NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Speech Codecs analysis, basic arithmetic operations profiling and efficient Hardware mapping

DIPLOMA THESIS

Michail Papanikolaou

Supervisor: Kiamal Z. Pekmestzi

Professor

Athens, July 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη βασικών αριθμητικών πράξεων που χρησιμοποιούνται από την οικογένεια των speech codecs και σχεδιασμός αριθμητικής μονάδας σε υλικό για αποδοτική υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μιχαήλ Παπανικολάου

Επιβλέπων: Κιαμάλ Ζ. Πεκμεστζή

Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5η Ιουλίου 2013.

Αθήνα, Ιούλιος 2013

.....	.....	.....
Κιαμάλ Πεκμεστζή	Δημήτριος Σούντρης	Γεώργιος Οικονομάκος
Καθηγητής	Επίκουρος Καθηγητής	Επίκουρος Καθηγητής

.....

ΜΙΧΑΗΛ ΠΑΠΑΝΙΚΟΛΑΟΥ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Table of Contents

Abstract

The purpose of the present diploma thesis is the hardware design of an arithmetic unit for efficient implementation of speech codecs. First of all, that was an idea of Mr. N. Moschopoulos who noticed that nowadays execution of speech codecs is done mainly by general purpose DSPs. It would be interesting to study arithmetic operations of speech codecs and search for an efficient arithmetic unit that it could work as (a part of) an Arithmetic and Logic Unit (ALU) dedicated to speech applications.

Initially, the method followed included studying of the basic principles of speech coding and the algorithms of selected speech codecs (chapter 1). That study was followed by the software implementation of the selected speech codecs by using C-programming combined with the software environment of Matlab (chapter 2). The next step was profiling of the arithmetic operations that take place on the selected speech codecs and the proposal of an efficient arithmetic unit (chapter 3). Finally, the proposed arithmetic unit was implemented in Verilog HDL and evaluated through simulation and synthesis (chapter 4).

Key Words

speech coding, waveform codecs, parametric codecs, hybrid codecs, G.711, G.726, G.722, G.722.2 G.723.1, G.729, iLBC, SILK, Opus, C programming with Matlab, profiling of arithmetic operations, arithmetic data dependencies, hardware design, Verilog HDL

Acknowledgements

I would really like to thank the supervisor professor Mr. K. Pekmestzi for his advices and his comments especially regarding the arithmetic unit implementation, the research associate Dr. N. Moshopoulos for his guidance, his willingness to share his experience and his ability to give solutions to a variety of problems that appeared during the whole period of this thesis, as well as the PhD students K. Tsoumanis, G. Zervakis and N. Eftaxiopoulos – Sarris for their willingness to provide me practical advices and invaluable help on the hardware implementations of the proposed arithmetic unit.

Περίληψη

Ο σκοπός της παρούσας διπλωματικής εργασίας είναι η σχεδίαση σε υλικό μιας αριθμητικής μονάδας για αποδοτική υλοποίηση των αλγορίθμων κωδικοποίησης – αποκωδικοποίησης φωνής. Αρχικά, αυτός ο σκοπός προέκυψε από μια σκέψη του κ. Ν. Μοσχόπουλου, ο οποίος είχε παρατηρήσει πως στις μέρες μας η υλοποίηση σε υλικό των αλγορίθμων κωδικοποίησης φωνής υλοποιείται κυρίως με Ψηφιακούς Επεξεργαστές (DSPs) γενικού σκοπού και σκέφτηκε πως θα ήταν ενδιαφέρον να μελετηθούν οι αριθμητικές πράξεις και να γίνει έρευνα για μια αποδοτική αριθμητική μονάδα η οποία θα μπορούσε να αποτελέσει μέρος μιας Αριθμητικής και Λογικής Μονάδας προσανατολισμένης ειδικά για την υλοποίηση των αλγορίθμων κωδικοποίησης φωνής.

Αρχικά, η μέθοδος που ακολουθήθηκε περιελάμβανε τη μελέτη των βασικών αρχών της κωδικοποίησης φωνής και των αλγορίθμων που επιλέχθηκαν (κεφάλαιο 1). Αυτή η μελέτη ακολουθήθηκε από την υλοποίηση των επιλεγόμενων αλγορίθμων σε λογισμικό με τη χρήση του προγραμματισμού σε C σε συνδυασμό με το περιβάλλον λογισμικού του Matlab (κεφάλαιο 2). Το επόμενο βήμα ήταν η μελέτη των αριθμητικών πράξεων που λαμβάνουν χώρα στους αλγορίθμους κωδικοποίησης – αποκωδικοποίησης φωνής και η πρόταση μιας αποδοτικής αριθμητικής μονάδας (κεφάλαιο 3). Τελικά, η προτεινόμενη αριθμητική μονάδα υλοποιήθηκε με τη γλώσσα περιγραφής υλικού Verilog και αξιολογήθηκε μέσω προσομοίωσης και σύνθεσης (κεφάλαιο 4).

Λέξεις – Κλειδιά

κωδικοποίηση φωνής, αλγόριθμοι κωδικοποίησης κυματομορφής, παραμετρικοί αλγόριθμοι κωδικοποίησης, υβριδικοί αλγόριθμοι κωδικοποίησης, G.711, G.726, G.722, G.722.2 G.723.1, G.729, iLBC, SILK, Opus, προγραμματισμός C με Matlab, ποσοτική και ποιοτική μελέτη αριθμητικών πράξεων, εξαρτήσεις αριθμητικών δεδομένων, σχεδίαση υλικού, Γλώσσα Περιγραφής Υλικού Verilog

Ευχαριστίες

Θα ήθελα πραγματικά να ευχαριστήσω τον επιβλέποντα καθηγητή κ. Κ. Πεκμεστζή για τις συμβουλές και τα σχόλια του, ειδικά σε σχέση με το ποια αριθμητική μονάδα έπρεπε να υλοποιήσουμε, το ερευνητικό συνεργάτη κ. Ν. Μοσχόπουλο για όλη την καθοδήγηση του, την προθυμία του να μοιραστεί την εμπειρία του, καθώς, και την ικανότητα του να δίνει λύσεις σε μια ποικιλία προβλημάτων που εμφανίστηκαν καθ' όλη τη διάρκεια εκπόνησης αυτής της εργασίας, όπως επίσης τους Υποψηφίους Διδάκτορες Κ. Τσουμάνη, Γ. Ζερβάκη και Ν. Ευταξιόπουλο – Σαρρή για την προθυμία τους να μου δώσουν πρακτικές συμβουλές και ανεκτίμητη βοήθεια κατά την υλοποίηση της προτεινόμενης αριθμητικής μονάδας.

1 Speech Codecs

1.1 Introduction

In general, ***speech coding*** *is a procedure to represent a digitized speech signal using as few bits as possible, maintaining at the same time a reasonable level of speech quality*. A not so popular name having the same meaning is ***speech compression***. Speech coding has matured to the point where it now constitutes an important application area of signal processing. Due to the increasing demand for speech communication, speech coding technology has received augmenting levels of interest from the research, standardization, and business communities. Advances in microelectronics and the vast availability of low-cost programmable processors and dedicated chips have enabled rapid technology transfer from research to product development; this encourages the research community to investigate alternative schemes for speech coding, with the objectives of overcoming deficiencies and limitations. The standardization community pursues the establishment of standard speech coding methods for various applications that will be widely accepted and implemented by the industry. The business communities capitalize on the ever-increasing demand and opportunities in the consumer, corporate, and network environments for speech processing products.

Speech coding is performed using numerous steps or operations specified as an ***algorithm***. *An* ***algorithm*** *is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.* Many signal processing problems—including speech coding—can be formulated as a well-specified computational problem; hence, a particular coding scheme can be defined as an algorithm. In general, an algorithm is specified with a set of instructions, providing the computational steps needed to perform a task. With these instructions, a computer or processor can execute them so as to complete the coding task. The instructions can also be translated to the structure of a digital circuit, carrying out the computation directly at the hardware level.

Figure 1.1 shows the block diagram of a speech coding system. The continuous time analog speech signal from a given source is digitized by a standard connection of filter (eliminates aliasing), sampler (discrete-time conversion), and analog-to-digital converter (uniform quantization is assumed).

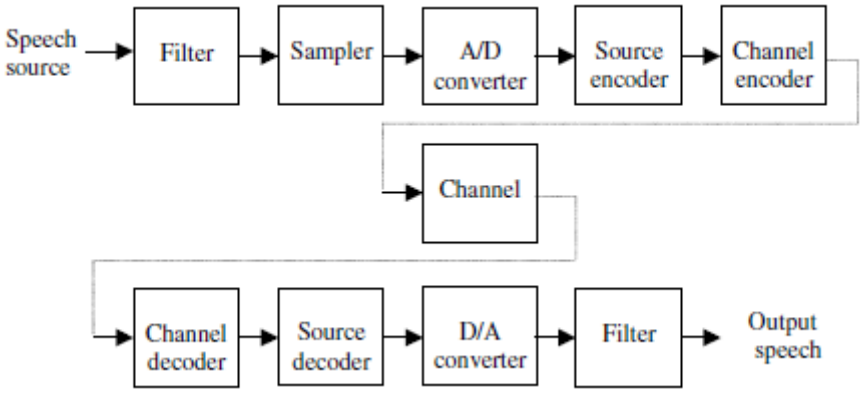


Figure 1.1: *Block diagram of a speech coding system*

The output is a discrete-time speech signal whose sample values are also discretized. This signal is referred to as the ***digital speech***. Traditionally, most speech coding systems were designed to support telecommunication applications, with the frequency contents limited between **300 and 3400 Hz**. According to the **Nyquist theorem**, the sampling frequency must be at least twice the bandwidth of the continuous-time signal in order to avoid aliasing. A value of **8 kHz** is commonly selected as the standard sampling frequency for speech signals. To convert the analog samples to a digital format using uniform quantization and maintaining toll quality—the digital speech will be roughly indistinguishable from the bandlimited input—more than 8 bits/sample is necessary. The use of 16 bits/sample provides a quality that is considered high. So, if the following parameters are assumed for the digital speech signal:

Sampling frequency = 8 kHz,

Number of bits per sample = 16

This gives rise to

Bit-rate = 8 kHz · 16 bits = 128 kbps

The above bit-rate, also known as ***input bit-rate***, is what the source encoder attempts to reduce (***Figure 1.1***). The output of the source encoder represents the ***encoded digital speech*** and in general has substantially lower bit-rate than the input. The codec G.723.1, for instance, can have an output rate of 5.3 kbps, a reduction of more than 24 times with respect to the input.

The encoded digital speech data is further processed by the channel encoder, providing error protection to the bit-stream before transmission to the communication channel, where various noise and interference can sabotage the reliability of the transmitted data. Even though in ***Figure 1.1*** the source encoder and channel encoder are separated, it is also possible to jointly implement them so that source and channel encoding are done in a single step.

The channel decoder processes the error-protected data to recover the encoded data, which is then passed to the source decoder to generate the output digital speech signal, having the original rate. This output digital speech signal is converted to continuous-time analog form through standard procedures: digital-to-analog conversion followed by antialiasing filtering.

In this thesis, the emphasis is on the source encoder and source decoder. For simplicity, they are referred to as the encoder and decoder, respectively (***Figure 1.2***).

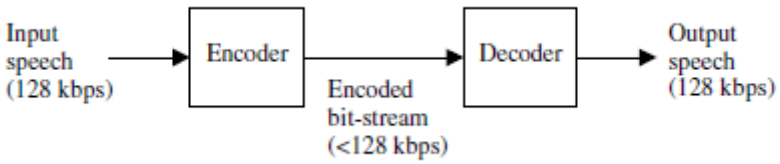


Figure 1.2: *Block diagram of a speech codec*

The input speech (a discrete-time signal having a bit-rate of 128 kbps) enters the encoder to produce the encoded bit-stream, or compressed speech data. Bit-rate of the bit-stream is normally much lower than that of the input speech. The decoder takes the encoded bit-stream as its input to produce the output speech signal, which is a discrete-time signal having the same rate as the input speech. Different methods provide differing speech quality and bit-rate, as well as implementation complexity.

The encoder/decoder structure presented in ***Figure 1.2*** is known as a ***speech codec***, where the input speech is encoded to produce a low-rate bit-stream. This bit-stream is input to the decoder, which constructs an approximation of the original signal.

Closing this section, the application of ***VoIP*** *(Voice over IP)* has to be referred. This is one of the main applications nowadays, where speech codecs are playing a very important role.

Codec	Coding Technique	Sampling Frequency (kHz)	Duration of Frame (ms)	Samples per frame	Bit Rate (kbit/s)	Complexity (MIPS)	Used Algorithms
G.711	Waveform	8	0.125	1	64	0.35	Companded PCM
G.726	Waveform	8	0.125	1	a)16, b)24, c)32, d)40	12	ADPCM
G.723.1	Hybrid	8	30	240	a) 5.3, b)6.3	19	a) ACELP, b) MPC-MLQ
iLBC	Hybrid	8	20 or 30	160 or 240	a) 15.2, b)13.3	a) 15, b)18	FB-LPC
G.729a	Hybrid	8	10	80	8	13	CS-ACELP
G.722	Waveform	16	0.0625	1	a) 48, b) 56, c) 64	10	SB-ADPCM
G.722.2	Hybrid	16	20	320	A variety of nine different Bit Rates from 6.60 to 23.85	38	ACELP

Table 1.1: *Speech Codecs of VoIP*

Table 1.1 displays some characteristics of the speech codecs that, almost all, IP Phones support. These characteristics are concerning:

- The **Coding Technique,** which is referred to the approach of the coding procedure that each codec follows. More are described in paragraph ***1.2.4***.
- The **Sampling Frequency,** which is referred to the frequency that the input speech is sampled.
- The **Duration of Frame,** which is referred to the duration of each speech frame that the codec processes while encoding/decoding.
- The **Samples of Frame,** which is referred to number of samples that each speech frame includes

- The **Bit Rate,** which is referred to the Bit Rate that the encoded speech is transmitted.
- The **Complexity,** which is referred to the complexity of each codec and it is measured in Million Instructions Per Second (MIPS).
- The **Used Algorithm,** which is referred to the digital processing algorithm that each codec implements.

It worths mentioning that 4 of the speech codecs of ***Table 1.1*** (*G.711, G.726, G.722, G.723.1)* were selected and analyzed within the framework of this thesis.

1.2 Speech Codecs Classification

The task of classifying modern speech codecs is not simple and is often confusing, due to the lack of clear separation between various approaches. This section presents some classification criteria. Readers must bear in mind that this is a constantly evolving area and new classes of codecs will be created while alternative techniques are introduced.

1.2.1 Classification by Sampling Frequency

Depending on the sampling frequency (f_s) of the input signal, that a speech codec can compress, the following categories can be recognized:

Category	Sampling Frequency (kHz)
Narrowband	8
Wideband	16
Super-Wideband	24
Ultra-Wideband	32
Fullband	48

Table 1.2: *Classification of Speech Codecs according to Sampling Frequency*

Figure 1.3 shows how higher sampling frequency gives a better *VoIP* service as it encodes more (actually, double) frequencies of the input speech. This is displayed as the codec *G.729* uses input speech that is sampled in 8 kHz and, on the other hand, codec G.722.2 uses input speech that is sampled in 16 kHz.

Figure 1.3: *Greater sampling frequency provides truer representation of speech*

Examples of the different categories from ***Table 1.2***, in relation to bit-rate, are displayed in ***Figure 1.4***. Generally, until now, the most of the codecs are Narrowband (G.711, G.726,

G.723.1, G.729 etc) or Wideband (G.722, G.722.2). The rest of the categories usually contain, some open-source codecs like *Opus* or *SILK* (speech codec of Skype).

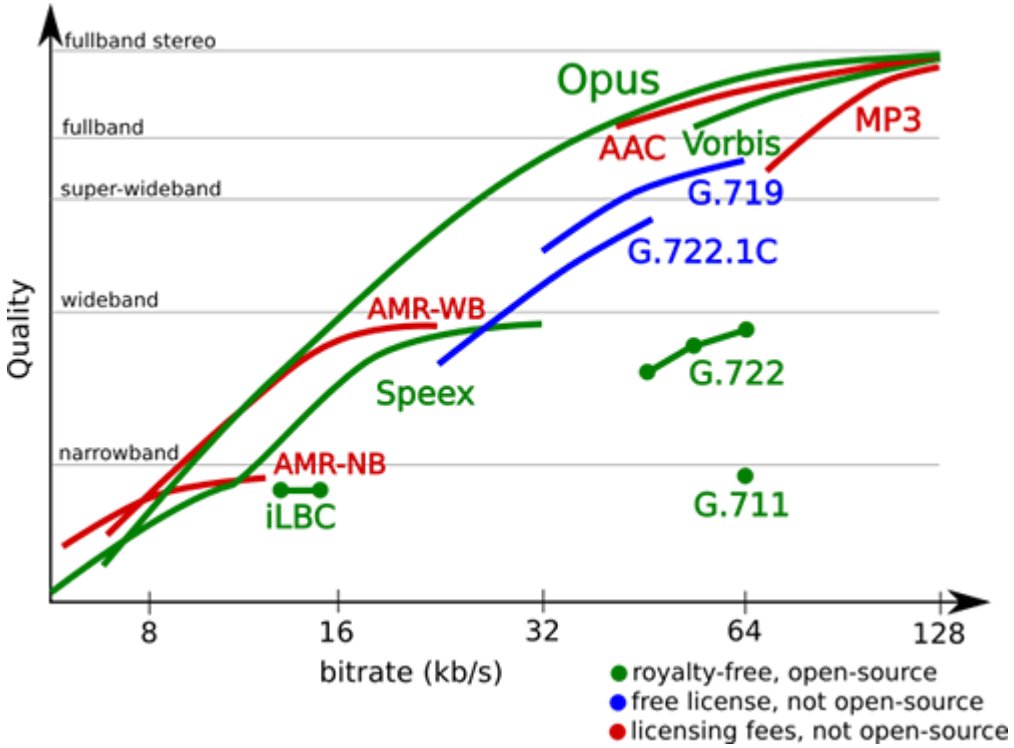


Figure 1.4: *Different categories of sampling frequency in relation to bit-rate*

1.2.2 Classification by Bit-Rate

All speech codecs are designed to reduce the reference bit-rate of 128 kbps to lower values. Based on the bit-rate of the encoded bit-stream, it is common to classify the speech codecs according to ***Table 1.3***. Generally, different coding techniques lead to different bit-rates. A given method works fine at a certain bit-rate range, but the quality of the decoded speech will drop radically if it is decreased below a certain threshold. The minimum bit-rate that speech codecs will achieve is limited by the information content of the speech signal. Current codecs can produce good quality at 2 kbps and above, suggesting that there is plenty of room for future improvement.

