |
     | I7 | , | M7 |
```

Table B5: MOVE instructions

	Data Registers: dreg
SB PX I0 – I7, M0 – M7, L0 – L7 CNTR ASTAT, MSTAT, SSTAT IMASK, ICNTL, IFC TX0, TX1, RX0, RX1	AX0, AX1, AY0, AY1, AR MX0, MX1, MY0, MY1, MR0, MR1, MR2 SI, SE, SR0, SR1

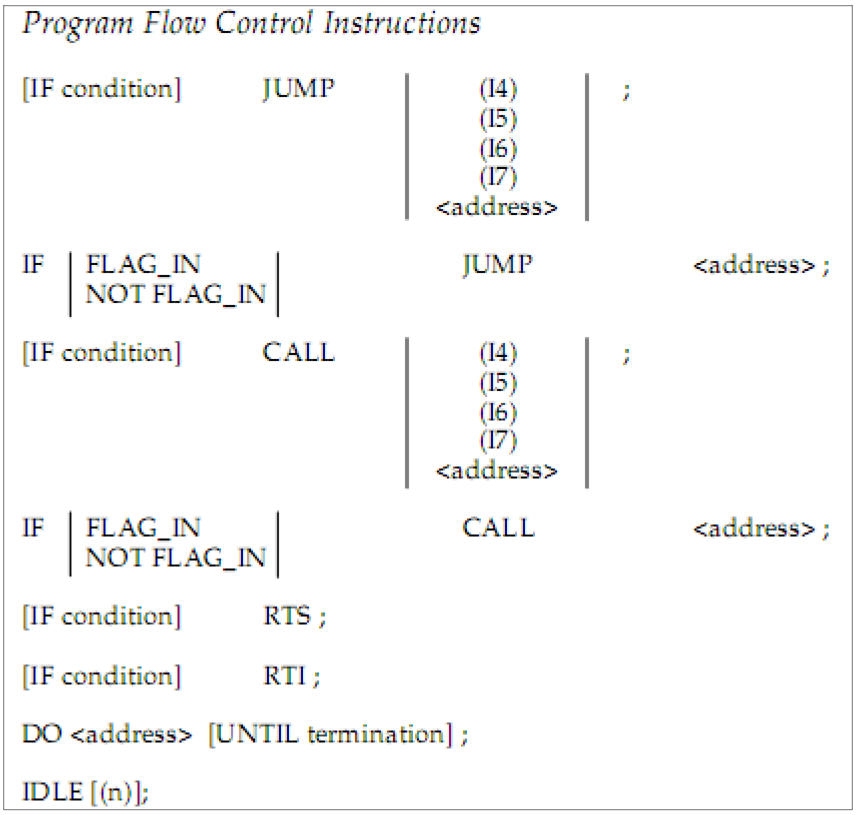
Table B6: Processor registers

B.4 Program flow control

Here is an example of one instruction:

IF EQ JUMP my_label;

JUMP is a familiar construct from many other languages. *My_label* is any identifier you wish to use as a label for the destination jumped to. Instead of the label, an index register in DAG2 may be explicitly used. The default scope for any label is the source code module in which it is declared. The assembler directive .ENTRY makes a label visible as an entry point for routines outside the module. Conversely, the .EXTERNAL directive makes it possible to use a label declared in another module.

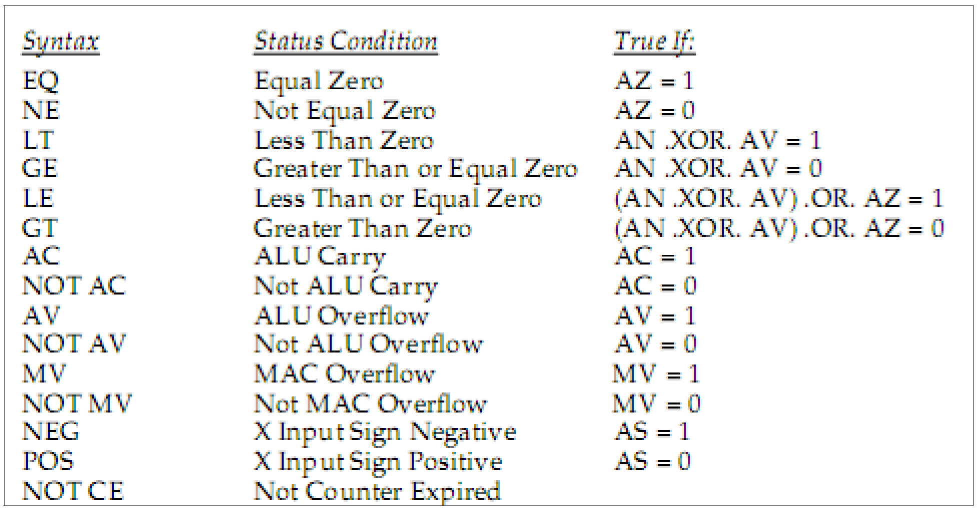


Program Flow Control Instructions

```
[IF condition]      JUMP      | (I4)      |  ;
                              | (I5)      |
                              | (I6)      |
                              | (I7)      |
                              | <address> |

IF | FLAG_IN     |            JUMP             <address> ;
   | NOT FLAG_IN |

[IF condition]      CALL      | (I4)      |  ;
                              | (I5)      |
                              | (I6)      |
                              | (I7)      |
                              | <address> |

IF | FLAG_IN     |            CALL             <address> ;
   | NOT FLAG_IN |

[IF condition]      RTS ;

[IF condition]      RTI ;

DO <address>  [UNTIL termination] ;