Category	Bit – Rate Range
High bit rate	>15 kbps
Medium bit rate	5 to 15 kbps
Low bit rate	2 to 15 kbps
Very Low bit rate	<2 kbps

Table 1.3: *Classification of Speech Codecs according to Bit – Rate*

1.2.3 Single-Mode and Multimode Codecs

Single-mode codecs are those that apply a specific, fixed encoding mechanism at all times, leading to a constant bit-rate for the encoded bit-stream. An example of such codecs is the pulse code modulation (PCM or *G.711*).

Multimode codecs were invented to take advantage of the dynamic nature of the speech signal, and to adapt to the time-varying network conditions. In this configuration, one of several distinct coding modes is selected, with the selection done by source control, when it is based on the local statistics of the input speech, or network control, when the switching obeys some external commands in response to network needs or channel conditions.

Figure 1.5 shows the block diagram of a multimode codec with source control. In this system, several coding modes are selected according to the properties of the signal at a given interval of time. In an open-loop system, the modes are selected by solely analyzing the input signal, while in a closed-loop approach, encoded outcomes of each mode are taken into account in the final decision. The mode selection information is transmitted as part of the bit-stream, which is used by the decoder to select the proper mode.

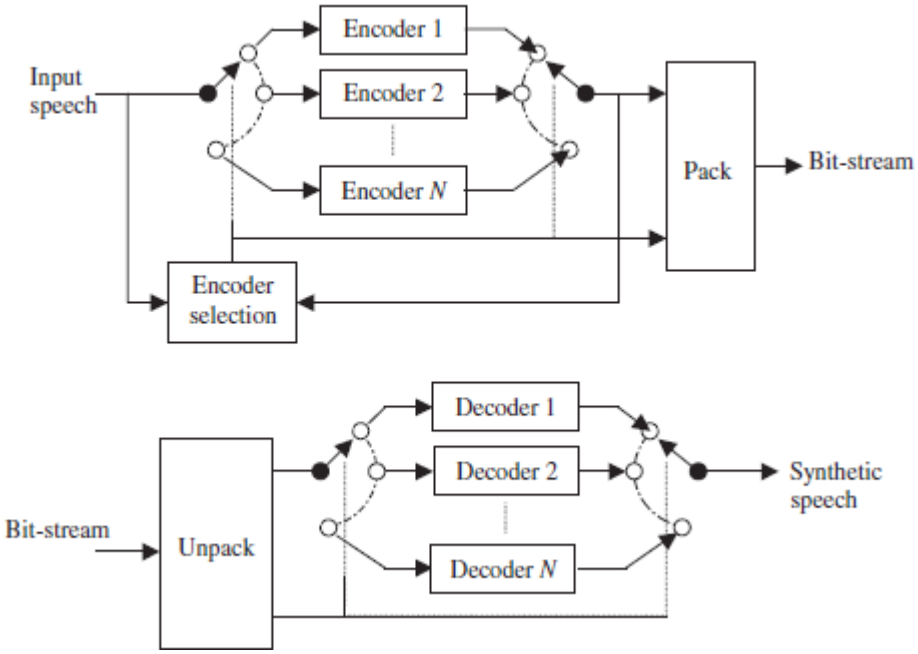


Figure 1.5: *Encoder (top) and decoder (bottom) of a source-controlled multimode codec*

Most multimode codecs have a variable bit-rate, where each mode has a particular, fixed value. Keeping the bit-rate varied allows more flexibility, leading to improved efficiency and a significant reduction in average bit-rate. Also, there are codecs which can adaptively switch between different sampling frequencies.

Examples of multimode codecs are *G.722.2*, *SILK* and *Opus*. Especially the last one (*Opus*), came up in 2012 and it can adaptively switch from 6 kb/s to 512 kb/s (as far as the bit-rate is concerned) and from Narrowband (8 kHz) to Fullband (48 kHz).

1.2.4 Classification by Coding Techniques

Waveform Codecs

An attempt is made to preserve the original shape of the signal waveform, and hence the resulting codecs can generally be applied to any signal source. These codecs are better suited for high bit-rate coding, since performance drops sharply with decreasing bit-rate. In practice, these codecs work best at a bit-rate of 32 kbps and higher. Signal-to-noise ratio (SNR) can be utilized to measure the quality of waveform codecs. Some examples of this

class include various kinds of pulse code modulation (PCM-G.711) and adaptive differential PCM (ADPCM-G.726).

Parametric Codecs

Within the framework of parametric codecs, the speech signal is assumed to be generated by a model, which is controlled by some parameters. During encoding, parameters of the model are estimated from the input speech signal, with the parameters being transmitted within the encoded bit-stream. This type of codec makes no attempt to preserve the original shape of the waveform, and hence SNR is a useless quality measure. Perceptual quality of the decoded speech is directly related to the accuracy and complexity of the underlying model. Due to this limitation, the codec is signal specific, having poor performance for non-speech signals.

There are several proposed models in the literature. The most successful, however, is based on **linear prediction**. In this approach, the human speech production mechanism is summarized using a time-varying filter, with the coefficients of the filter found using the linear prediction analysis procedure.

This class of codecs works well for low bit-rate. Increasing the bit-rate normally does not translate into better quality, since it is restricted by the chosen model. Typical bit-rate is in the range of 2 to 5 kbps. Example codecs of this class include linear prediction coding (LPC) and mixed excitation linear prediction (MELP).

Hybrid Codecs

As its name implies, a hybrid codec combines the strength of a waveform codec with that of a parametric codec. Like a parametric codec, it relies on a speech production model; during encoding, parameters of the model are located. Additional parameters of the model are optimized in such a way that the decoded speech is as close as possible to the original waveform, with the closeness often measured by a **perceptually weighted error signal**. As in waveform codecs, an attempt is made to match the original signal with the decoded signal in the time domain.

This class dominates the medium bit-rate codecs, with the code-excited linear prediction (CELP) algorithm and its variants (ACELP – G.723.1, CS-ACELP – G.729) the most outstanding representatives. From a technical perspective, the difference between a hybrid codec and a parametric codec is that the former attempts to quantize or represent the excitation signal

to the speech production model, which is transmitted as part of the encoded bit-stream. The latter, however, achieves low bit-rate by discarding all detail information of the excitation signal; only coarse parameters are extracted.

A hybrid codec tends to behave like a waveform codec for high bit-rate, and like a parametric codec at low bit-rate, with fair to good quality for medium bit-rate.

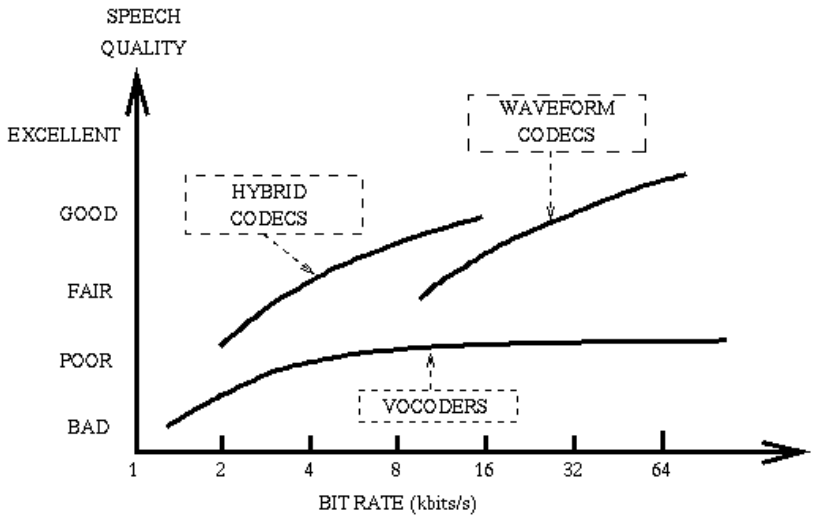


Figure 1.6: *Speech Quality versus Bit Rate for Common Classes of Codecs*

1.3 Speech Production and Modeling

The methods of Digital Signal Processing, that usually take place in speech coding and especially in parametric and hybrid speech codecs, are not about to be described in depth. The purpose is to outline the philosophy of parametric coding and how it was implemented, by understanding and modeling the human speech production system.

1.3.1 Origin of Speech Signals

The speech waveform is a sound pressure wave originating from controlled movements of anatomical structures making up the human speech production system. A simplified structural view is shown in ***Figure 1.7***. Speech is basically generated as an acoustic wave that is radiated from the nostrils and the mouth when air is expelled from the lungs with the resulting flow of air perturbed by the constrictions inside the body. It is useful to interpret speech production **in terms of acoustic filtering**. The three main cavities of the speech production system are nasal, oral, and pharyngeal forming **the main acoustic filter**. The filter is excited by the air from the lungs and is loaded at its **main output** by a radiation impedance associated with the lips.

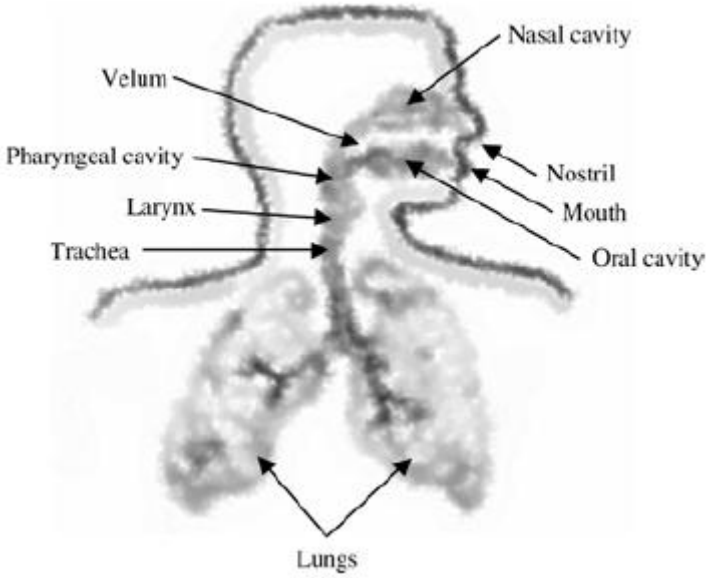


Figure 1.7: *Diagram of the human speech production system*

The form and shape of the vocal and nasal tracts change continuously with time, creating an acoustic filter with time-varying frequency response. As air from the lungs travels through

the tracts, the frequency spectrum is shaped by the frequency selectivity of these tracts. The resonance frequencies of the vocal tract tube are called **formant frequencies** or simply ***formants***, which depend on the shape and dimensions of the vocal tract.

Inside the larynx is one of the most important components of the speech production system—the vocal cords. Vocal cords are a pair of elastic bands of muscle and mucous membrane that open and close rapidly during speech production. The speed by which, the cords open and close **is unique for each individual** and define the feature and personality of the particular voice.

1.3.2 Classification of Speech Signals

Roughly speaking, a speech signal can be classified as voiced or unvoiced. ***Voiced sounds*** are generated when the vocal cords vibrate in such a way that the flow of air from the lungs is interrupted periodically, creating a sequence of pulses to excite the vocal tract. With the vocal cords stationary, the turbulence created by the flow of air passing through a constriction of the vocal tract generates unvoiced sounds. In time domain, *voiced sound is characterized by strong periodicity* present in the signal, with the fundamental frequency referred to as the *pitch frequency*, or simply **pitch**. For men, pitch ranges from 50 to 250 Hz, while for women the range usually falls somewhere in the interval of 120 to 500 Hz. Unvoiced sounds, on the other hand, do not display any type of periodicity and are essentially random in nature.

It is necessary to indicate that the voiced / unvoiced classification might not be absolutely clear for all frames, since during transitions (voiced to unvoiced or vice versa) there will be randomness and quasiperiodicity that is difficult to judge as strictly voiced or strictly unvoiced.

For most speech codecs, the signal **is processed on a frame-by-frame basis**, where a frame consists of a finite number of samples. The length of the frame is selected in such a way that the statistics of the signal remain almost constant within the interval. This length is typically between 20 and 30 ms, or 160 and 240 samples for 8-kHz sampling.

1.3.3 Modeling the Speech Production System

In general terms, ***a model* is a simplified representation of the real world**. It is designed to help the better understanding of the world and, ultimately, to duplicate many of the behaviors and characteristics of real-life phenomenon.

However, it is incorrect to assume that the model and the real world that it represents are identical in every way. In order for the model to be successful, it must be able to replicate partially or completely the behaviors of the particular object or fact that it intends to capture or simulate. The model may be a physical one (i.e., a model airplane) or it may be a mathematical one, such as a formula.

The human speech production system can be modeled using a rather simple structure*: the lungs*—generating the air or energy to excite the vocal tract—are represented by **a white noise source**. The *acoustic path inside the body* with all its components is associated with **a time-varying filter**. The concept is illustrated in ***Figure 1.8***. This simple model is indeed the core structure of many speech coding algorithms. By using a system identification technique called *linear prediction*, it is possible to estimate the parameters of the time-varying filter from the observed signal.

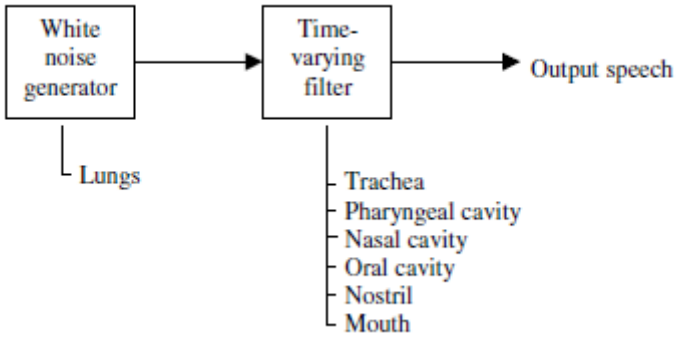


Figure 1.8: *Correspondence between the human speech production system with a simplified model based on time-varying filter.*

The assumption of the model is that the energy distribution of the speech signal in frequency domain is totally due to the time-varying filter, with the lungs producing an excitation signal having a flat-spectrum white noise. This model is rather efficient and many analytical tools have already been developed around the concept. The idea is the well-known *autoregressive model*.

1.3.4 A Glimpse of Parametric Speech Coding

Consider a speech frame corresponding to an unvoiced segment with 256 samples. Applying the samples of the frame to a linear prediction analysis procedure, the coefficients of an associated filter are found. This filter has system function

$$H(z) = \frac{1}{1 + \sum_{i=1}^{10} a_i z^{-i}},$$

with the coefficients denoted by α_i, i=1 to 10.

White noise samples are created using a unit variance Gaussian random number generator; when passing these samples (with appropriate scaling) to the filter, the output signal is obtained. ***Figure 1.9*** compares the original speech frame, with two realizations of filtered white noise. As can be seen, there is no time-domain correspondence between the three cases. However, when these three signal frames are played back to a human listener (converted to sound waves), the perception is almost the same!

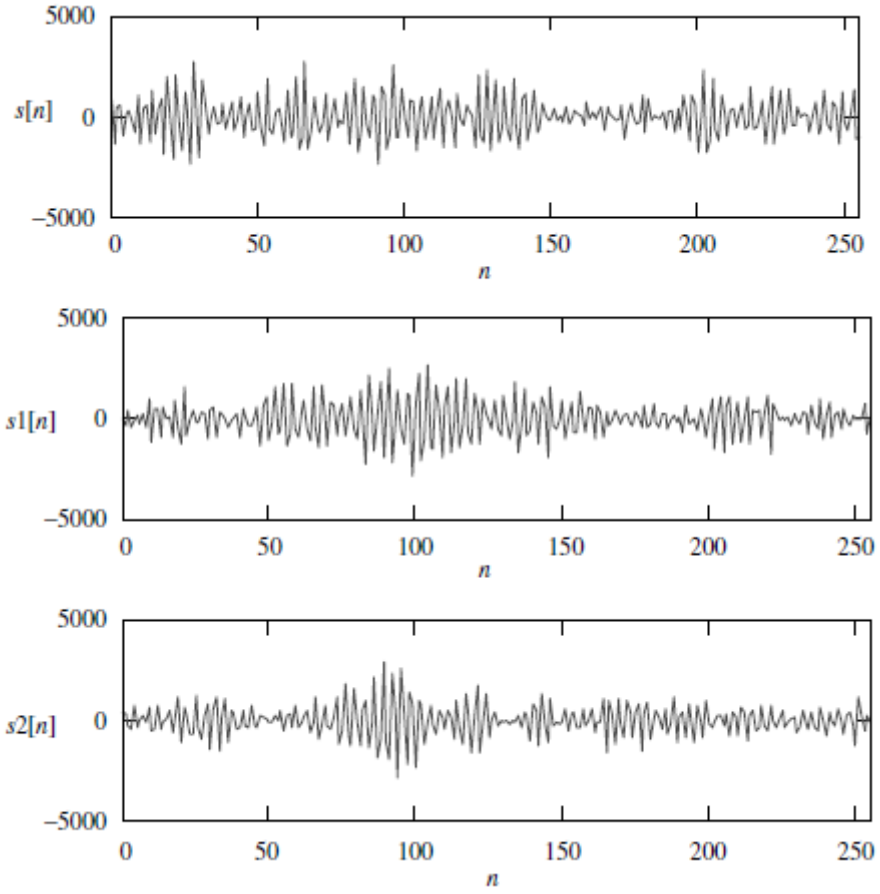


Figure 1.9: *Comparison between an original unvoiced frame (top) and two synthesized frames.*

Even if they look so different in the time domain, the perception is almost the same, because they all have a similar magnitude spectrum, as plotted in ***Figure 1.10***. As it can be noticed, the frequency contents are similar, and since the human auditory system is not very sensitive toward phase differences, all three frames sound almost identical. The original frequency spectrum is captured by the filter, with all its coefficients. Thus, the flat-spectrum white noise is shaped by the filter so as to produce signals having a spectrum similar to the original speech. Hence, linear prediction analysis is also known as a *spectrum estimation* technique.