IDLE [(n)];
```

Table B7: Program flow control instructions

RTS (return from subroutine) and RTI (return from interrupt) provide for conditional return from CALL or interrupt vectors respectively. Table B7 gives a summary of all program flow control instructions. The condition codes are described in Table B8.

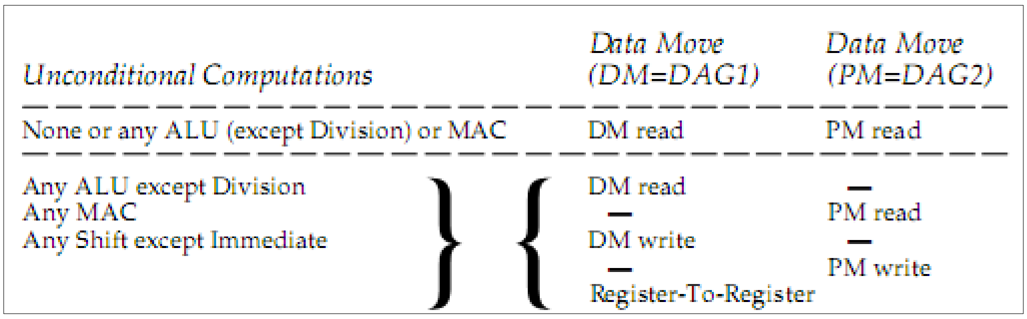


Syntax	*Status Condition*	*True If:*
EQ	Equal Zero	AZ = 1
NE	Not Equal Zero	AZ = 0
LT	Less Than Zero	AN .XOR. AV = 1
GE	Greater Than or Equal Zero	AN .XOR. AV = 0
LE	Less Than or Equal Zero	(AN .XOR. AV) .OR. AZ = 1
GT	Greater Than Zero	(AN .XOR. AV) .OR. AZ = 0
AC	ALU Carry	AC = 1
NOT AC	Not ALU Carry	AC = 0
AV	ALU Overflow	AV = 1
NOT AV	Not ALU Overflow	AV = 0
MV	MAC Overflow	MV = 1
NOT MV	Not MAC Overflow	MV = 0
NEG	X Input Sign Negative	AS = 1
POS	X Input Sign Positive	AS = 0
NOT CE	Not Counter Expired	

Table B8: IF condition codes

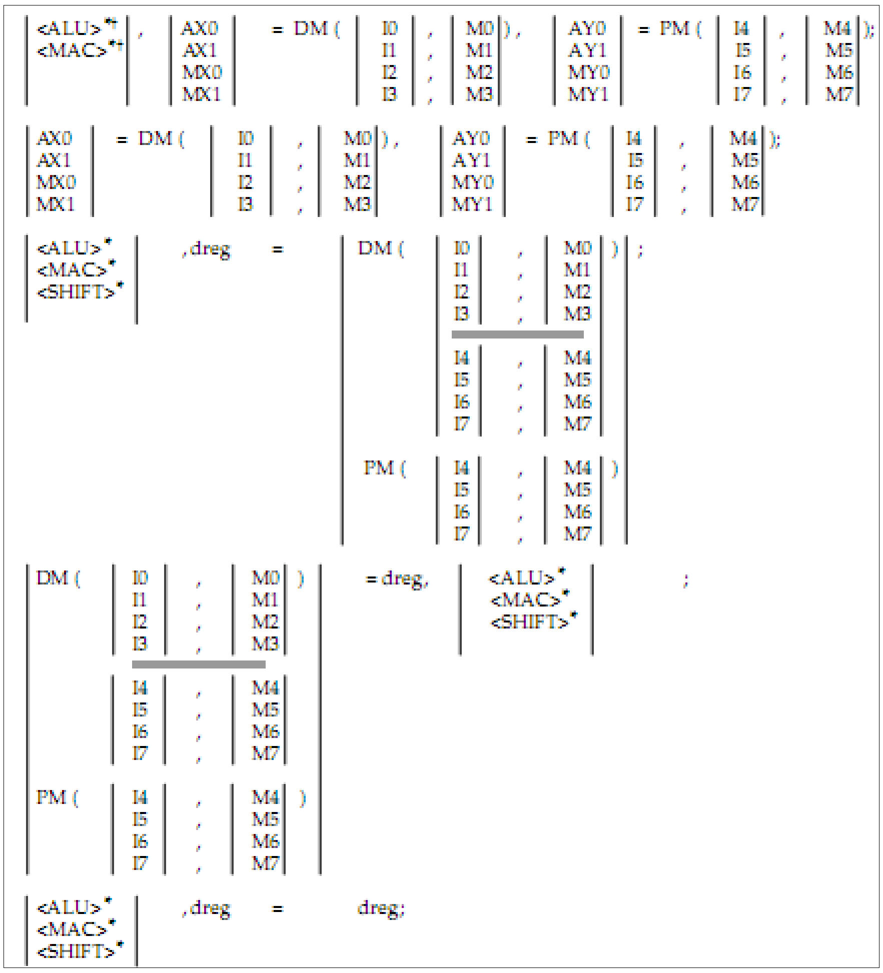
B.5 Multifunction instructions

Multifunction operations take advantage of the inherent parallelism of the ADSP-2100 family architecture by providing combinations of data moves, memory reads/ memory writes, and computation, all in a single cycle. Table B9 shows the legal combinations for multifunction instructions: you may combine operations on the same row with each other. And, finally, table B10 shows the syntax of the multifunction instructions.



Unconditional Computations	*Data Move (DM=DAG1)*	*Data Move (PM=DAG2)*
None or any ALU (except Division) or MAC	DM read	PM read
Any ALU except Division Any MAC Any Shift except Immediate	DM read	—
	—	PM read
	DM write	—
	—	PM write
	Register-To-Register	

Table B9: Valid combinations for multifunction instructions



```
| <ALU>*+ | ,  | AX0 |  = DM ( | I0 | , | M0 | ) ,  | AY0 |  = PM ( | I4 | , | M4 | );
| <MAC>*+ |    | AX1 |          | I1 | , | M1 |      | AY1 |          | I5 | , | M5 |
|         |    | MX0 |          | I2 | , | M2 |      | MY0 |          | I6 | , | M6 |
|         |    | MX1 |          | I3 | , | M3 |      | MY1 |          | I7 | , | M7 |

| AX0 |  = DM ( | I0 | , | M0 | ) ,  | AY0 |  = PM ( | I4 | , | M4 | );
| AX1 |          | I1 | , | M1 |      | AY1 |          | I5 | , | M5 |
| MX0 |          | I2 | , | M2 |      | MY0 |          | I6 | , | M6 |
| MX1 |          | I3 | , | M3 |      | MY1 |          | I7 | , | M7 |

| <ALU>*   | ,dreg  =  | DM ( | I0 | , | M0 | ) | ;
| <MAC>*   |           |      | I1 | , | M1 |   |
| <SHIFT>* |           |      | I2 | , | M2 |   |
|          |           |      | I3 | , | M3 |   |
|          |           |      | ------------- |   |
|          |           |      | I4 | , | M4 |   |
|          |           |      | I5 | , | M5 |   |
|          |           |      | I6 | , | M6 |   |
|          |           |      | I7 | , | M7 |   |
|          |           | PM ( | I4 | , | M4 | ) |
|          |           |      | I5 | , | M5 |   |
|          |           |      | I6 | , | M6 |   |
|          |           |      | I7 | , | M7 |   |

| DM ( | I0 | , | M0 | ) |  = dreg,  | <ALU>*   |   ;
|      | I1 | , | M1 |   |           | <MAC>*   |
|      | I2 | , | M2 |   |           | <SHIFT>* |
|      | I3 | , | M3 |   |
|      | ------------- |   |
|      | I4 | , | M4 |   |
|      | I5 | , | M5 |   |
|      | I6 | , | M6 |   |
|      | I7 | , | M7 |   |
| PM ( | I4 | , | M4 | ) |
|      | I5 | , | M5 |   |
|      | I6 | , | M6 |   |
|      | I7 | , | M7 |   |

| <ALU>*   | ,dreg  =  dreg;
| <MAC>*   |
| <SHIFT>* |
```

Table B10: Multifunction instructions

Appendix C

Implementation of the Levinson-Durbin algorithm

C.1 ETSI ANSI-C code

```
/*
*****************************************************************************
*                    MODULE INCLUDE FILE AND VERSION ID
*****************************************************************************
*/
#include "levinson.h"
const char levinson_id[] = "@(#)$Id $" levinson_h;

/*
*****************************************************************************
*                    INCLUDE FILES
*****************************************************************************
*/
#include <stdlib.h>
#include <stdio.h>
#include "typedef.h"
#include "basic_op.h"
#include "oper_32b.h"
#include "count.h"
#include "cnst.h"

int Levinson (
    LevinsonState *st,
    Word16 Rh[],      /* i : Rh[m+1] Vector of autocorrelations (msb) */
    Word16 Rl[],      /* i : Rl[m+1] Vector of autocorrelations (lsb) */
    Word16 A[],       /* o : A[m]    LPC coefficients  (m = 10)       */
    Word16 rc[]       /* o : rc[4]   First 4 reflection coefficients  */
)
{
    Word16 i, j;
    Word16 hi, lo;
    Word16 Kh, Kl;               /* reflexion coefficient; hi and lo      */
    Word16 alp_h, alp_l, alp_exp; /* Prediction gain; hi lo and exponent   */
    Word16 Ah[M + 1], Al[M + 1];  /* LPC coef. in double prec.            */
    Word16 Anh[M + 1], Anl[M + 1];/* LPC coef.for next iteration in double
                                 prec. */
    Word32 t0, t1, t2;           /* temporary variable                   */

    /* K = A[1] = -R[1] / R[0] */
```

```
t1 = L_Comp (Rh[1], Rl[1]);
t2 = L_abs (t1);                    /* abs R[1]        */
t0 = Div_32 (t2, Rh[0], Rl[0]);     /* R[1]/R[0]       */
test ();
if (t1 > 0)
   t0 = L_negate (t0);              /* -R[1]/R[0]      */
L_Extract (t0, &Kh, &Kl);           /* K in DPF        */

rc[0] = round (t0);                 move16 ();

t0 = L_shr (t0, 4);                 /* A[1] in         */
L_Extract (t0, &Ah[1], &Al[1]);     /* A[1] in DPF     */

/*  Alpha = R[0] * (1-K**2) */

t0 = Mpy_32 (Kh, Kl, Kh, Kl);       /* K*K             */
t0 = L_abs (t0);                    /* Some case <0 !! */
t0 = L_sub ((Word32) 0x7fffffffL, t0); /* 1 - K*K       */
L_Extract (t0, &hi, &lo);           /* DPF format      */
t0 = Mpy_32 (Rh[0], Rl[0], hi, lo); /* Alpha in        */

/* Normalize Alpha */

alp_exp = norm_l (t0);
t0 = L_shl (t0, alp_exp);
L_Extract (t0, &alp_h, &alp_l);     /* DPF format    */