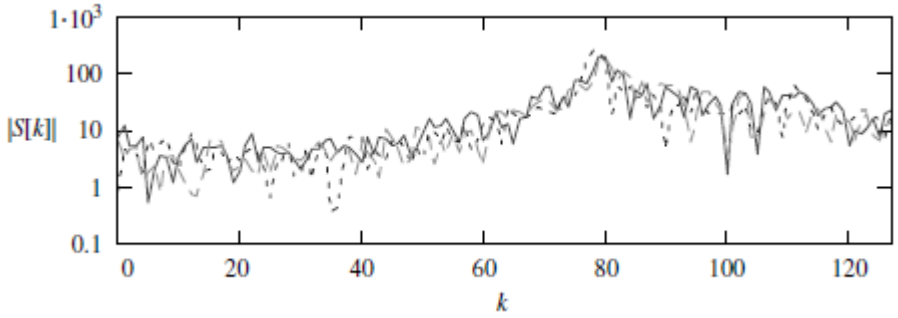


Figure 1.10: *Comparison between the magnitude of the DFT for the three signal frames of Figure 1.9*

It is known that the objective in speech coding is to represent the speech frame with a lower number of bits. The original number of bits for the speech frame is

Original number of bits = 256 samples x 16 bits/sample = 4096 bits.

As indicated previously, by finding the coefficients of the filter using linear prediction analysis, it is possible to generate signal frames having similar frequency contents as the original, with almost identical sounds. Therefore, the frame can be represented alternatively using ten filter coefficients, plus a scale factor. The scale factor is found from the power level of the original frame. In bibliography, it is mentioned that the set of coefficients can be represented with less than 40 bits, while 5 bits are good enough for the scale factor. This leads to

Alternative number of bits = 40 bits + 5 bits = 45 bits.

Therefore, an order of magnitude saving is achieved in terms of the number of required bits by using this alternative representation, fulfilling in the process our objective of bit reduction. This simple speech coding procedure is summarized below.

- **Encoding**

1) Derive the filter coefficients from the speech frame.
2) Derive the scale factor from the speech frame.
3) Transmit filter coefficients and scale factor to the decoder.

- **Decoding**

1) Generate white noise sequence.
2) Multiply the white noise samples by the scale factor.
3) Construct the filter using the coefficients from the encoder and filter the scaled white noise sequence. Output speech is the output of the filter.

By repeating the above procedures for every speech frame, a time-varying filter is created, since its coefficients are changed from frame to frame. Note that this overly simplistic scheme is for illustration only: much more elaboration is necessary to make the method useful in practice. However, the core ideas for many speech codecs are not far from this uncomplicated example.

1.4 Detailed Description of selected Codecs

1.4.1 G.711 (PCM)

In the early 1960's interest was expressed in encoding the analog signals in telephone networks, mainly to reduce costs in switching and multiplexing equipment and to allow the integration of communication and computing, increasing the efficiency in operation and maintenance.

In 1972, the then CCITT (renamed ITU-T in 1993) published the Recommendation G.711 that constitutes the principal reference as far as transmission systems are concerned. The basic principle of the algorithm is to code speech using 8 bits per sample, the input voiceband signal being sampled at 8 kHz, keeping the telephony bandwidth of 300–3400 Hz. With this combination, each voice channel requires 8kHz x 8 bits = 64 kbit/s.

There are two different versions of G.711, according to the region of the world in which it is implemented. These are **a-law** for Europe and **μ-law (or u-law)** for North America.

1.4.1.1 *Applications*

G.711 is considered to be a speech codec of high prevalence. Even if is the oldest one, the simplicity of his algorithm makes it ubiquitous, except, maybe, for the applications that bandwidth is really restricted. Every IP Phone, nowadays, supports G.711. Also, this codec is used in Radio over IP (*RoIP*) systems and in some Fax over IP devices (*FoIP*). **RoIP** is a method of communication that interconnects standard two-way radios with an IP network such as the public Internet and **FoIP** is the technology that enables the internetworking of fax machines with a packet-based network

1.4.1.2 *Parameters*

As it is mentioned above speech is sampled in 8 kHz sampling frequency. In the *a-law* version **input speech** is consisted of 13-bit uniform PCM numbers and in the *μ-law* version of 14-bit uniform PCM numbers. While encoding, numbers in the range of {-4092, 4092} (A-law) or in the range of {-8192, 8192} (μ-law) are assigned to an 8-bit form. On decoding, the 8-bit form comes back to the 13- or 14-bit form, according to law that is used.

1.4.1.3 *Algorithm*

The algorithm of G.711 is based on quantization. Maybe the most natural quantization scheme would be linear (or uniform) quantization. But one drawback of this approach is that the **signal-to-noise ratio (SNR)** varies with the amplitude of the input signals: the smaller the amplitude, the smaller the SNR. And, from the quality point of view, if a signal has a wide variance, or a variance that changes with time (as in the case of speech signals), the SNR will also change, resulting in a wide-varying quality of the system.

To avoid this problem logarithmic quantization was selected which will result into a more uniform quantization noise. With this in mind, several studies were carried out in late 1960's to choose a good algorithm for this purpose. This led to the definition of the two transmission schemes, one using the a-law compression characteristic:

$$f(x) = \begin{cases} \dfrac{A_0 \left| x \right|}{1 + \ln A_0} sgn(x), & \left| x \right| \le \dfrac{A}{A_0} \\ \dfrac{\mathrm{A}(1 + \ln(A_0 \mu |x|/A))}{1 + \ln A_0} sgn(x), & \dfrac{A}{A_0} \le \left| x \right| \le A \end{cases}$$

and the other using the μ-law compression characteristic:

$$f(x) = A \frac{\ln(1 + \mu \left| x \right| / A)}{\ln(1 + \mu)} sgn(x), \ \left| x \right| \le A.$$

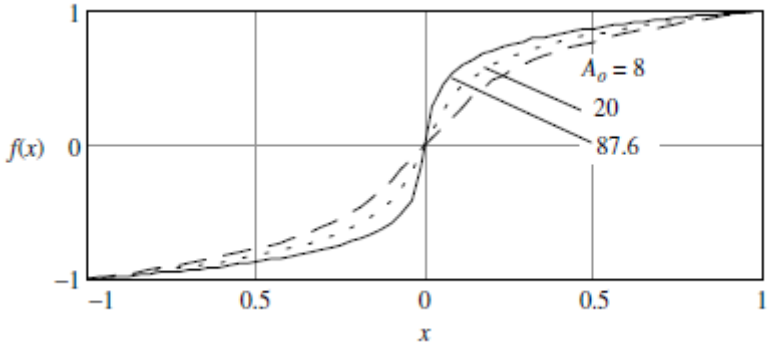


Figure 1.11: *Plots of a-law characteristics with A_0 = 87.6, 20, and 8. A = 1 for all cases*

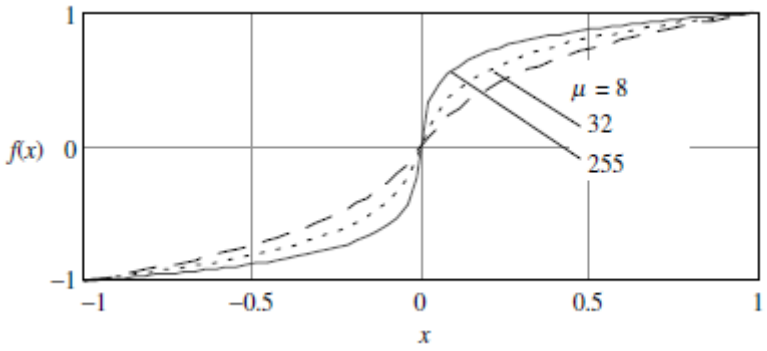


Figure 1.12: *Plots of μ-law characteristics with μ = 255, 32, and 8. A = 1 for all cases*

As it can be noticed in ***Figure 1.11*** and ***Figure 1.12***, lower value of A_0 or μ results to an almost linear quantization behavior. Also, both characteristics behave as linear for small amplitude signals (being then equivalent to a linear quantization scheme), but are truly logarithmic for large signals.

The G.711 standard does not specify the law as defined above, but rather uses a good linear-piecewise approximation for 8 bit samples, which has easier implementation (in hardware), as well as other properties.

This approximation uses bit 1 for sign (1 for positive, 0 for negative), bits 2–4 to indicate a segment, and bits 5–8 for level. Within each segment, the quantization is linear (4 bits, or 16 levels), having 13 segments of distinct slopes for a-law, and 15 for μ-law.

The a-law works with signals in the range from -4096 to 4096, implying in a range of 13 bits. As for the μ-law, the linear signals are accepted in the range -8159 to 8159, which is represented by 14 bits. Besides this, in the dynamic range sense, a- and μ- law are equivalent to 12 and 13 bit linear quantization, respectively.

One detail for the a-law is that the even bits are inverted. The reason for this comes from problems observed in transmission systems when long sequences of zeros happen, because small amplitudes, in a law, to be coded mostly using '0' bits. With this bit-inversion, long sequences of bits '0' become less probable, thus improving performance.

The conversion rule for a/μ-law from/to linear is described in terms of tables in G.711 (***Figure 1.13*** and ***Figure 1.14***). A good reason for this, is that there is no closed form for the compression of linear samples (although it is possible to find a closed formulae for the expansion algorithm). Hence, two implementations are possible: table look-up, and

algorithmic. For in-chip (LSI) implementations, the first one may be preferred, because it is simpler to implement, at the cost of a wider chip area. For other applications, such as using Digital Signal Processors (DSPs), or software implementations, table look-up would occupy too much memory, and the algorithmic solution would be preferred.

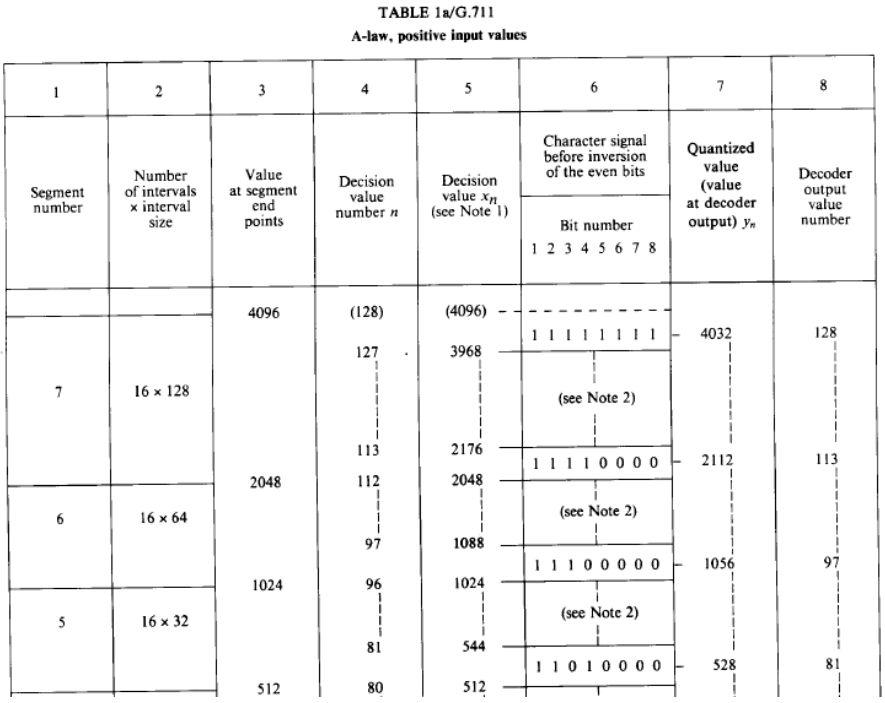


TABLE 1a/G.711

A-law, positive input values

1	2	3	4	5	6	7	8
Segment number	Number of intervals × interval size	Value at segment end points	Decision value number n	Decision value x_n (see Note 1)	Character signal before inversion of the even bits Bit number 1 2 3 4 5 6 7 8	Quantized value (value at decoder output) y_n	Decoder output value number
		4096	(128)	(4096)			
7	16 × 128		127	3968	1 1 1 1 1 1 1 1 (see Note 2)	4032	128
		2048	113	2176	1 1 1 1 0 0 0 0 (see Note 2)	2112	113
6	16 × 64		112	2048			
		1024	97	1088	1 1 1 0 0 0 0 0 (see Note 2)	1056	97
5	16 × 32		96	1024			
		512	81	544	1 1 0 1 0 0 0 0	528	81
			80	512			

Figure 1.13: *Table of conversion for a-law from/to liner (snapshot from ITU-T G.711 Recommendation)*

TABLEAU 2a / G.711

μ-law, positive input values

1	2	3	4	5	6	7	8
Segment number	Number of intervals × interval size	Value at segment end points	Decision value number n	Decision value x_n (see Note 1)	Character signal Bit number 1 2 3 4 5 6 7 8	Quantized value (value at decoder output) y_n	Decoder output value number
		8159	(128)	(8159)			
8	16 × 256		127	7903	1 0 0 0 0 0 0 0 (see Note 2)	8031	127
			113	4319	1 0 0 0 1 1 1 1	4191	112
7	16 × 128	4063	112	4063	(see Note 2)		
			97	2143	1 0 0 1 1 1 1 1	2079	96
6	16 × 64	2015	96	2015	(see Note 2)		
			81	1055	1 0 1 0 1 1 1 1	1023	80
5	16 × 32	991	80	991	(see Note 2)		

Figure 1.14: *Table of conversion for a-law from/to liner (snapshot from ITU-T G.711 Recommendation)*

1.4.2 G.726 (ADPCM)

In 1982, a group was established by CCITT to study the standardization of a speech coding technique that could reduce the 64 kbps rate used in digital links, as per ITU-T Recommendation G.711, by half (**32 kbps**) while maintaining the same voice quality.

After considering contributions received from several organizations, there was a general feeling that the **ADPCM** (Adaptive Differential Pulse Code Modulation) technique could provide a good quality codec. This process of finalizing an algorithm took 18 months of development and testing, to culminate in an ITU Recommendation, published in October, 1984 as Recommendation G.721.

Meanwhile, problems were found with the G.721 algorithm of 1984 regarding voice-band data and changes had to be done to the algorithm. These changes were approved in 1986 and published in the next series of Recommendations of the CCITT. Also, in that Study Period (1985-1988), a need for other rates was identified, and a new Recommendation, G.723, was approved to extend the bitrate to **24** and **40 kbps**.

In the Study Period of 1989–1992, these two Recommendations have been joined into a single one, keeping full compatibility with the former ones, and adding a lower rate of **16 kbps**. This new Recommendation was named G.726, and the former G.721 and G.723 have been replaced. The most commonly used mode is **32 kbps**, since this is half the rate of G.711, thus increasing the usable network capacity by 100%.

1.4.2.1 *Applications*

G.726 is present in the most IP phones. It is primarily used on international trunks in the phone network. G.726 is, also, the standard codec used in DECT cordless phone systems and on some Canon cameras. Finally, it is used, like G.711, in RoIP systems.

1.4.2.2 *Parameters*

In G.726 speech is, also, sampled in 8 kHz sampling frequency. Actually, it includes the processing of G.711 in the beginning of encoding (***Figure 1.15*** *– 64kbps PCM input*) and in the end of decoding (***Figure 1.16*** *– 64kbps PCM output*). As a result, if the *α-law* version of G.711 is included, **input speech** is consisted of 13-bit uniform PCM numbers and, if the *μ-law* version, of 14-bit uniform PCM numbers, respectively and they are assigned to an 8-bit form.

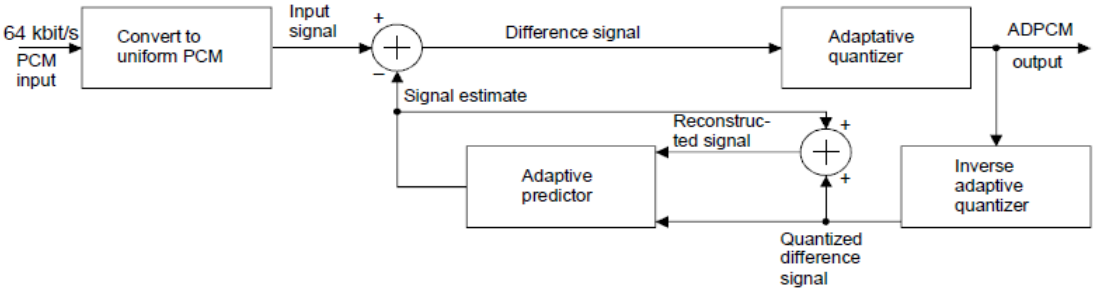


Figure 1.15: *G.726 Encoder*

Then, while encoding and according to the mode of G.726 that is used (40, 32, 24 or 16 kbps), 8-bit form is transformed to a 5- , 4-, 3- or 2-bit form, respectively, and this is the transmitted data.

Finally, decoder restores data to the 8-bit form and this is brought back to the 13- or 14-bit uniform PCM by the G.711 decoding part that is included.

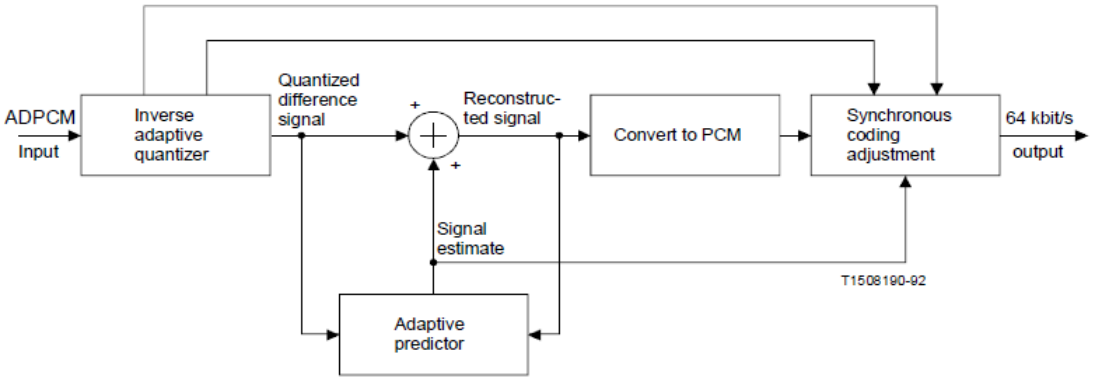


Figure 1.16: *G.726 Decoder*

1.4.2.3 *Algorithm*

In this section the algorithm of 32 kbps mode of G.726 is described, as applied at the most common applications nowadays. Other modes use almost the same algorithm.

The basic idea behind the G.726 codec is to code into **4-bit samples** the input speech-band signals, sampled at 8 kHz and represented by the **8-bit of G.711** *a-* or *μ- law* samples. The decoder just implements the reverse procedure.

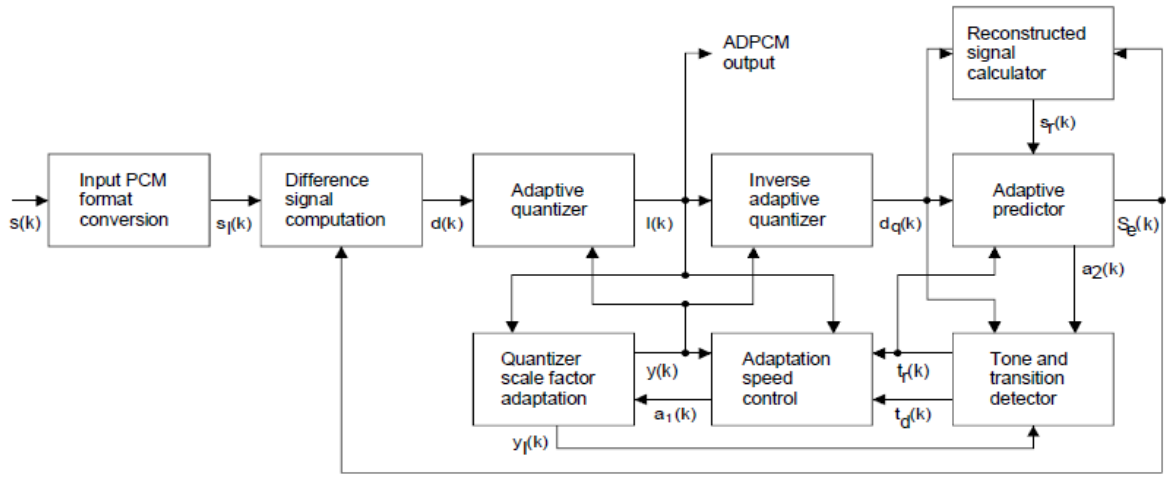


Figure 1.17: *G.726 Encoder Block Schematic*

The ADPCM algorithm of the G.726 exploits **the predictability** of the speech signals. Therefore, an adaptive predictor is used to compute the difference signal ***d(k)*** (based on the expanded input log-pcm sample ***s(k)***), which is then quantized by an adaptive quantizer

using 4 bits. These bits are sent to the decoder and then fed into an inverse quantizer. The difference signal is used to calculate the reconstructed signal, ***$s_r(k)$***, which is compressed (*a-* or *μ-law*) and output from the decoder ***$s_d(k)$***).

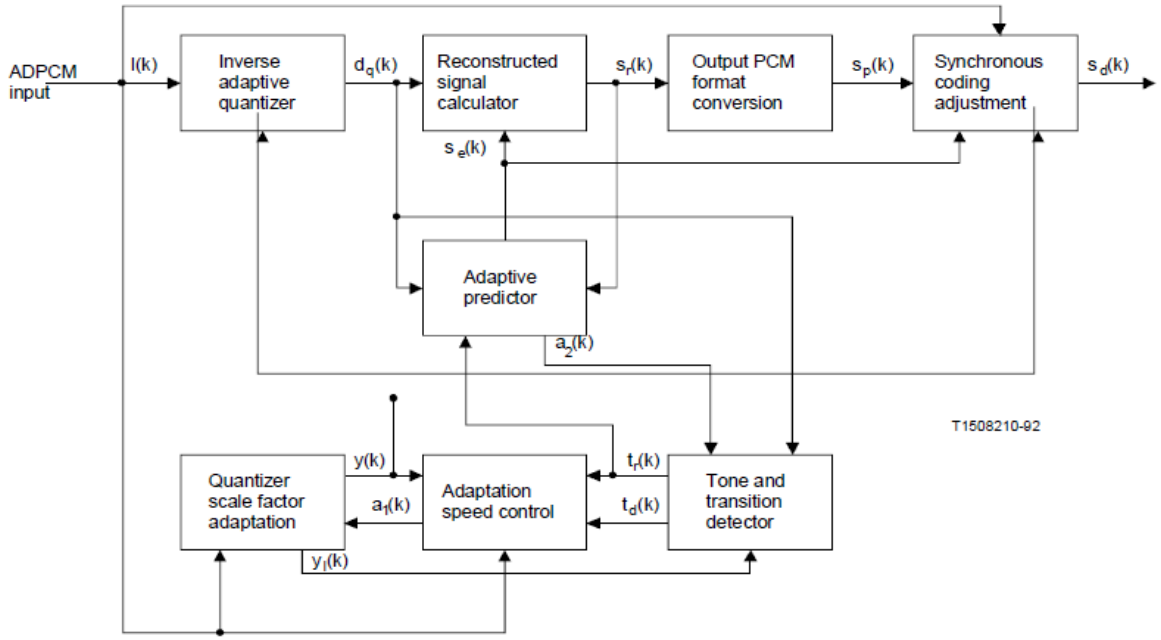


Figure 1.18: *G.726 Decoder Block Schematic*

To provide some insight in the building blocks of the G.726 algorithm, a short description of each of them is given.

PCM format conversion

The input signal ***s(k)***, in either *a-* or *μ*-law format, must be converted into linear samples. This expansion is accomplished using the same algorithm in G.711, but converting from signed magnitude to 14-bit two's complement samples.

Difference Signal Computation

This block simply calculates the difference between the (expanded) input signal and the estimated signal:

$d(k) = s_l(k) - s_e(k)$

Adaptive Quantizer

A 15-level, non-uniform adaptive quantizer is used to quantize the difference signal. Before the quantization, this signal is converted to a logarithmic representation (in order to take advantage of the fact that *multiplication* in linear domain is *addition* in logarithmic domain) and scaled by a factor (***y(k)***), that is computed in the scale factor adaptation block (see below).

The output of this block is ***I(k)***, and it is used twice; first, the ADPCM coded (quantized) sample; second, the input to the backward part of the G.726 algorithm, to provide information for quantization of the next samples. One relevant point to note here is that the backward adaptation is done using the quantized sample. If one starts the decoder from this very point, one will find identical behavior. That is why only the quantized samples are needed in the decoder.

Inverse Adaptive Quantizer

The inverse adaptive quantizer takes the signal ***I(k)*** and converts it back to the linear domain, generating a quantized version of the difference signal, $d_q(k)$. This is the input to the adaptive predictor, such that the estimated signal is based on a quantized version of the difference signal, instead of on the un-quantized (original) one.

Quantizer Scale Factor Adaptation

This block computes ***y(k)***, the factor used in the adaptive quantizer and inverse quantizer for domain conversion. As input, this block needs ***I(k)***, but also $a_l(k)$, the adaptation speed control parameter. The reason for the latter is that the scaling algorithm has two modes (bimodal adaptation), one fast and another slow. This has been done to accommodate signals that in nature produce difference signals with large fluctuations (e.g. speech) and small fluctuations (e.g. tones and voice-band data), respectively.

This block computes two scale factor (fast, $y_u(k)$, and slow, $y_l(k)$) based on ***I(k)***, which combined using $a_l(k)$ produce ***y(k)***.

Adaptation Speed Control

This block evaluates the parameter $a_l(k)$, which can be seen as a proportion of the speed (fast or slow) of the input signal, and is in the range [0, 1]. If 0, the data are considered to be *slowly* varying; if 1, they are considered to be *fast* varying.

To accomplish this, two measures of the average magnitude of $I(k)$ are computed ($d_{ms}(k)$ and $d_{ml}(k)$). These, in conjunction with delayed tone detect and transition detect flags ($t_d(k)$ and $t_r(k)$, calculated in the Tone Transition and Detector block), are used to evaluate $a_p(k)$, whose delayed version ($a_p(k - 1)$) is used in the definition of $a_l(k)$, limiting the range to [0, 1].

An analysis of $a_p(k)$ gives insight on the nature of the signal: if around the value of 2, this means that the average magnitude of $I(k)$ is changing, or that a tone has been detected, or that it is idle channel noise; on the other side, if near 0, the average magnitude of $I(k)$ remains relatively constant.

Adaptive Predictor and Reconstructed Signal Calculator

The adaptive predictor has as its main function to compute the signal estimate based on the quantized difference signal, $d_q(k)$. It has 6 zeroes and 2 poles, structure that covers well the kind of input signals expected for the algorithm. With these coefficients, and past values of $d_q(k)$ and $s_e(k)$, the updated value for the signal estimate $se(k)$ is computed.

The two sets of coefficients (one for the pole section, $a_i(k)$, i = 1,2, other for the zero section, $b_i(k)$, i = 1,..., 6) are updated using a simplified gradient algorithm. At this point, since a situation in which the poles cause instability may arise, the two pole coefficients a_i have their ranges limited. In addition, if a transition from partial band signal is detected (signaled by $t_r(k)$), the predictor is reset (all coefficients are set to 0), remaining disabled until $\mathbf{t_r}$ comes back to zero.

The reconstructed signal $s_r(k)$ is calculated using the signal estimate $s_e(k)$ and the quantized difference signal $d_q(k)$.

Tone Transition and Detector

This block was added to improve algorithm performance of G.721 for signals originating from FSK modems operating in the character mode. First, it checks if the signal has partial band (e.g., a tone) by looking at the predictor coefficient $a_2(k)$, that defines the signal $t_d(k)$. Second, a transition from partial band signal indicator $t_r(k)$ is set, such that predictor coefficients can be set to 0 and the quantizer can be forced into the fast mode of operation.

Output PCM Format Conversion

This block is unique to the decoder. Its sole function is to compress the reconstructed signal $s_r(k)$, which is in linear PCM format, using α- or μ-law, and is a complement of the PCM format conversion block.

Synchronous Coding Adjustment

This block is also unique to the decoder. It has been devised in order to prevent cumulative distortions occurring on synchronous tandem coding (ADPCM–PCM–ADPCM, etc., in purely digital connections, i.e., with no intermediate analog conversions), provided that:

- the transmission of the ADPCM and the intermediate PCM are error-free, and
- the ADPCM and the intermediate PCM are not disturbed by digital signal processing devices.

Extension for linear input and output signals

An extension of the G.726 algorithm was carried out in 1994 to include, as an option, linear input and output signals. The specification for such linear interface is given in its Annex A. This extension bypasses the PCM format conversion block for linear input signals, and both the Output PCM Format Conversion and the Synchronous Coding Adjustment blocks, for linear output signals. These linear versions of the input and output signals are 14-bit, 2's complement samples.

The effect of removing the PCM encoding and decoding is to decrease the coding degradation by 0.6 to 1 qdu, depending on the network configuration considered.

1.4.3 G.722 (SB – ADPCM)

With the emergence of ISDN networks offering digital connectivity at 64 kbps between subscribers, the possibility was given to improve the standard telephone quality by increasing the transmitted bandwidth. A bandwidth of 50-7000 Hz corresponding to a sampling of 16 kHz was chosen because it provides a substantial improvement of the quality for applications where the speech is to be heard through high quality loudspeakers e.g. for audio or video conference services, commentary broadcasting, and high quality hands-free phones.

An expert group was created in November 1983 whose mandate was to define a standard for 7 kHz speech coding within 64 kbps. After many contributions received from several organizations, it has been decided to choose a codec which combined sub-band filtering and adaptive differential pulse-code modulation algorithms (SB-ADPCM). The final recommendation was produced in March 1986 and approved in July 1986 by CCITT as Recommendation G.722.

In 2006, upon ETSI DECT request, packet loss concealment (PLC) procedures for G.722 were standardized to ensure a sufficient robustness over the DECT wireless interface.

1.4.3.1 *Applications*

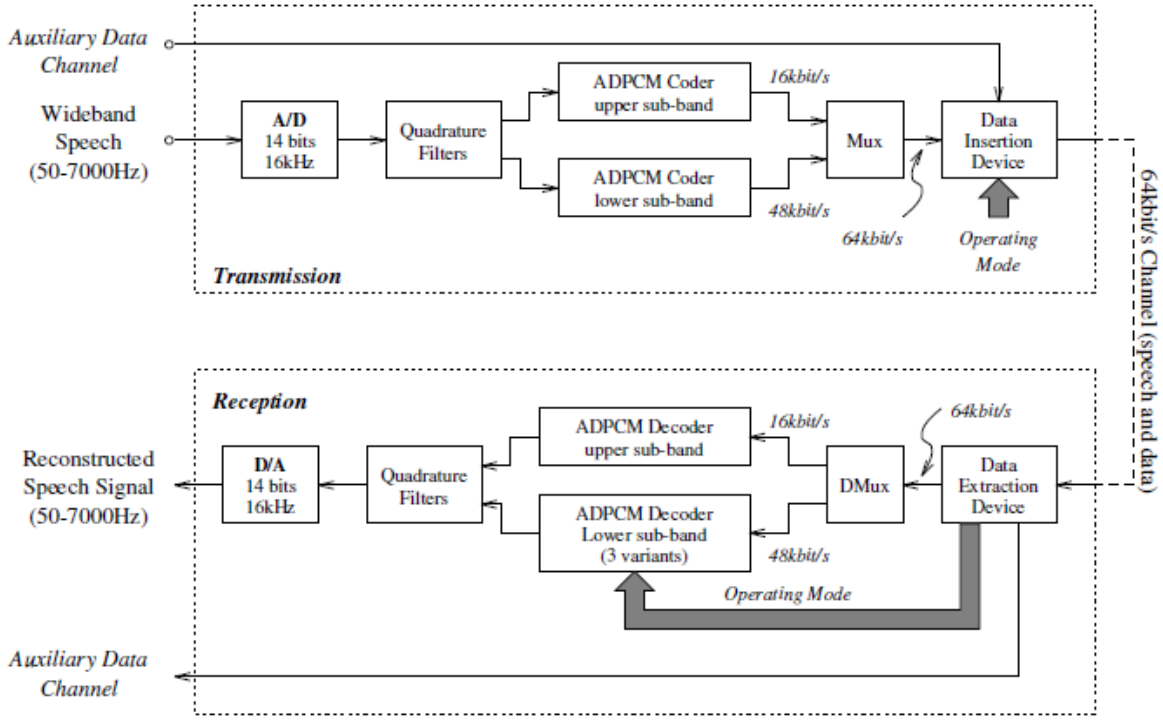
G.722 is present in the most IP phones. It is used for *VoIP* applications, such as on a local area network where network bandwidth is readily available, and offers a significant improvement in speech quality over older narrowband codecs such as G.711. G.722 has also been widely used by radio broadcasters for sending commentary grade audio over a single 56 or 64 kbps/s ISDN B-channel. Finally, G.722 at 64 kbps/s is the mandatory codec in ETSI New Generation DECT (NG-DECT) standards. These standards are intended for wideband audio enabled devices to be connected on VoIP networks.

1.4.3.2 *Parameters*

As it is mentioned above speech is sampled in 16 kHz sampling frequency. ***Figure 1.19*** shows the block diagrams of encoding-decoding, and explains the 3 different modes of G.722 (64, 56, 48 kbps).

In order to improve the transmitted speech quality, the input signal has to be converted after antialiasing filtering by an analog-to-digital (A/D) converter operating at 16 kHz

sampling rate and with a resolution of at least 14 uniform PCM bits. Input, is encoded to an 8-, 7- or 6-bit form, according to the operating mode, and, similarly, at the receive side, a digital-to-analog (D/A) converter operating at 16 kHz sampling rate and with a resolution of at least 14 uniform PCM bits should be used.



Notes:

* Operating modes:

Mode 1:	64kbit/s for speech and 0kbit/s for auxiliary data
Mode 1-bis:	56kbit/s for speech and 0kbit/s for auxiliary data
Mode 2:	56kbit/s for speech and 8kbit/s for auxiliary data
Mode 3:	48kbit/s for speech and 16kbit/s for auxiliary data
Mode 3-bis:	48kbit/s for speech, 6.4kbit/s for auxiliary data and 1.6kbit/s for service channel framing and mode control

* Operating modes 1-bis and 3-bis are applicable only to US national 56 kbit/s networks

* The signal in the 64kbit/s channel comprises 64, 56 or 48 kbit/s for speech and 0, 8 or 16 kbit/s for data, depending on the operating mode.

Figure 1.19: *G.722 Encoder and Decoder Block Diagrams*

1.4.3.3 *Algorithm*

Figure 1.20 shows block diagram of the SB-ADPCM encoder which comprises the following main blocks.

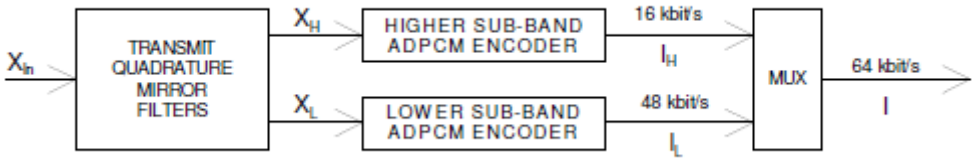


Figure 1.20: *Block Diagram of the SB-ADPCM encoder*

Transmit quadrature mirror filters

The input signal X_{in} is first filtered by two quadrature mirror filters (QMF) which split the frequency band [0, 8000 Hz] into two equal sub-bands. The outputs X_L and X_H of the lower and higher sub-bands are down-sampled at 8 kHz by the filtering procedure.

Lower sub-band ADPCM encoder

Figure 1.21 gives block diagram of the lower sub-band ADPCM encoder. To transmit the lower band, the encoder was designed to operate at 6, 5 or 4 bits per sample, corresponding to 48, 40 or 32 kbps, respectively. It is an embedded ADPCM with 4 core bits and 2 additional bits. The embedded property was introduced to prevent degradation in speech quality when the encoder and the decoder operate during short intervals in different modes.

Adaptive quantizer

A 60-level non-uniform adaptive quantizer is used to quantize the difference e_L between the input signal X_L and the estimated signal S_L. The output of the quantizer I_L is the ADPCM codeword for the lower sub-band.

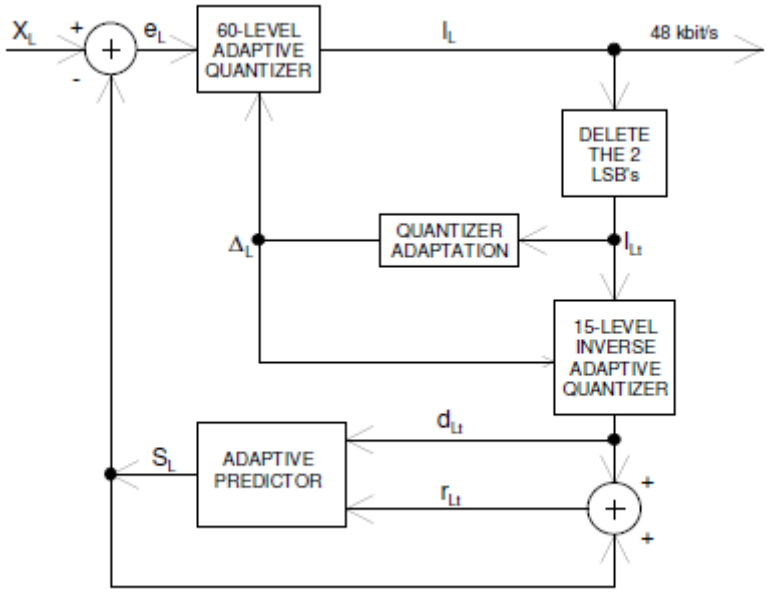


Figure 1.21: *Block Diagram of the lower sub-band ADPCM encoder*

Inverse adaptive quantizer

In the feedback loop the two least significant bits of I_L are deleted to produce a 4-bit signal I_{Lt} which is used for the adaptation of the quantizer scale factor and applied to a 15-level inverse adaptive quantizer to produce the quantized difference signal d_{Lt}.