/*--------------------------------------*
 * ITERATIONS  I=2 to M                 *
 *--------------------------------------*/

for (i = 2; i <= M; i++)
{
   /* t0 = SUM ( R[j]*A[i-j] ,j=1,i-1 ) +  R[i] */

   t0 = 0;                          move32 ();
   for (j = 1; j < i; j++)
   {
      t0 = L_add (t0, Mpy_32 (Rh[j], Rl[j], Ah[i - j], Al[i - j]));
   }
   t0 = L_shl (t0, 4);

   t1 = L_Comp (Rh[i], Rl[i]);
   t0 = L_add (t0, t1);             /* add R[i]        */

   /* K = -t0 / Alpha */

   t1 = L_abs (t0);
   t2 = Div_32 (t1, alp_h, alp_l); /* abs(t0)/Alpha              */
   test ();
   if (t0 > 0)
      t2 = L_negate (t2);          /* K =-t0/Alpha               */
   t2 = L_shl (t2, alp_exp);       /* denormalize; compare to Alpha */
   L_Extract (t2, &Kh, &Kl);       /* K in DPF                   */

   test ();
   if (sub (i, 5) < 0)
   {
      rc[i - 1] = round (t2);     move16 ();
   }
   /* Test for unstable filter. If unstable keep old A(z) */

   test ();
   if (sub (abs_s (Kh), 32750) > 0)
   {
      for (j = 0; j <= M; j++)
      {
         A[j] = st->old_A[j];        move16 ();
```

```
        }

        for (j = 0; j < 4; j++)
        {
          rc[j] = 0;              move16 ();
        }

        return 0;
      }
      /*------------------------------------------*
       *  Compute new LPC coeff. -> An[i]         *
       *  An[j]= A[j] + K*A[i-j]     , j=1 to i-1 *
       *  An[i]= K                                *
       *------------------------------------------*/

      for (j = 1; j < i; j++)
      {
        t0 = Mpy_32 (Kh, Kl, Ah[i - j], Al[i - j]);
        t0 = L_add(t0, L_Comp(Ah[j], Al[j]));
        L_Extract (t0, &Anh[j], &Anl[j]);
      }
      t2 = L_shr (t2, 4);
      L_Extract (t2, &Anh[i], &Anl[i]);

      /*  Alpha = Alpha * (1-K**2) */

      t0 = Mpy_32 (Kh, Kl, Kh, Kl);           /* K*K             */
      t0 = L_abs (t0);                        /* Some case <0 !! */
      t0 = L_sub ((Word32) 0x7fffffffL, t0);  /* 1 - K*K        */
      L_Extract (t0, &hi, &lo);               /* DPF format      */
      t0 = Mpy_32 (alp_h, alp_l, hi, lo);

      /* Normalize Alpha */

      j = norm_l (t0);
      t0 = L_shl (t0, j);
      L_Extract (t0, &alp_h, &alp_l);         /* DPF format    */
      alp_exp = add (alp_exp, j);             /* Add normalization to
                                       alp_exp */

      /* A[j] = An[j] */

      for (j = 1; j <= i; j++)
      {
        Ah[j] = Anh[j];                   move16 ();
        Al[j] = Anl[j];                   move16 ();
      }
    }

    A[0] = 4096;                          move16 ();
    for (i = 1; i <= M; i++)
    {
      t0 = L_Comp (Ah[i], Al[i]);
      st->old_A[i] = A[i] = round (L_shl (t0, 1));move16 (); move16 ();
    }

    return 0;
}
```

C.2 Implemented ADI code

```
.MODULE levinson_mod;

#include "gsmamr/cnst.h"
#include "gsmamr/temp_mem.h"
```

```
/* ******************** Global temporary memory ******************** */
/*
 *            Restrictions in this module:
 *            -  pm_tmp (30 - end) and all tmp can be used
 */

#define    Anh_Anl           pm_tmp + 30                   /* 2*(M+1) */
/*  To store Anh and Anl: Anh[0], Anl[0], Anh[1], Anl[1]... */

                                                       /* TOTAL USED: 22 (+30) */

#define    cnt               tmp                          /* 1 */
#define    pt_Rh             tmp + 1                      /* 1 */
#define    pt_Rl             tmp + 2                      /* 1 */
#define    pt_A              tmp + 3                      /* 1 */
#define    alp_h             tmp + 4                      /* 1 */
#define    alp_l             tmp + 5                      /* 1 */
#define    alp_exp           tmp + 6                      /* 1 */

                                                       /* TOTAL USED: 7 */

/* ***************************************************************** */

.VAR/DM/CIRC Ah_Al[2 * (M+1)];
  /*  To store Ah and Al: Ah[0], Al[0], Ah[1], Al[1]... */

.EXTERNAL div_32;
.EXTERNAL ses_amrst__olda;

.ENTRY levinson;

/* ***************************************************************** */
/* ***************************************************************** */
/*
 *      Function: levinson
 *
 *      Parameters:
 *
 *             *st    = (global variable)
 *
 *             Rh[]   = I6
 *             Rl[]   = I7
 *             A[]    = I0
 */

levinson:

                      M7 = -1;
                      AR = PASS 1,                 MY0 = PM(I6,M5);
                      MR = 0,                      MX0 = PM(I7,M5);
                      MR1 = PM(I6,M7);
                      MY1 = PM(I7,M7);
                      MR = MR + AR * MY1 (SS);

                      AR = PASS MR1;
                      IF EQ AR = ABS MR0;
                      SI = AR; /* SI stores the sign of t1 for further use */

                      AR = PASS MR1;
                      IF GE JUMP lev2;

lev1:                 DIS AR_SAT;
                      AR = - MR0;
                      ENA AR_SAT;
                      MR0 = AR,                    AR = -MR1 + C - 1;
                      MR1 = AR;

lev2:         /* t0 = Div_32 (t2, Rh[0], Rl[0]) */

                      /*  MR  /  MY0, MX0  */

                      CALL div_32; /* Returned SR */

                      AR = SI;
                      MR0 = SR0;
                      AR = PASS AR,                MR1 = SR1;
                      IF LE JUMP lev5;
                      DIS AR_SAT;
                      AR = - SR0;
                      ENA AR_SAT;
                      MR0 = AR,                    AR = -SR1 + C - 1;
                      MR1 = AR;

lev5:                 SE = -1;
                      SR = LSHIFT MR0 (LO),        MX1 = MR1;
                      MX0 = SR0;
                      /* MX1=Kh, MX0=Kl */

                      SE = -4;
                      SR = ASHIFT MR1 (HI);
                      SR = SR OR LSHIFT MR0 (LO);

                      I1 = ^Ah_Al + 2;
                      L1 = %Ah_Al;
                      SE = -1;
                      DM(I1,M1) = SR1,             SR = LSHIFT SR0 (LO);
                      DM(I1,M1) = SR0;

                      AR = PASS 1,                 MY1 = MX1;
                      MR = MX1 * MY1 (SS),         MY0 = MX0;
                      MF = MX1 * MY0 (SS);
                      MR = MR + AR * MF (SS);
                      IF MV SAT MR;
                      MF = MX0 * MY1 (SS);
                      MR = MR + AR * MF (SS);
                      IF MV SAT MR;

                      AR = PASS MR1;        /*  L_abs (t0)  */
                      IF GE JUMP lev7;

lev6:                 DIS AR_SAT;
                      AR = - MR0;
                      ENA AR_SAT;
                      MR0 = AR, AR = -MR1 + C - 1;
                      MR1 = AR;

lev7:                 AF = PASS MR0,               AY1 = MR1;
                      AX0 = 0xFFFF;
                      AX1 = 0x7FFF;
                      DIS AR_SAT;
                      AR = AX0 - AF;
                      ENA AR_SAT;
                      MR0 = AR,                    AR = AX1 - AY1 + C - 1;
                      MR1 = AR;

                      MY1 = MR1;
                      SR = LSHIFT MR0 BY -1 (LO);
                      /*  MY1 = hi, SR0 = lo  */

                      AR = PASS 1,                 MX1 = PM(I6,M5);/*Rh[0]*/
                      MY0 = SR0,                   MR = MX1 * MY1 (SS);
                      MF = MX1 * MY0 (SS),         MX0 = PM(I7,M5);/*Rl[0]*/
                      MR = MR + AR * MF (SS);
                      IF MV SAT MR;
                      MF= MX0 * MY1 (SS);
                      MR = MR + AR * MF (SS);
                      IF MV SAT MR;

                      SE = EXP MR1 (HI);
                      SE = EXP MR0 (LO);
                      SR = NORM MR1 (HI);
                      SR = SR OR NORM MR0 (LO);
                      DM(alp_exp) = SE;

                      DM(alp_h) = SR1;
                      SR = LSHIFT SR0 BY -1 (LO);
                      DM(alp_l) = SR0;


               /********** Iterations i=2 to M **************/

                      DM(pt_Rh) = I6;       /*  &Rh[1]  */
                      DM(pt_Rl) = I7;       /*  &Rl[1]  */
```

```
                              CNTR = M - 1;
                              M3 = -3;
                              DM(cnt) = M1;    /* "cnt" is "i-1" in the C code  */

                              DO lev8 UNTIL CE;

                                     AR = DM(cnt);
                                     CNTR = AR;
                                     SR = ASHIFT AR BY 1 (LO);
                                     AY0 = ^Ah_Al;
                                     AR = SR0 + AY0;
                                     I1 = AR;          /*  I1 = &Ah_Al[2*(i-1)]  */

                                     SR = LSHIFT AR BY 100 (LO); /*  SR = 0  */
                                     I6 = DM(pt_Rh);
                                     I7 = DM(pt_Rl);

                                     DO lev8_1 UNTIL CE;

                                            AR = PASS 1,          MX1 = DM(I1,M1),MY1 =PM(I6,M5);

                                            MR = MX1 * MY1 (SS),  MX0 = DM(I1,M3),MY0 = PM(I7,M5);
                                            MF = MX0 * MY1 (SS),          AY0 = SR0;
                                            MR = MR + AR * MF (SS),       AY1 = SR1;
                                            IF MV SAT MR;
                                            MF = MX1 * MY0 (SS);
                                            MR = MR + AR * MF (SS);
                                            IF MV SAT MR;

                                            DIS AR_SAT;
                                            AR = MR0 + AY0;
                                            ENA AR_SAT;
                                            SR0 = AR,                     AR = MR1 + AY1 + C;
lev8_1:                              SR1 = AR;


                                     AR = 4;
                                     SE = AR,                     AR = PASS SR0;
                                     SR = ASHIFT SR1 (HI),        MR1 = PM(I6,M5);
                                     SR = SR OR LSHIFT AR (LO),   MX0 = PM(I7,M5);

                                     MR0 = 0;
                                     MY0 = 1;
                                     MR = MR + MX0 * MY0 (SS),    AY0 = SR0;