Quantizer adaptation

In order to maintain a wide dynamic range and minimize complexity, the quantizer scale factor adaptation is performed in the base 2 logarithmic domain. The log-to- linear conversion is accomplished using a lookup table. There is no adaptation of the speed control parameter as in 32 kbps/s ADPCM because the encoder is designed to transmit more than voiceband data.

Adaptive predictor and reconstructed signal computation

The adaptive predictor structure is 2 poles and 6 zeroes. The two sets of coefficients (one for the poles and the other for the zeroes section) are updated using a simplified gradient algorithm. Stability constraints are applied to the poles in order to prevent possible unstable conditions. However, no predictor reset is applied for some specifics inputs conditions as it is

done in G.726 algorithm. The reconstructed signal r_{Lt} is computed by adding the quantized difference signal d_{Lt} to the signal estimate S_L produced by the adaptive predictor. The use of a 4-bit operation instead of a 6-bit operation in the feedback loops of the lower band ADPCM encoder and decoder allows for the insertion of data in the two least significant bits without causing mistracking in the decoder.

Higher sub-band ADPCM encoder

Figure 1.22 shows block diagram of the higher sub-band ADPCM encoder. This encoder is designed to operate at 2 bits per sample, corresponding to a fixed bitrate of 16 kbps. The encoder algorithm is very similar to the lower band one but with the following main differences. The quantizer is a 4-level non-linear adaptive quantizer. The higher sub-band ADPCM encoder is not embedded; hence the inverse quantizer uses the 2 bits in the feedback loop.

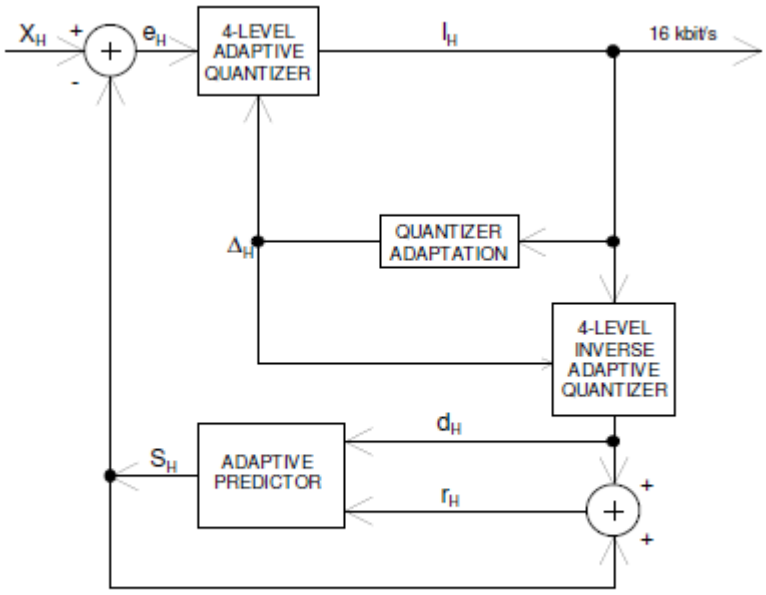


Figure 1.22: *Block Diagram of the higher sub-band ADPCM encoder*

Multiplexer

The resulting codewords from the higher and lower sub-bands I_H and I_L are combined to obtain the output codeword I with an octet format for transmission every 8 kHz frame

resulting in 64 kbps. Note that the 8 kHz clock may be provided by the network as it is always done for 64 kbps *α-law* or *μ-law* log-PCM (G.711) systems.

Figure 1.23 shows block diagram of the SB-ADPCM decoder.

Demultiplexer

The demultiplexer decomposes the received 64 kbps/s octet formatted signal I_r into two signals I_{Lr} and I_{Hr} which form the codeword inputs for the lower and higher sub-band ADPCM decoders, respectively.

Lower sub-band ADPCM decoder

Figure 1.24 shows a block diagram of the lower sub-band decoder. This decoder operates in three different modes depending on the received mode indication: 64, 56 and 48 kbps. The block which produces the estimate signal is identical to the feedback portion of the lower sub-band ADPCM encoder. The reconstructed signal r_L is produced by adding the signal estimate to the relevant quantized difference signals d_{L6}, d_{L5} or d_{L4}, which are selected according to the received indication of the mode of operation.

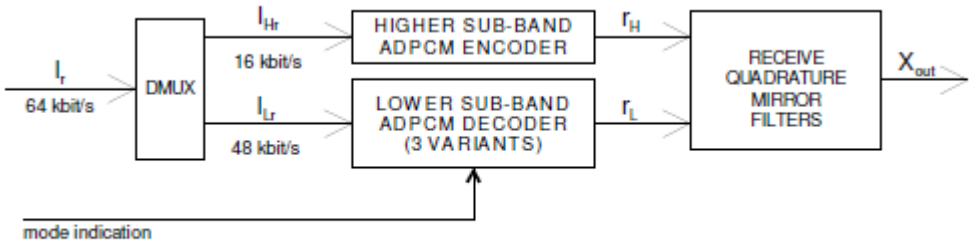


Figure 1.23: *Block Diagram SB-ADPCM decoder*

Higher sub-band ADPCM decoder

This decoder (see ***Figure 1.25***) is identical to the feedback portion of the higher sub-band ADPCM encoder. Here, the output is the reconstructed signal r_H.

Receive QMF

The receive QMF are two reconstruction filters which interpolate the outputs of the lower and higher sub-band ADPCM decoders from 8 to 16 kHz (r_H and r_L) and generate 16 kHz sampling reconstructed output X_{out}. Signal X_{out} is converted to analog by the digital to analog converter of the receiving side.

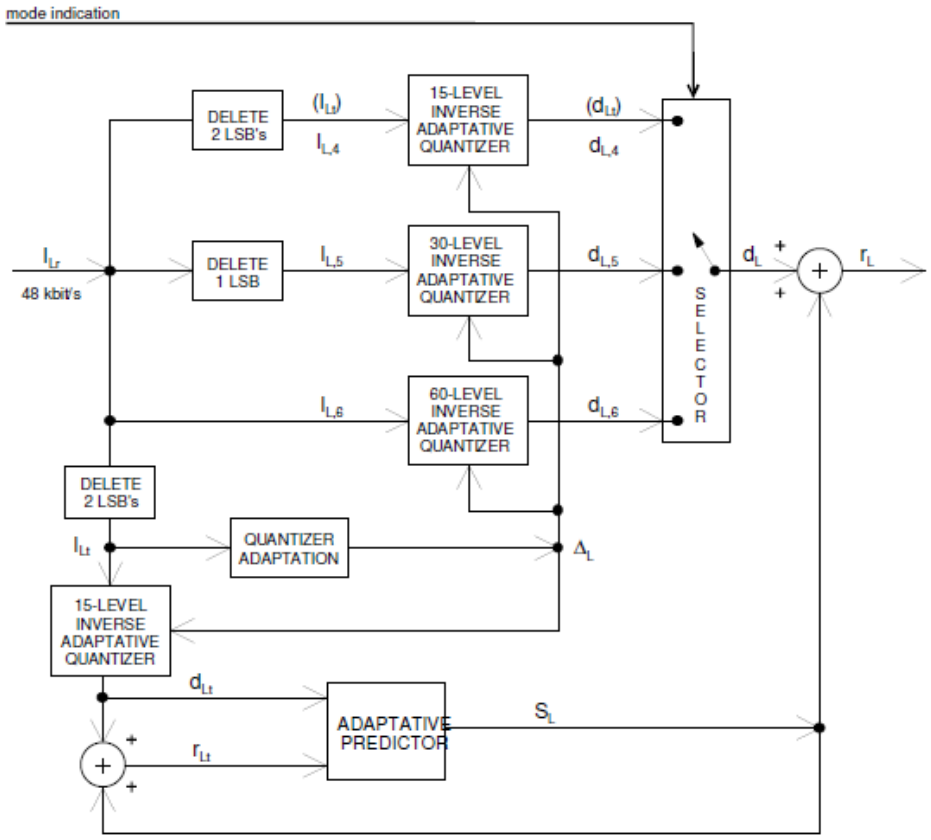


Figure 1.24: *Block Diagram of the lower sub-band ADPCM decoder*

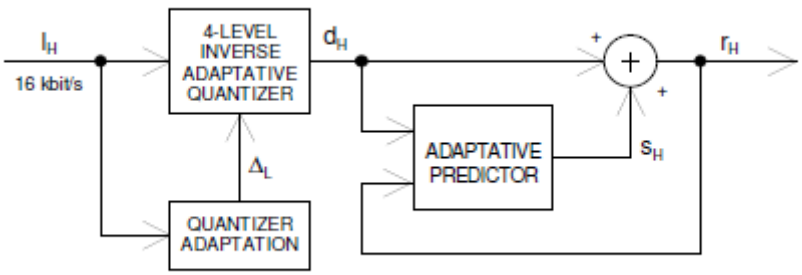


Figure 1.25: *Block Diagram of the higher sub-band ADPCM decoder*

1.4.4 G.723.1 (ACELP/MP-MLQ)

G.723.1 can be used for compressing the speech or other audio signal component of multimedia services at a very low bit rate. This codec has two bit rates associated with it: ***5.3*** and ***6.3 kbps***. The higher bit rate has better quality. The lower bit rate gives good quality and provides system designers with additional flexibility. Both rates are a mandatory part of the encoder and decoder. It is possible to switch between the two rates at any frame boundary.

This speech codec, unlikely with the previous three codecs that were described and they are **waveform codecs**, is a **hybrid codec**. As a result, it was optimized to represent speech with a high quality at the above rates using a limited amount of complexity. It encodes speech or other audio signals in frames, using linear predictive analysis-by-synthesis coding.

1.4.4.1 *Applications*

G.723.1 is also one of the main codecs of VoIP and, as a result it is present in the most IP phones. G.723.1 is a required audio codec in the H.324 ITU-T recommendation for *H.324* terminals offering audio communication and, also, in *3GPP 3G-324M* specification support for G.723.1 is not mandatory, but recommended. Finally, it can be used for transmission of recorded messages and it is used, like G.711 and G.726, in RoIP systems.

1.4.4.2 *Parameters*

This codec is designed to operate with a digital signal obtained by first performing telephone bandwidth filtering of the analogue input, then sampling at 8000 Hz and then converting to 16-bit linear PCM for the input to the encoder. The output of the decoder should be converted back to analogue by similar means.

In order to clarify how the different bitrates (5.3 and 6.3 kbps) of G.723.1 are produced the following have to be mentioned. The encoder transmits, every 30ms (because input speech is processed in frames of 30 ms), 189 bits or 158 bits in high-rate and low-rate mode, respectively. That makes bitrates of: $189 \times \frac{1000}{30} = 6300\ bps = 6.3\ kbps$ and $158 \times \frac{1000}{30} = 5267\ bps \cong 5.3\ kbps$, respectively.

1.4.4.3 *Algorithm*

G.723.1 is a hybrid speech codec and, as it was mentioned, parameters of the speech production model (parametric coding) are optimized in such a way that the decoded speech is as close as possible to the original waveform (waveform coding). As it is above-mentioned, in this thesis, complicated methods of signal processing are not about to be described in depth.

Initially, some things about ***CELP*** algorithm of hybrid coding have to be described, as G.723.1 is based in almost the same algorithm (**Adaptive CELP**).

Figure 1.26 shows how CELP algorithm approaches the human speech production model.

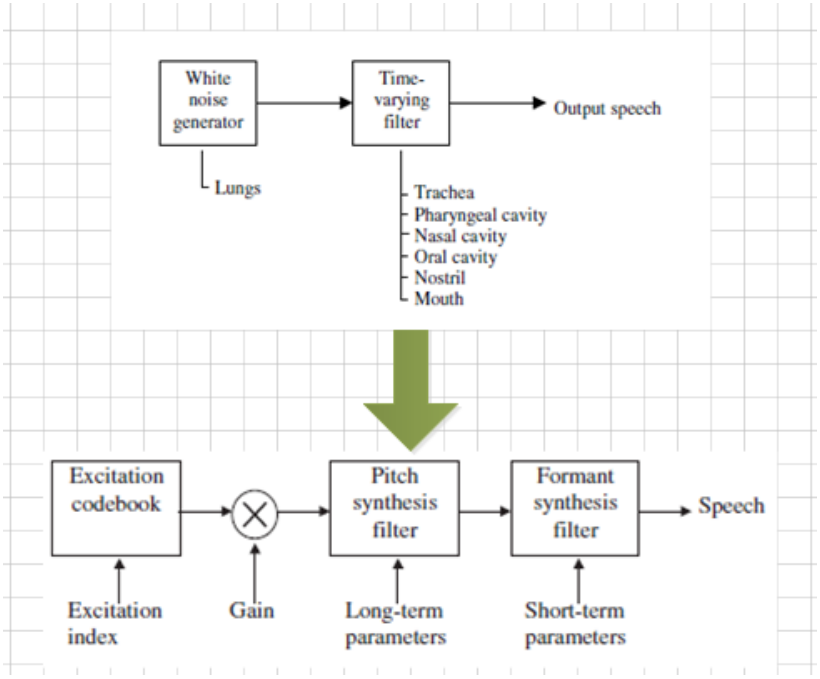


Figure 1.26: *The CELP model of speech production*

This algorithm is based on the principles of linear prediction **analysis-by-synthesis** coding (***Figure 1.27***) and attempts to minimize a **perceptually weighted error signal**.

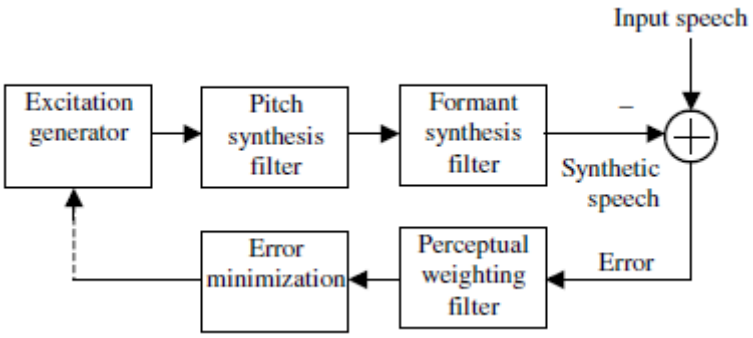


Figure 1.27: *Analysis-by-synthesis loop of a CELP encoder with perceptual weighting*

In order to understand the meaning of *analysis-by-synthesis* the following things have to be considered. In a hybrid speech codec, the speech signal is represented by a combination of parameters: **gain**, **filter coefficients**, **voicing strengths**, and so on. In an **open-loop** system, the parameters are extracted from the input signal, which are quantized and later used for synthesis.

A more effective method is to use the parameters to synthesize the signal during encoding and fine-tune them so as to generate the most accurate reconstruction.

Conceptually, this is a closed-loop optimization procedure, where the goal is to choose the best parameters so as to match as much as possible the synthetic speech with the original speech. Since the signal is synthesized during encoding for analysis purposes, the principle is known as *analysis-by-synthesis*.

General Description of G.723.1 Encoder

In G.723.1, the encoder operates on blocks (frames) of 240 samples each. That is equal to 30 ms at an 8-kHz sampling rate. Each block is first high pass filtered to remove the DC component and then divided into four subframes of 60 samples each. For every subframe, a 10th order Linear Prediction Coder (LPC) filter is computed using the unprocessed input signal. The LPC filter for the last subframe is quantized using a Predictive Split Vector Quantizer (PSVQ). The unquantized LPC coefficients are used to construct the short-term perceptual weighting filter, which is used to filter the entire frame and to obtain the perceptually weighted speech signal.

For every two subframes (120 samples), the open-loop pitch period, L_{OL}, is computed using the weighted speech signal. This pitch estimation is performed on blocks of 120 samples. The pitch period is searched in the range from 18 to 142 samples.

From this point the speech is processed on a 60 samples per subframe basis.

Using the estimated pitch period computed previously, a harmonic noise shaping filter is constructed. The combination of the LPC synthesis filter, the formant perceptual weighting filter, and the harmonic noise shaping filter is used to create an impulse response. The impulse response is then used for further computations.

Using the pitch period estimation, L_{OL}, and the impulse response, a closed-loop pitch predictor is computed. A fifth order pitch predictor is used. The pitch period is computed as a small differential value around the open-loop pitch estimate. The contribution of the pitch predictor is then subtracted from the initial target vector. Both the pitch period and the differential value are transmitted to the decoder.

Finally the non-periodic component of the excitation is approximated. For the high bit rate, Multipulse Maximum Likelihood Quantization (MP-MLQ) excitation is used, and for the low bit rate, an algebraic-code-excitation (ACELP) is used.

The block diagram of the encoder is shown in ***Figure 1.28***.