                                     DIS AR_SAT;
                                     AR = MR0 + AY0,              AY1 = SR1;
                                     ENA AR_SAT;
                                     MR0 = AR,                    AR = MR1 + AY1 + C;
                                     MR1 = AR;

                                     IF EQ AR = ABS MR0;
                                     SI = AR; /*  SI stores the sign of t0 for further usage  */


                                     AR = PASS MR1;
                                     IF GE JUMP lev8_6;

lev8_5:                              DIS AR_SAT;
                                     AR = - MR0;
                                     ENA AR_SAT;
                                     MR0 = AR,                    AR = -MR1 + C - 1;
                                     MR1 = AR;

lev8_6:        /* t0 = Div_32 (t1, alp_h, alp_l) */

                                     MY0 = DM(alp_h);
                                     MX0 = DM(alp_l);

                                     /*  MR  /  MY0, MX0  */

                                     CALL div_32; /* Returned SR */

                                     AR = SI;
                                     MR0 = SR0;
                                     AR = PASS AR,                 MR1 = SR1;
                                     IF LE JUMP lev8_9;

                                     DIS AR_SAT;
                                     AR = - SR0;
                                     ENA AR_SAT;
                                     MR0 = AR,                     AR = -SR1 + C - 1;
                                     MR1 = AR;

       lev8_9:                       AR = DM(alp_exp);
                                     AR = - AR;

                                     SE = AR;
                                     SR = ASHIFT MR1 (HI);
                                     SR = SR OR LSHIFT MR0 (LO);

       lev8_12:                      SE = -1;
                                     SI = SR1;
                                     AX0 = SR0, /* SI,AX0=t2 */    SR = LSHIFT SR0 (LO);
                                     AR = SI;
                                     /* SI=Kh, SR0=Kl */

                                     AY0 = 32750;
                                     AR = ABS AR,                  MX0 = SR0; /* MX0=Kl */
                                     AR = AR - AY0,                MX1 = SI; /* MX1=Kh */
                                     IF LE JUMP lev8_13;

                                            I5 = ^ses_amrst__olda; /* Unstable. Keep old A(z) */
                                            CNTR = M + 1;
                                            DO lev8_12_1 UNTIL CE;

                                                    AR = PM(I5,M5);
       lev8_12_1:                                  DM(I0,M1) = AR;

                      /*  pop stacks (because we exit the function without ending lev8 loop) */
                                            POP CNTR, POP PC, POP LOOP;

                                            RTS;


                              /* Compute new LPC coeff. -> An[i] */

       lev8_13:                      DM(pt_A) = I0;   /* Save I0 */
                                     AR = DM(cnt);
                                     CNTR = AR;
                                     SR = ASHIFT AR BY 1 (LO);
                                     AY0 = ^Ah_Al;
                                     AR = SR0 + AY0;
                                     I1 = AR;          /*  I1 = &Ah_Al[2*(i-1)]  */
                                     I0 = ^Ah_Al + 2;  /*  I0 = &Ah_Al[2]  */
                                     I5 = ^Anh_Anl + 2;  /*  I5 = &Anh_Anl[2]  */
                                     SE = -1;

                                     DO lev8_14 UNTIL CE;

                                            MR1 = DM(I0,M1);
                                            MR0 = 0;
                                            AR = DM(I0,M1);
                                            MY0 = 1;
                                            MR = MR + AR * MY0 (SS),      MY1 = DM(I1,M1);

                                            AY1 = MR1;
                                            AR = PASS 1,                  AY0 = MR0;
                                            MR = MX1 * MY1 (SS),          MY0 = DM(I1,M3);
                                            MF = MX1 * MY0 (SS);
                                            MR = MR + AR * MF (SS);
                                            IF MV SAT MR;
                                            MF = MX0 * MY1 (SS);
                                            MR = MR + AR * MF (SS);
                                            IF MV SAT MR;

                                            DIS AR_SAT;
                                            AR = MR0 + AY0;
                                            ENA AR_SAT;
                                            MR0 = AR,                     AR = MR1 + AY1 + C;
```

```
                                SR = LSHIFT MR0 (LO),        PM(I5,M5) = AR;  /* Save
Anh[j] */
lev8_14:                PM(I5,M5) = SR0; /* Save Anl[j] */


                        I0 = DM(pt_A);    /* Restore I0 */
                        SE = -4;
                        SR = ASHIFT SI (HI),                 SI = AX0;
                        SR = SR OR LSHIFT SI (LO);

                        PM(I5,M5) = SR1; /* Save Anh[i] */
                        SR = LSHIFT SR0 BY -1 (LO);
                        PM(I5,M5) = SR0; /* Save Anl[i] */

                /* Calculate Alpha */

                        AR = PASS 1,                         MY1 = MX1;
                        MR = MX1 * MY1 (SS),                 MY0 = MX0;
                        MF = MX1 * MY0 (SS);
                        MR = MR + AR * MF (SS);
                        IF MV SAT MR;
                        MF = MX0 * MY1 (SS);
                        MR = MR + AR * MF (SS);
                        IF MV SAT MR;

                        AR = PASS MR1;        /*  L_abs (t0)  */
                        IF GE JUMP lev8_16;

lev8_15:                DIS AR_SAT;
                        AR = - MR0;
                        ENA AR_SAT;
                        MR0 = AR,                    AR = -MR1 + C - 1;
                        MR1 = AR;

lev8_16:                AF = PASS MR0,               AY1 = MR1;
                        AX0 = 0xFFFF;
                        AX1 = 0x7FFF;
                        DIS AR_SAT;
                        AR = AX0 - AF;
                        ENA AR_SAT;
                        MR0 = AR,                    AR = AX1 - AY1 + C - 1;

                        SR = LSHIFT MR0 BY -1 (LO);
                        /*  AR = hi, SR0 = lo  */

                        MX1 = DM(alp_h);
                        MY1 = AR,                    AR = PASS 1;
                        MR = MX1 * MY1 (SS),         MY0 = SR0;
                        MF = MX1 * MY0 (SS);
                        MR = MR + AR * MF (SS);
                        IF MV SAT MR;
                        MX0 = DM(alp_l);
                        MF = MX0 * MY1 (SS);
                        MR = MR + AR * MF (SS);
                        IF MV SAT MR;

                        SE = EXP MR1 (HI);
                        SE = EXP MR0 (LO);
                        SR = NORM MR1 (HI);
                        SR = SR OR NORM MR0 (LO);

                        DM(alp_h) = SR1;
                        SR = LSHIFT SR0 BY -1 (LO);
                        DM(alp_l) = SR0;

                        AR = DM(alp_exp);
                        AY0 = SE;
                        AR = AR + AY0;
                        DM(alp_exp) = AR;

                        I1 = ^Ah_Al + 2;  /*  I1 = &Ah_Al[2]  */
                        I5 = ^Anh_Anl + 2;  /*  I5 = &Anh_Anl[2]  */
                        CNTR = DM(cnt); /* Actually, it's cnt+1, so an additional
                               iteration will be done manually after the loop */

                        DO lev8_17 UNTIL CE;
```

```
                                     AR = PM(I5,M5);  /*  Ah[j] = Anh[j]  */
                                     DM(I1,M1) = AR;
                                     AR = PM(I5,M5);  /*  Al[j] = Anl[j]  */

lev8_17:                     DM(I1,M1) = AR;

                             AR = PM(I5,M5);
                             DM(I1,M1) = AR;
                             AR = PM(I5,M5);
                             DM(I1,M1) = AR;

                             AR = DM(cnt);   /*  increment cnt  */
                             AR = AR + 1;
lev8: /* loop end */  DM(cnt) = AR;

                      DM(I0,M1) = 4096;      /*  A[0] = 4096  */

                      I1 = ^Ah_Al + 2;  /*  I1 = &Ah_Al[2]  */
                      I5 = ^ses_amrst__olda + 1;
                      CNTR = M;
                      DO lev9 UNTIL CE;

                             MR1 = DM(I1,M1);
                             MR0 = 0;
                             AR = DM(I1,M1);
                             MY0 = 1;
                             MR = MR + AR * MY0 (SS);

                             SR= ASHIFT MR1 BY 1 (HI);
                             SR= SR OR LSHIFT MR0 BY 1 (LO);

                             AR = SR0 + 0x8000;   /* rounding */
                             AR = SR1 + C;

                             DM(I0,M1) = AR;  /*  Save A[i]  */

lev9: /* loop end */  PM(I5,M5) = AR;  /*  Save old_A[i]  */


                      RTS;

/* ********************************************************************** */
/* ********************************************************************** */