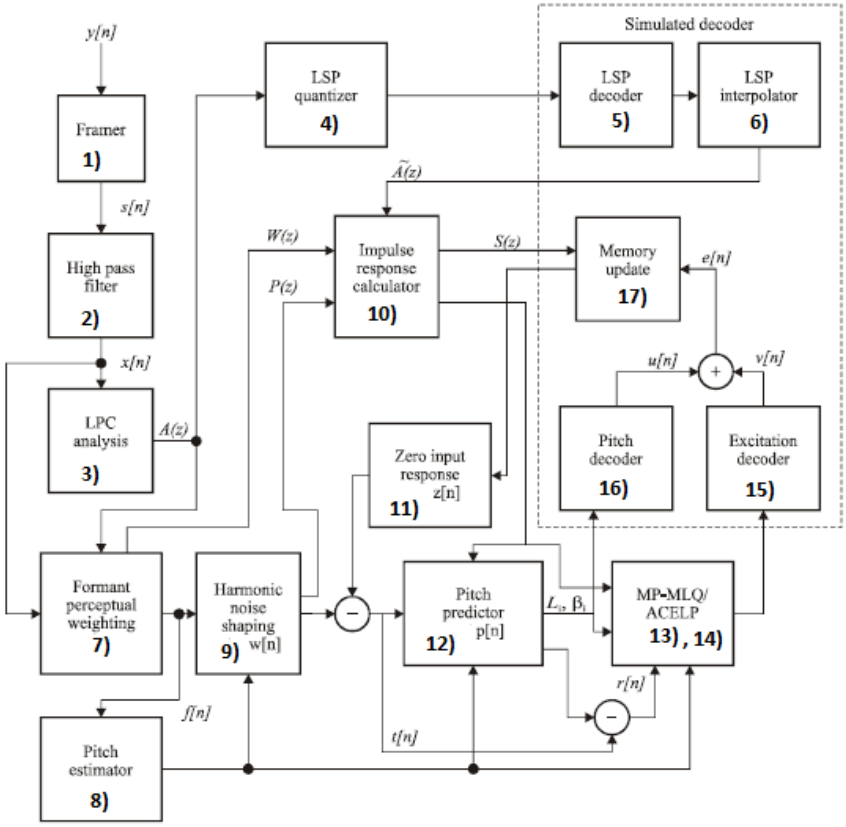


Figure 1.28: *Block diagram of G.723.1 encoder*

Framer

The coder processes the speech by buffering consecutive speech samples, ***y*[*n*]**, into frames of 240 samples, ***s*[*n*]**. Each frame is divided into two parts of 120 samples for pitch estimation computation. Each part is divided by two again, so that each frame is finally divided into four subframes of 60 samples each.

High pass filter

This block removes the DC element from the input speech, *s*[*n*]. The output of this filter is: $x[n]_{n = 0..239}$.

LPC analysis

The LPC analysis is performed on signal ***x*[*n*]** in the following way. Tenth order Linear Predictive (LP) analysis is used. For each subframe, a window of 180 samples is centred on the subframe. A **Hamming window** is applied to these samples. Eleven autocorrelation coefficients are computed from the windowed signal. A white noise correction factor of (1025/1024) is applied by using the formula *R*[0] = *R*[0](1 + 1/1024). The other 10

autocorrelation coefficients are multiplied by the binomial window coefficients table. The Linear Predictive Coefficients (LPC) are computed using the conventional **Levinson-Durbin recursion**. For every input frame, four LPC sets are computed, one for every subframe. These LPC sets are used to construct the short-term perceptual weighting filter.

LSP quantizer

First, a small additional bandwidth expansion (7.5 Hz) is performed. Then, the resulting $A_3(z)$ LP filter is quantized using a **predictive split vector quantizer**.

LSP decoder

The decoding of LSP coefficients is performed.

LSP interpolation

Linear interpolation is performed between the decoded LSP vector, $\widetilde{p}_n$, and the previous LSP vector, $\widetilde{p}_{n-1}$, for each subframe. Four interpolated LSP vectors, $\{ \widetilde{p}_i \}$, $i = 0..3$, are converted to LPC vectors, $\{ \widetilde{a}_i \}$ $i = 0..3$. The quantized LPC synthesis filter, $\widetilde{A}_i(z)$, is used for generating the decoded speech signal.

Formant perceptual weighting filter

For each subframe a formant perceptual weighting filter ($W_i(z)$) is constructed, using the unquantized LPC coefficients $\{a_{ij}\}$ $j = 1..10$. The input speech frame $\{x[n]\}$ $n = 0..239$ is then divided to four subframes and each subframe is filtered, using the above-constructed filter, and the weighted output speech signal $\{f[n]\}$ $n = 0..239$ is obtained.

Pitch estimation

Two pitch estimates are computed for every frame, one for the first two subframes and one for the last two. The open-loop pitch period estimate, L_{OL}, is computed using the perceptually weighted speech $f[n]$. A cross-correlation criterion, C_{OL}, maximization method is used to determine the pitch period.

From this point on, all the computational blocks are performed on a once per subframe basis.

Harmonic noise shaping

In order to improve the quality of the encoded speech, a harmonic noise shaping filter (***P*$_i$*(z)***) is constructed. After computing the harmonic noise filter coefficients, the formant perceptually weighted speech, ***f*[*n*]**, is filtered using ***P*(*z*)** to obtain the target vector, ***w*[*n*]**.

Impulse response calculator

For closed-loop analysis, the following combined filter, ***S*$_i$*(z)***, is used:

$$\boldsymbol{S_i}(\boldsymbol{z}) = \widetilde{\boldsymbol{A}}_{\boldsymbol{i}}(\boldsymbol{z}) \cdot \boldsymbol{W_i}(\boldsymbol{z}) \cdot \boldsymbol{P_i}(\boldsymbol{z}) .$$

Zero input response and ringing subtraction

The zero input response of the combined filter, ***S*$_i$*(z)***, is obtained by computing the output of that filter when the input signal is all zero-valued samples. The zero input response is denoted $\{z[n]\}_{n = 0..59}$. The ringing subtraction is performed by subtracting the zero input response from the harmonic weighted speech vector, $\{w[n]\}_{n = 0..59}$. The resulting vector is defined as ***t*[*n*] = *w*[*n*] – *z*[*n*].**

Pitch Predictor

The pitch prediction contribution is treated as a conventional adaptive codebook contribution. The pitch predictor is a fifth order pitch predictor. For subframes 0 and 2, the closed-loop pitch lag is selected from around the appropriate open-loop pitch lag in the range ± 1 and coded using 7 bits. (Note that the open-loop pitch lag is never transmitted.) For subframes 1 and 3 the closed-loop pitch lag is coded differentially using 2 bits and may differ from the previous subframe lag only by –1, 0, +1 or +2. The quantized and decoded pitch lag values will be referred to as ***L*$_i$**. The pitch predictor gains are vector quantized using two codebooks with 85 or 170 entries for the high bit rate and 170 entries for the low bit rate. The contribution of the pitch predictor, $\{p[n]\}_{n = 0..59}$, is subtracted from the target vector $\{t[n]\}_{n = 0..59}$, to obtain the residual signal $\{r[n]\}_{n = 0..59}$.

$$r[n] = t[n] - p[n]$$

High rate excitation (MP-MLQ)

The residual signal $\{r[n]\}_{n = 0..59}$, is transferred as a new target vector to the MP-MLQ block. This block performs the quantization of this vector. The quantization process is approximating the target vector ***r*[*n*]** by ***r'*[*n*]**. The purpose of this block is to estimate the unknown parameters of the algorithm that minimize the mean square of the error signal ***err[n]***. The parameters estimation and quantization processes are based on an analysis-by-synthesis method.

Low rate excitation (ACELP)

The same purpose with **13)** but it used in low rate mode and uses a different hybrid coding method. More information can be found on *ITU-T Rec. G.723.1 (05/2006)*.

Excitation Decoder

It decodes the parameters of the previous step and forms the excitation vector ***e[n]***.

Decoding of the pitch information

The decoding of pitch information is performed

Memory update

The last task of the *ith* subframe before proceeding to encode the next subframe is to update the memories of the synthesis filter $\widetilde{\boldsymbol{A}}_{\boldsymbol{i}}(\boldsymbol{z})$, the formant perceptual weighting filter $\boldsymbol{W}_{\boldsymbol{i}}(\boldsymbol{z})$, and the harmonic noise shaping filter $\boldsymbol{P}_{\boldsymbol{i}}(\boldsymbol{z})$. To accomplish this, the complete response of combined filter $\boldsymbol{S}_{\boldsymbol{i}}(\boldsymbol{z})$, is computed by passing the reconstructed excitation sequence through this filter. At the end of the excitation filtering, the memory of the combined filter is saved and will be used to compute the zero input response during the encoding of the next speech vector.

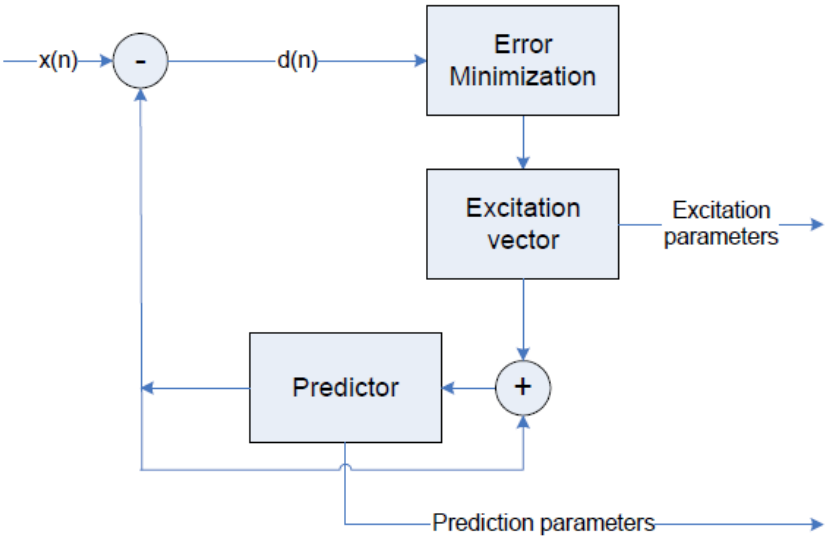


Figure 1.29: *A hybrid encoder*

General Description of G.723.1 Decoder

The decoder operation is also performed on a frame-by-frame basis. First the quantized LPC indices are decoded and then the decoder constructs the LPC synthesis filter. For every subframe, both the adaptive codebook excitation and fixed codebook excitation are decoded and input to the synthesis filter. The adaptive postfilter consists of a formant and a forward-backward pitch postfilter. The excitation signal is input to the pitch postfilter, which in turn is input to the synthesis filter whose output is input to the formant postfilter. A gain scaling unit maintains the energy at the input level of the formant postfilter.

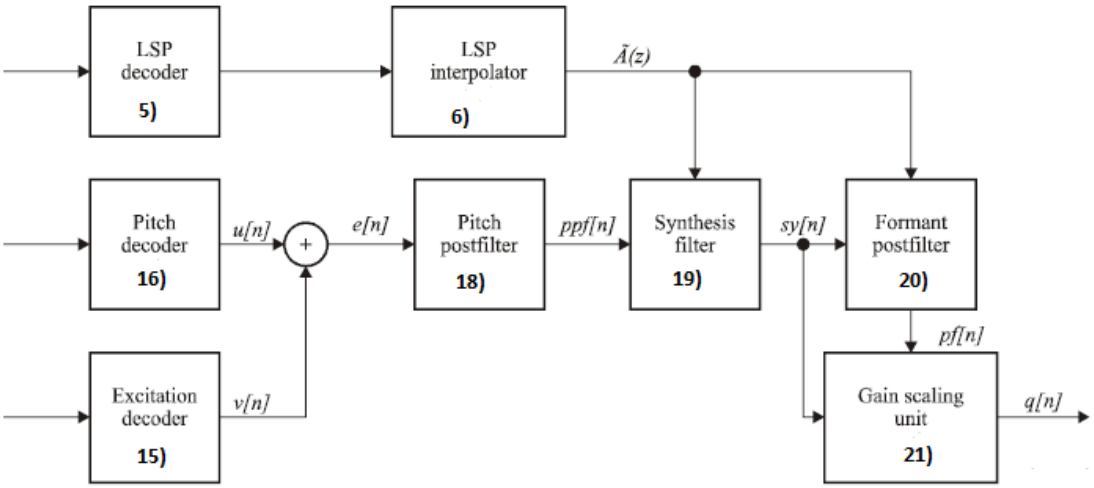


Figure 1.30: *Block diagram of G.723.1 decoder*

Pitch postfilter

A pitch postfilter is used to improve the quality of the synthesized signal. It is important to note that the pitch postfilter is performed for every subframe, and to implement it, it is required that the whole frame excitation signal $\{e[n]\}_{n=0..239}$ is generated and saved. The quality improvement is obtained by increasing the SNR at multiples of the pitch period.

LPC synthesis filter

The 10th order LPC synthesis filter $\widetilde{A}_i(z)$ is used to synthesize the speech signal ***sy[n]*** from the decoded pitch postfilter residual ***ppf[n]***.

Formant postfilter

A conventional ARMA postfilter is used. The postfiltered signal ***pf[n]*** is obtained as the output of the formant postfilter with input signal ***sy[n]***.

Gain scaling unit

This unit receives two input vectors, the synthesized speech vector $\{sy[n]\}_{n=0..59}$ and the postfiltered output vector $\{pf[n]\}_{n=0..59}$. Firstly, the amplitude ratio g_s is computed and, then, the output vector ***q[n]*** is obtained by scaling the postfiltered signal ***pf[n]*** . Finally, the gain g[n] is updated.

2 Selected Codecs Software Mapping

2.1 Introduction

After studying the selected speech codecs, the next step, is software implementation. The Fixed Point C-code is free and available from ITU (this does not happen if a speech codec is property of a company) and that was, basically, the source code that has been used.

In order to create a more user-friendly platform, the C-code was compiled and run inside the software environment of Matlab (by MathWorks, Inc). For this reason, the MEX-files tool of Matlab was used, which provides flexible usage of C-functions. Another reason for using Matlab was the tools provided for manipulating speech files and plotting their waveforms. As a result, a direct evaluation of the decoding result could be noticed.

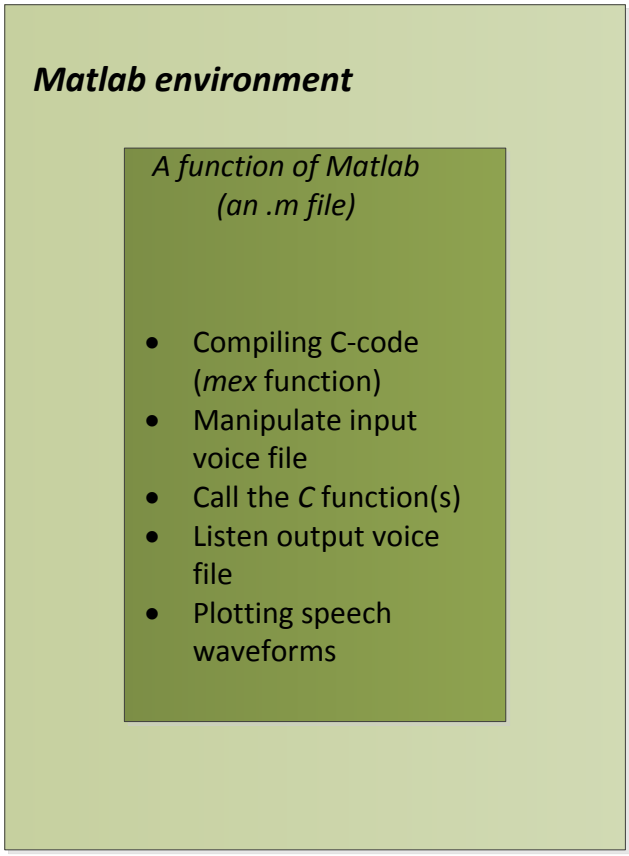


Figure 2.1: *The environment of speech codecs' software implementation*

The main purpose of this implementation was to create a platform (see ***Figure 2.1***) that it would be evolved to the Profiler of the speech codecs' arithmetic operations (this will be described in *Chapter 3*).

2.2 ITU Codes

For the software implementation of ***G.711***, ***G.726*** and ***G.722***, the Fixed Point C-code that it is included in ***Recommendation ITU-T G.191*** ("*Software tools for speech and audio coding standardization"*) was used and, for ***G.723.1*** the Fixed Point C-code from ***Recommendation ITU-T G.723.1***, respectively.

For each speech codec there is a **main** function that, basically, calls the two principal functions of software implementation: ***encode*** and ***decode***, taking care of the operations of the encoder and decoder, respectively. In both of them, usually, other functions are called in order to implement a specific part (block) of encoding or decoding.

Input speech is obtained (*fread* C-function) from an *input speech file*, which consists of 16-bit numbers (type *short* of C), and is processed from the function *encode*. The speech produced from encoding is written (*fprintf* C-function) to the *encoded speech file* or it is stored to a buffer. Then, function *decode* processes the encoded speech and writes the *output (decoded) speech* to the *output speech file* (again in a 16-bit form). The general flow chart of the whole process is shown in ***Figure 2.2***.

The *main* function for each speech codec, usually, provides the user with extra options through the values of some global variables, which are specified by the arguments defined on the executing command. Some of these options are:

- Perform encoding or decoding only
- Bit-Rate mode
- Choosing (for G.711 and G.726) to perform a- or μ-law
- Using extra features (e.g. Voice Activity Detection algorithm (VAD) that compresses transmitted bits during silent intervals)

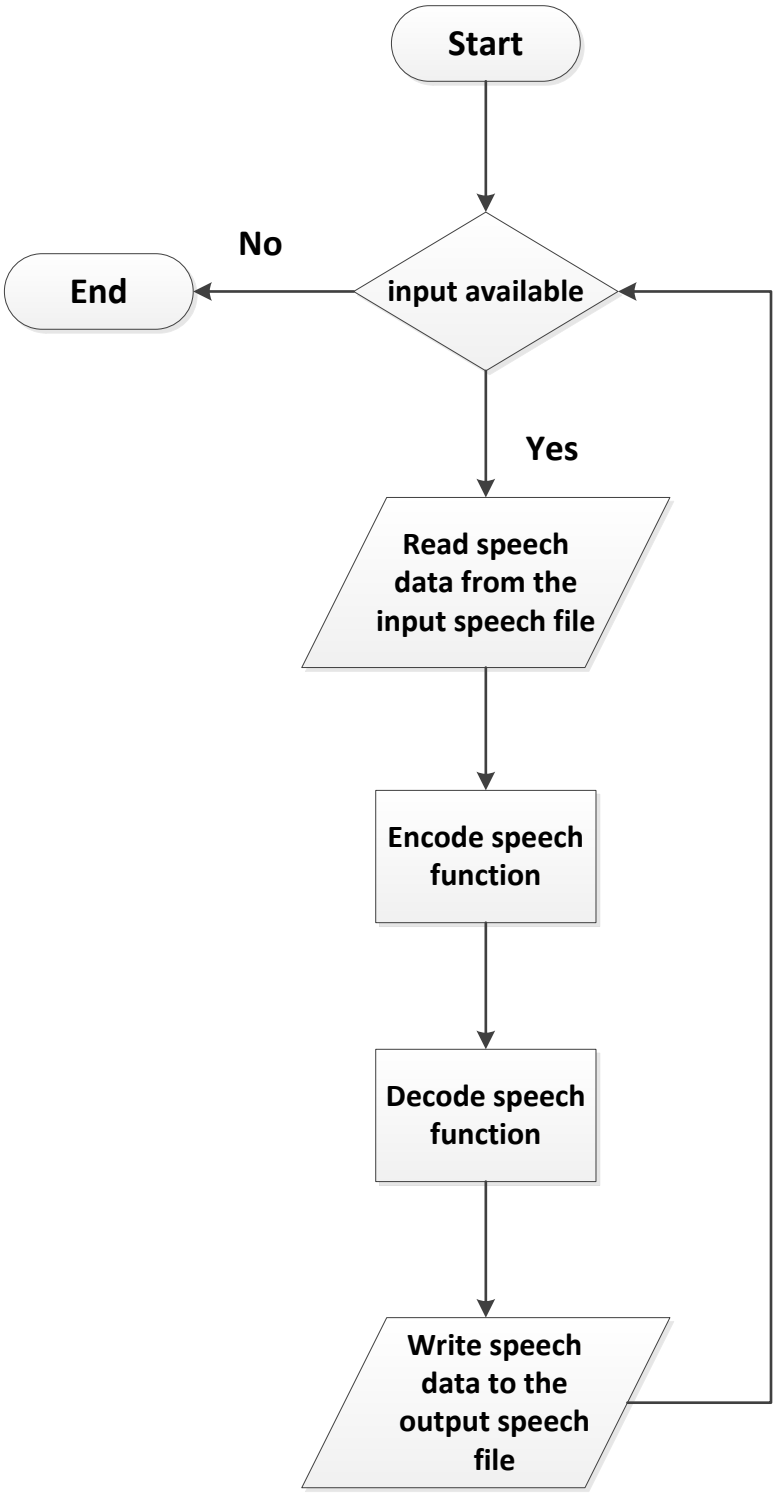


Figure 2.2: General *flow chart of the main function of speech codecs*

2.3 Matlab implementation

2.3.1 Software amendments

The software environment that it was created in Matlab, for selected codecs software mapping, can be divided to two parts:

1) The Matlab (.m) file
2) The *main* (mex) C-function

Actually, the second is included in the first, but they will be described separately in the following sections, in order to completely clarify the structure of the created platform.