.ENDMOD;
```

Bibliography

- [1] J.R.Deller, J.G.Proakis, John Hansen. *Discrete-Time processing of speech signals.* Wiley-IEEE Press. 1999.
- [2] L. R. Rabiner, R. W. Schafer. *Digital processing of speech signals.* Prentice-Hall. 1978.
- [3] F. J. Owens. *Signal processing of speech.* Macmillan New Electronics. 1993.
- [4] A. J. Rubio Ayuso. *Speech recognition and coding. New advances and trends.* Springer. 1995.
- [5] *ADSP-2100 familiy user's manual.* Analog Devices, Inc. 2003.
- [6] *Using the ADSP. Parts 1 and 2.* Analog Devices, Inc. 1998.
- [7] *EZ-KIT Lite reference manual.* Analog Devices, Inc. 2002.
- [8] Rulph Chassaing. *Digital signal processing with C and the TMS320C30.* Wiley-Interscience. 1992.
- [9] Sylvia Goldsmith. *Real-time systems development.* Butterworth-Heinemann. 2003.
- [10] Paul M. Embree. *C algorithms for real-time DSP.* Prentice-Hall. 1995.
- [11] Jaime González Guijarro. *Implementación del codificador de voz "Half-rate" según la norma GSM 6.20 sobre el DSP TMS320C31.* ETSETB. 1995.
- [12] Roger S. Pressman. *Ingeniería del software. Un enfoque práctico.* McGraw-Hill. 2002.
- [13] Frank Fallside, William A. Woods. *Computer speech processing.* Prentice-Hall. 1985.
- [14] *ETSI EN 301 703: Digital cellular telecommunications system (Phase 2+); Adaptive Multi-Rate (AMR); Speech processing functions; General description.* European Telecommunications Standards Institute. 1999.

- [15] *ETSI EN 301 704: Digital cellular telecommunications system (Phase 2+); Adaptive Multi-Rate (AMR) speech transcoding.* European Telecommunications Standards Institute. 1999.
- [16] *ETSI EN 301 705: Digital cellular telecommunications system (Phase 2+); Substitution and muting of lost frames for Adaptive Multi Rate (AMR) speech traffic channels.* European Telecommunications Standards Institute. 1999.
- [17] *ETSI EN 301 706: Digital cellular telecommunications system (Phase 2+); Comfort noise aspects for Adaptive Multi-Rate (AMR) speech traffic channels.* European Telecommunications Standards Institute. 1999.
- [18] *ETSI EN 301 707: Digital cellular telecommunications system (Phase 2+); Discontinuous Transmission (DTX) for Adaptive Multi-Rate (AMR) speech traffic channels* European Telecommunications Standards Institute. 1999.
- [19] *ETSI EN 301 708: Digital cellular telecommunications system (Phase 2+); Voice Activity Detector (VAD) for Adaptive Multi-Rate (AMR) speech traffic channels;* European Telecommunications Standards Institute. 1999.
- [20] *ETSI EN 301 712: Digital cellular telecommunications system (Phase 2+); Adaptive Multi Rate (AMR) speech; ANSI-C code for the AMR speech codec.* European Telecommunications Standards Institute. 1999.
- [21] *ETSI EN 301 713: Digital cellular telecommunications system (Phase 2+); Test sequences for the Adaptive Multi-Rate (AMR) speech codec.* European Telecommunications Standards Institute. 1999.
- [22] GSM Association. http://www.gsmworld.com
- [23] European Telecommunications Standards Institute. http://www.etsi.org

Terms & Acronyms

1G

The first generation of analogue mobile phone technologies including AMPS, TACS and NMT

2G

The second generation of digital mobile phone technologies including GSM, CDMA IS-95 and D-AMPS IS-136

2.5G

The enhancement of GSM which includes technologies such as GPRS

3G

The third generation of mobile phone technologies covered by the ITU IMT-2000 family

3GPP

The 3rd Generation Partnership Project, a grouping of international standards bodies, operators and vendors with the responsibility of standardising the WCDMA based members of the IMT-2000 family

ADI

Analog Devices' proprietary assembling language, used by their family of DPS processors

ADPCM

Adaptive Differential Pulse Code Modulation; a form of voice compression that typically uses 32kbit/s

AMR

Adaptive Multi-Rate codec. Developed in 1999 for use in GSM networks, the AMR has been adopted by 3GPP for 3G

ANSI

American National Standards Institute. An non-profit making US organisation which does not carry out standardisation work but reviews the work of standards bodies and assigns them category codes and numbers

CDMA

Code Division Multiple Access; also known as spread spectrum, CDMA cellular systems utilise a single frequency band for all traffic, differentiating the individual transmissions by assigning them unique codes before transmission. There are a number of variants of CDMA (see W-CDMA, B-CDMA, TD-SCDMA et al)

CELP

Code Excited Linear Prediction; an analogue to digital voice coding scheme, there are a number of variants used in cellular systems

Codec

A word formed by combining coder and decoder the codec is a device which encodes and decodes signals. The voice codec in a cellular network converts voice signals into and back from bit strings. In GSM networks, in addition to the standard voice codec, it is possible to implement Half Rate (HR) codecs and Enhanced Full Rate (EFR) codecs

D/A

Digital to Analogue conversion

DAC

Digital to Analogue Convertor

DPCM

Differential Pulse Code Modulation

DSP

Digital Signal Processing

DTX

Discontinuous Transmission

Duplex

The wireless technique where one frequency band is used for traffic from the network to the subscriber (the downlink) and another, widely separated, band is used for traffic from the subscriber to the network (the uplink)

EDGE

Enhanced Data rates for GSM Evolution; effectively the final stage in the evolution of the GSM standard, EDGE uses a new modulation schema to enable theoretical data speeds of up to 384kbit/s within the existing GSM spectrum. An alternative upgrade path towards 3G services for operators, such as those in the USA, without access to new spectrum. Also known as Enhanced GPRS (E-GPRS)

EFR

Enhanced Full Rate; a alternative voice codec that provides improved voice quality in a GSM network (see codec)

ETSI

European Telecommunications Standards Institute: The European group responsible for defining telecommunications standards

FDMA

Frequency Division Multiple Access-a transmission technique where the assigned frequency band for a network is divided into sub-bands which are allocated to a subscriber for the duration of their calls

GPRS

General Packet Radio Service; standardised as part of GSM Phase 2+, GPRS represents the first implementation of packet switching within GSM, which is a circuit switched technology. GPRS offers theoretical data speeds of up to 115kbit/s using multislot techniques. GPRS is an essential precursor for 3G as it introduces the packet switched core required for UMTS

GSM

Global System for Mobile communications, the second generation digital technology originally developed for Europe but which now has in excess of 71 per cent of the world market. Initially developed for operation in the 900MHz band and subsequently modified for the 850, 1800 and 1900MHz bands. GSM originally stood for Groupe Speciale Mobile, the CEPT committee which began the GSM standardisation process

HSCSD

High Speed Circuit Switched Data; a special mode in GSM networks which provides higher data throughput By cocatenating a number of timeslots, each delivering 14.4kbit/s, much higher data speeds can be achieved

ITU

International Telecommunications Union

ITU-T

ITU Telecommunications Standardisation Sector

PCM

Pulse Code Modulation; the standard digital voice format at 64kbit/s

QAM

Quadrature Amplitude Modulation

QCELP

Quadrature Code Excited Linear Prediction

RAM

Random Access Memory

RELP

Regular pulse Excitation Linear Prediction coding

TDMA

Time Division Multiple Access; a technique for multiplexing multiple users onto a single channel on a single carrier by splitting the carrier into time slots and allocating these on a as-needed basis

UMTS

Universal Mobile Telecommunications System; the European entrant for 3G; now subsumed into the IMT-2000 family as the WCDMA technology.

Vocoder

Voice coder

VoIP

Voice over Internet Protocol

VSELP

Vector Sum Excited Linear Prediction

WCDMA

Wideband CDMA; the technology created from a fusion of proposals to act as the European entrant for the ITU IMT-2000 family