2.3.1.1 *The Matlab (.m) file*

A matlab file was created for each speech codec, depending on the options that it is needed to be made. For example, the encoding law (a- or μ-law) is an option only in G.711 and G.726 and not to the other two speech codecs. ***Figure 2.3*** shows the general flow chart of the .m file and is followed by a short description of the numbered blocks.

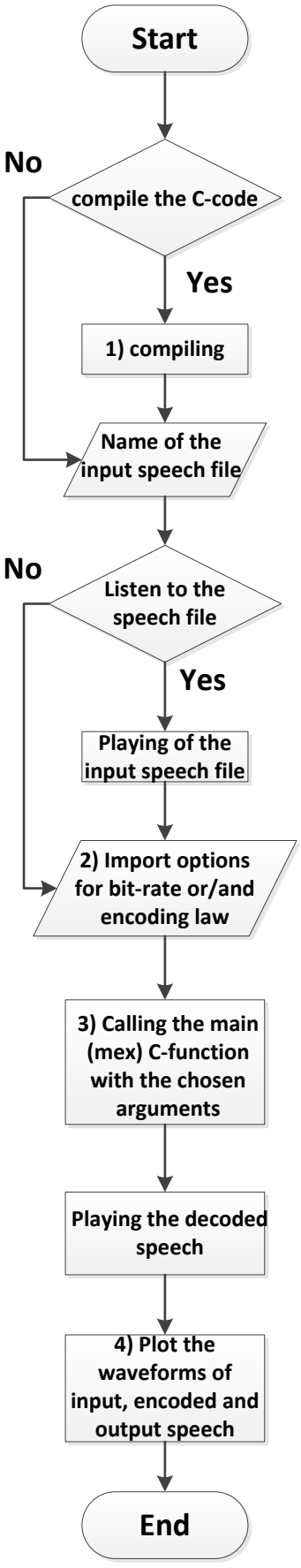


Figure 2.3: *Flow chart of the Matlab (.m) file*

1) Compiling

This is an available option of this platform that the user has in order to compile the C-code. C-compiling takes place very easily in Matlab with the use of *mex* function. With this tool, Matlab uses a C-compiler that it is available in the current computer system (*Microsoft Visual Studio 2010* was used for this thesis) and there is no need for complicated make-files, as it is usual in cases that parts of the code exists in different files.

The first file after the word *mex* is that in which the *main* function exists. The files that follow are them in which other functions of the C-code exist. An example (taken from the .m file of G.722) for the compiling command is:

mex *G722_profiled.c profiler.c g722.c basop32.c enh1632.c enh40.c control.c count. funcg722.c;*

2) Import options for bit-rate or/and encoding law

In this step user is urged to choose:

- *Bit-rate*
  I. 16, 24, 32 or 40 kbps for ***G.726***
  II. 48, 56 or 64 kbps for ***G.722***
  III. 5.3 or 6.3 kbps for ***G.723.1***
- *Encoding law*

  Available only in G.711 and G.726 and it is an option between *α*- or *μ-law*.

3) Calling the main *(mex)* C-function with the chosen arguments

In this step the C-code (encoding-decoding) is executed just by using the name of the file which includes the *main (mex)* function. Also, the arguments that are needed should be included. An example of this calling taken from the aforementioned case is:

G722 (input_file, bit_rate);

4) Plot the waveforms of input, encoded and output (decoded) speech

In this final step the waveforms of Input, Encoded and Output speech are plotted. Figure from ***Figure 2.4*** to ***Figure 2.7*** depict these waveforms for each

speech codec. Finally, ***Figure 2.8*** shows the messages in the command window of Matlab, while is executing G.711 encoding.

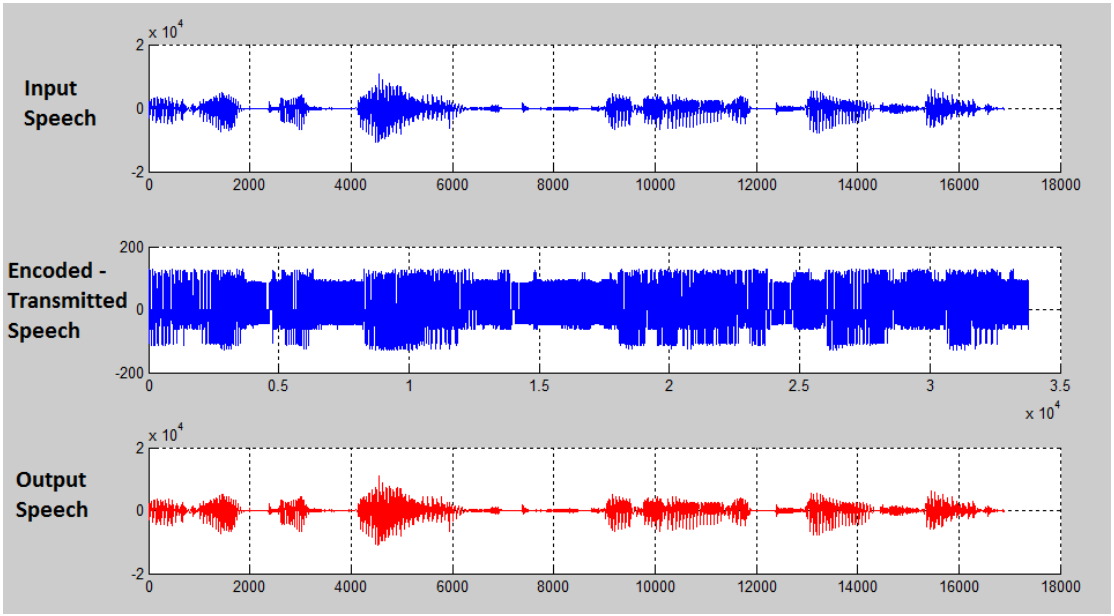


Figure 2.4: *Speech waveforms of G.711 (a-law)*

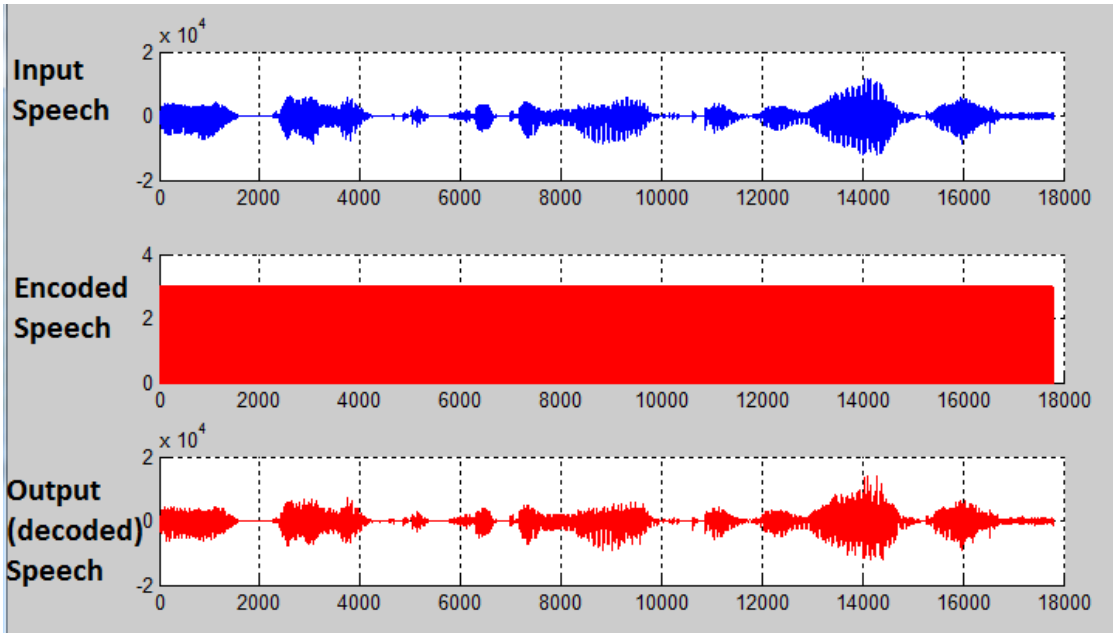


Figure 2.5: *Speech waveforms of G.726 (u-law, 16kbps)*

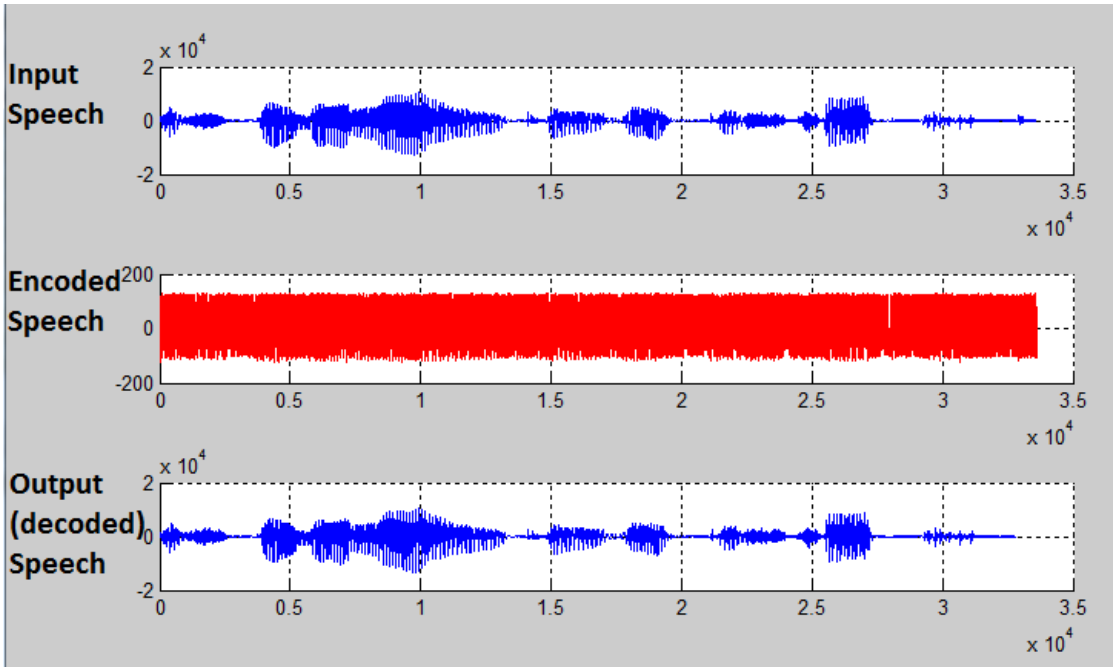


Figure 2.6: *Speech waveforms of G.722 (64kbps)*

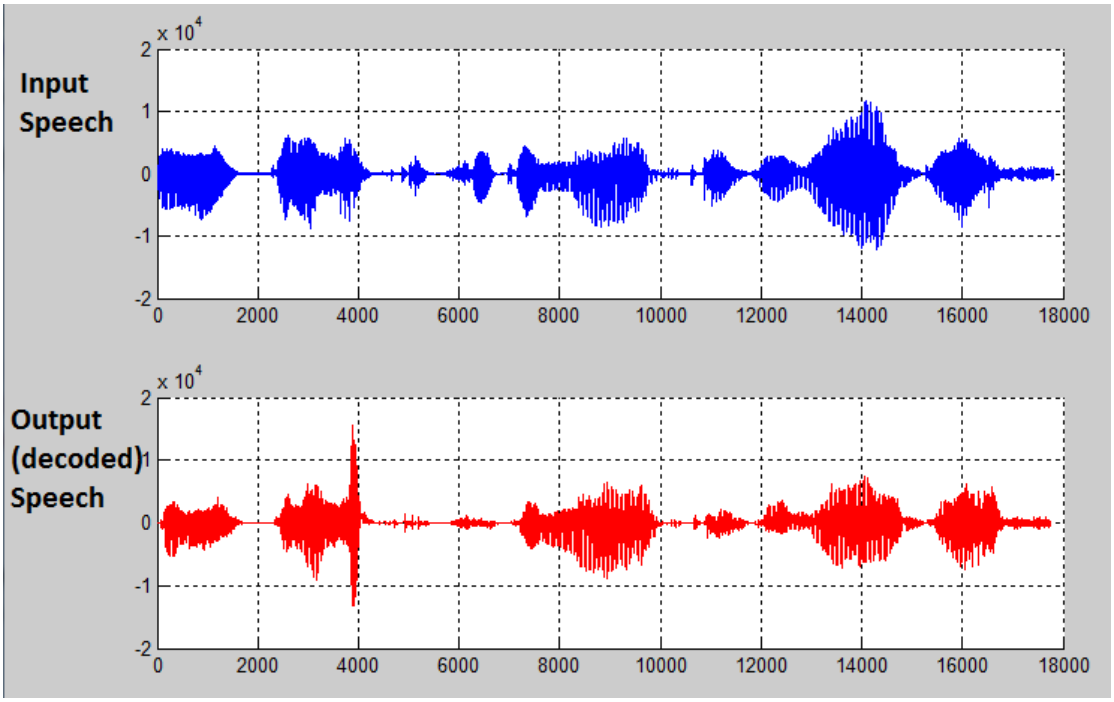


Figure 2.7: *Speech waveforms of G.723.1 (5.3 kbps). Some distortion can be noticed*

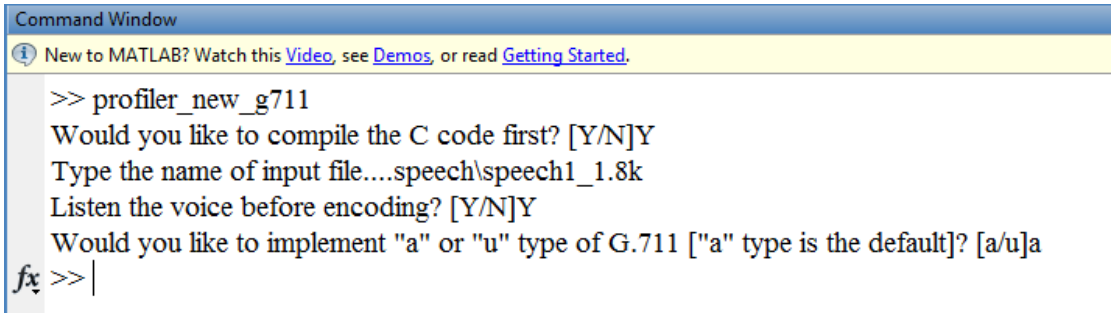


Figure 2.8: *Command window of Matlab while executing G.711 encoding*

2.3.1.2 *The main (mex) C-function*

The main (*mex*) function is, actually, the same main function that the aforementioned ITU Recommendations provide. The only amendments that took place were:

- Omitting some of the options that ITU codes provide (e.g. performing only encoding or only decoding), and keeping only the options concerning bit-rate and encoding law.
- Amendments concerning the transform from pure C-code to *mex* C-code (C-code that can be called from a Matlab function). These are:
    1. Instead of the usual syntax *void* ***main*** *()*, it has to be written:

       void ***mexFunction****(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ])*. The list of arguments lets the Matlab code, that calls the main function, to pass arguments, as it is mentioned that happened in **3)** of *2.3.1.1* . For this case, Matlab supports a general type of data called *mxArray* in order to pass arguments. An array (prhs) of arguments (mxArrays) is constructed. For example at the function call: **G.722 (input_file, bit-rate)**, *input_file* is prhs[0] and *bit-rate* is prhs[1].

       In order to use the arguments as a specific type, inside the *mexFunction*, there are special functions. For example function *mxArrayToString* that makes a String from an mxArray is used in the following way:

       mode = mxArrayToString(prhs[1]);

, where *mode* is declared as a string (char *mode) inside the *mexFunction*.

2. In mex C-code, some functions have different names (e.g. *mexPrintf* instead of *printf*), but they have the usual syntax and functionality.

2.3.2 Experimental parameters

In order to test the software platform that was created, eight different human voices were used as samples. Four of them were female voices and four of them were male. For each voice there were two samples of an, approximately, two-seconds duration each. The used samples were characterized by a *8kHz* sampling frequency as it is needed for the three of the selected codecs (G.711, G.726, G.723.1). For G.722, which is a wideband speech codec, the same voices were used, but characterized by a *16kHz* sampling frequency. The following figures (***Figure 2.9*** to ***Figure 2.24***) depict the waveforms (for the both samples of each person) in the domains of time and frequency (F_S=*8kHz*) for the used samples of human speech. For the waveforms of frequency domain, a 50 msec window of speech was used for each person, supposing that the signal is linear and time-invariant in that piece of time and, as a result, a DFT can be performed. The computations took place in Matlab.

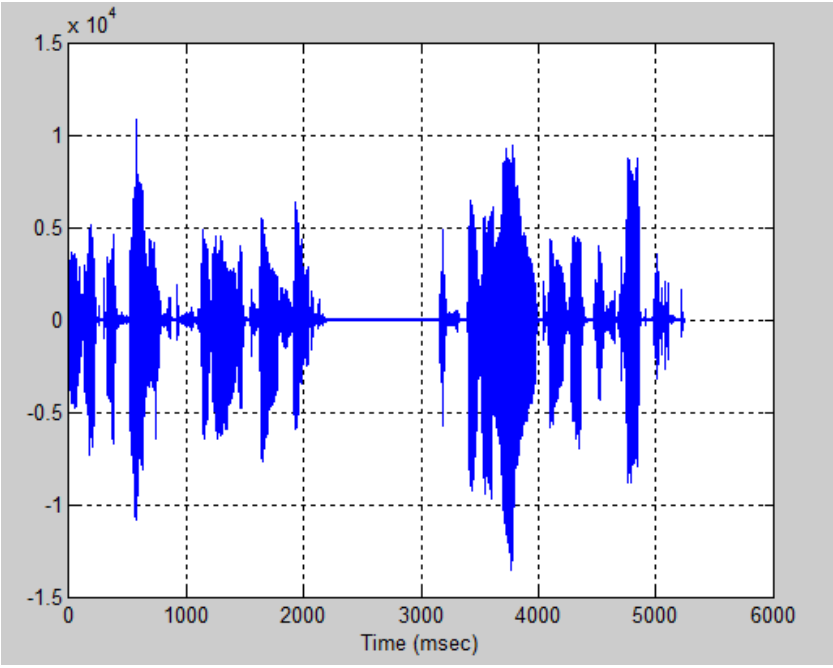


Figure 2.9: *Time domain waveform for the first male speech sample*

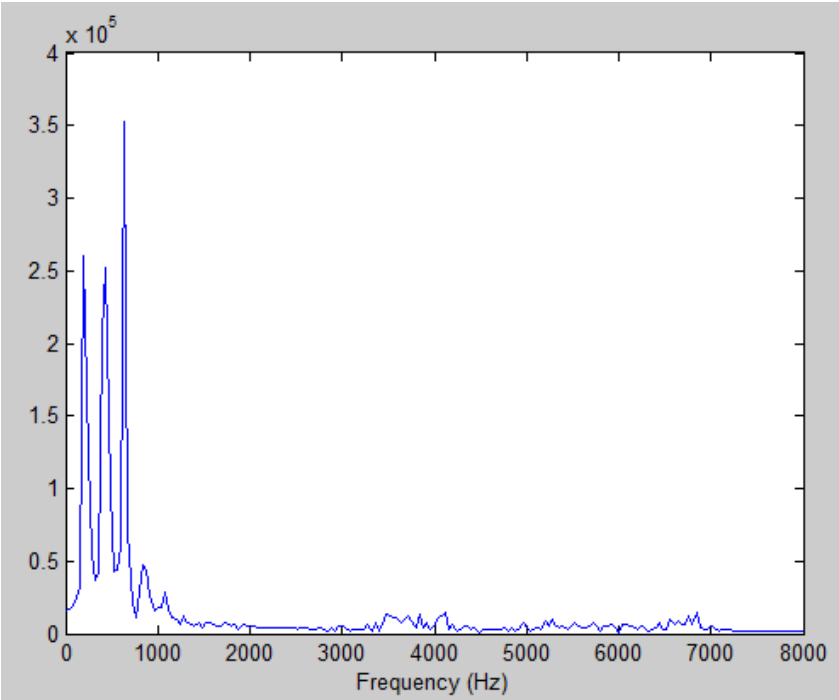


Figure 2.10: *Frequency domain waveform for the first male speech sample*

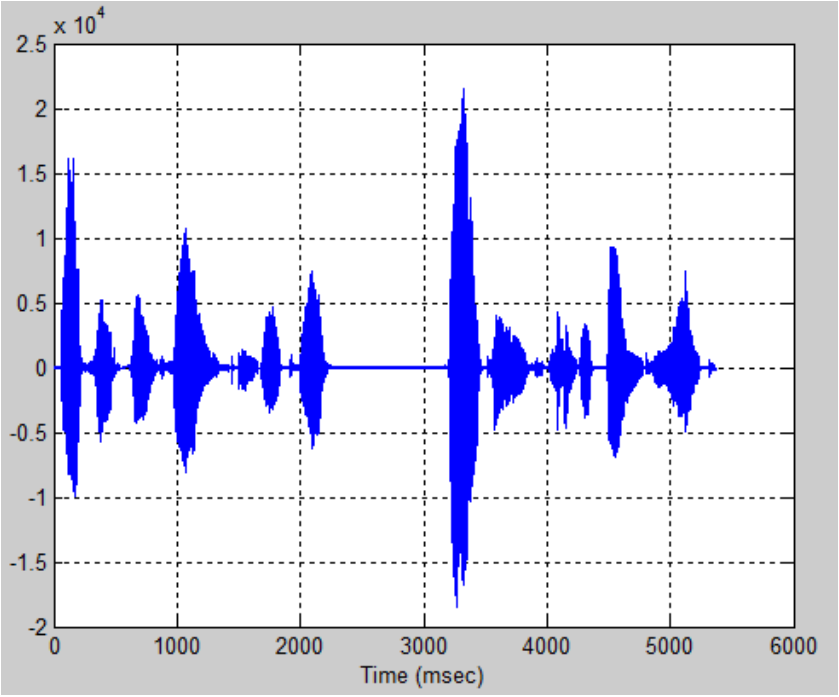


Figure 2.11: *Time domain waveform for the second male speech sample*

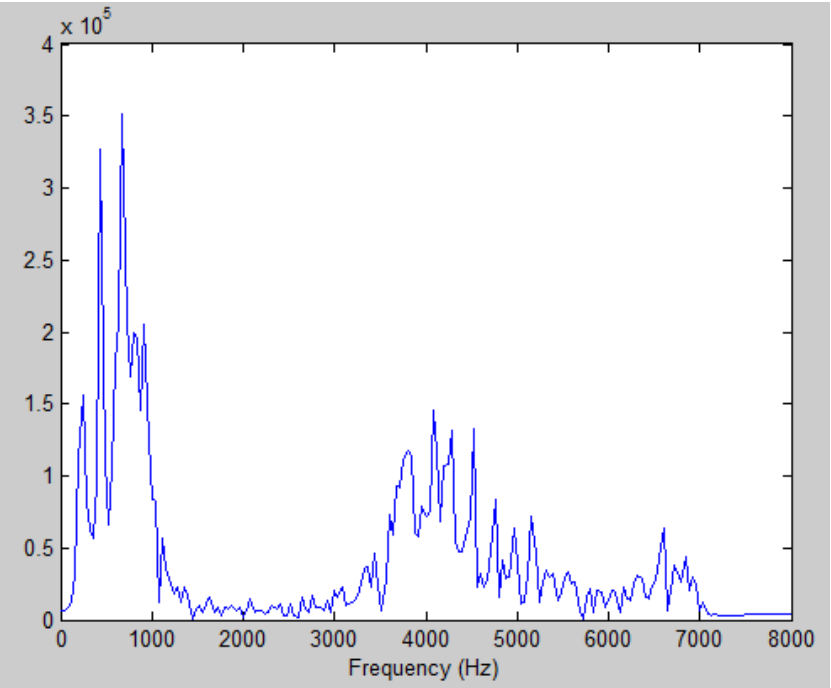


Figure 2.12: *Frequency domain waveform for the second male speech sample*

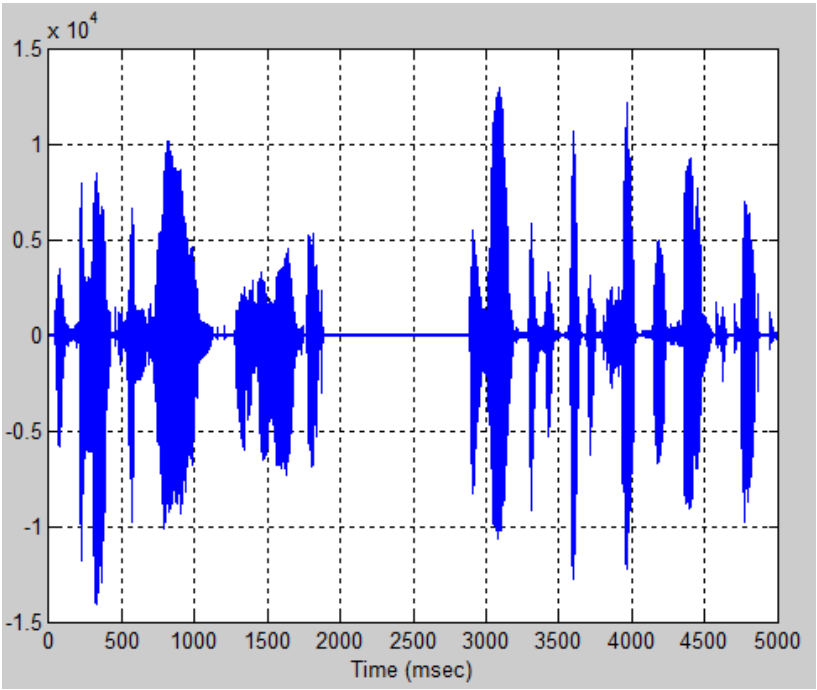


Figure 2.13: *Time domain waveform for the third male speech sample*

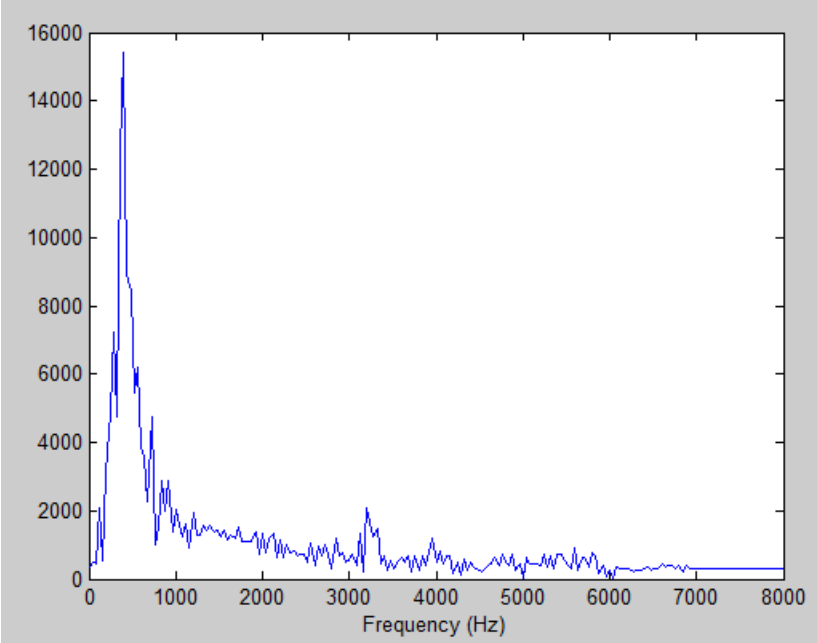


Figure 2.14: *Frequency domain waveform for the third male speech sample*

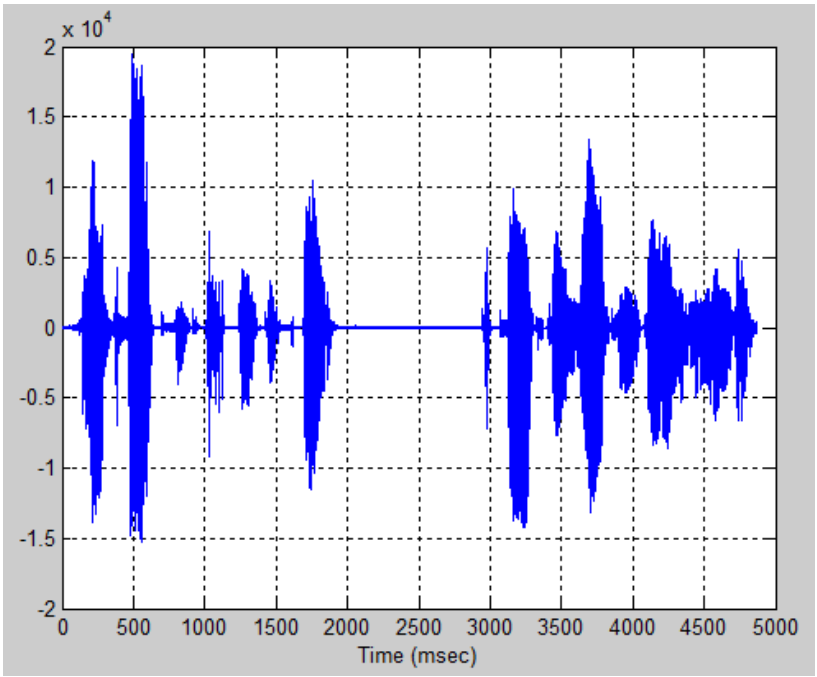


Figure 2.15: *Time domain waveform for the fourth male speech sample*

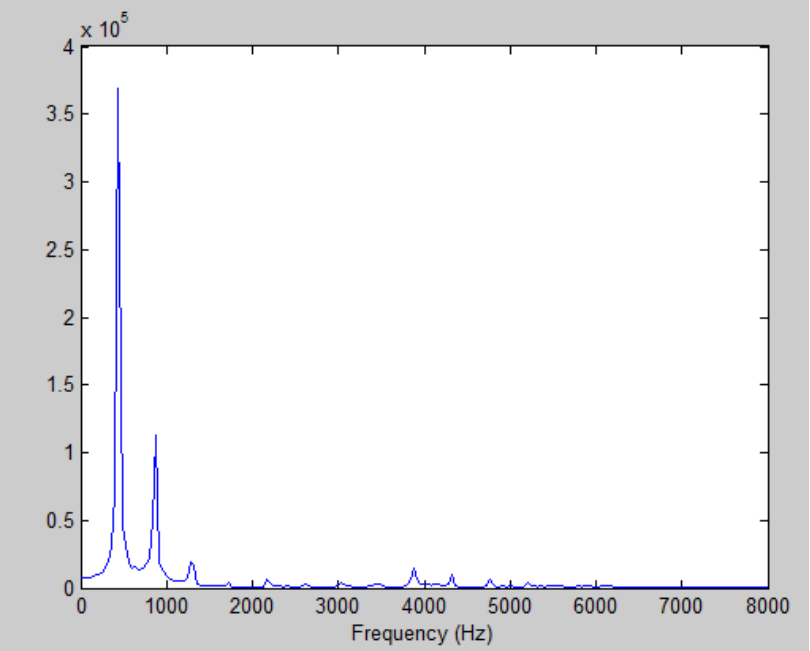


Figure 2.16: *Frequency domain waveform for the fourth male speech sample*

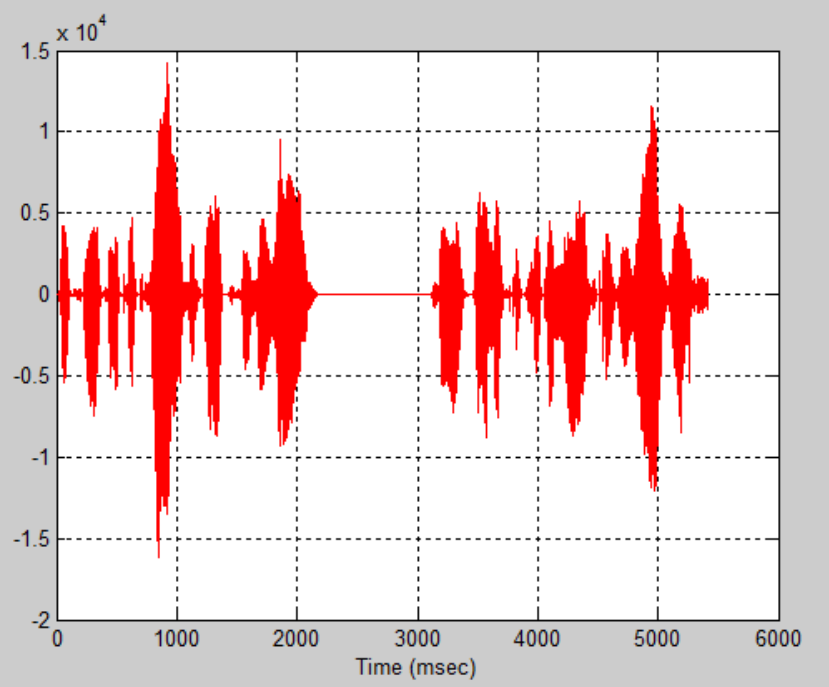


Figure 2.17: *Time domain waveform for the first female speech sample*

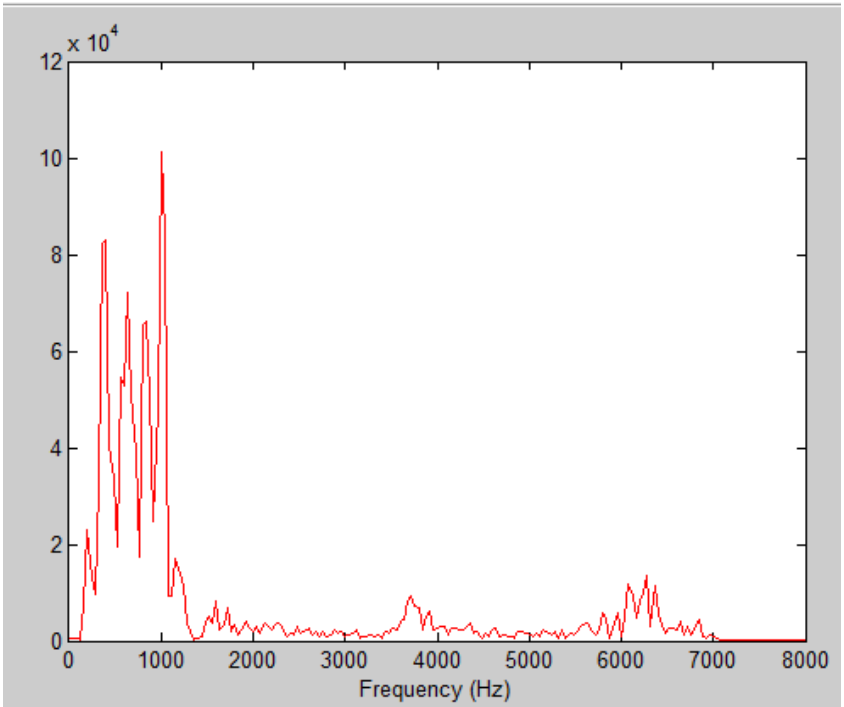


Figure 2.18: *Frequency domain waveform for the first female speech sample*

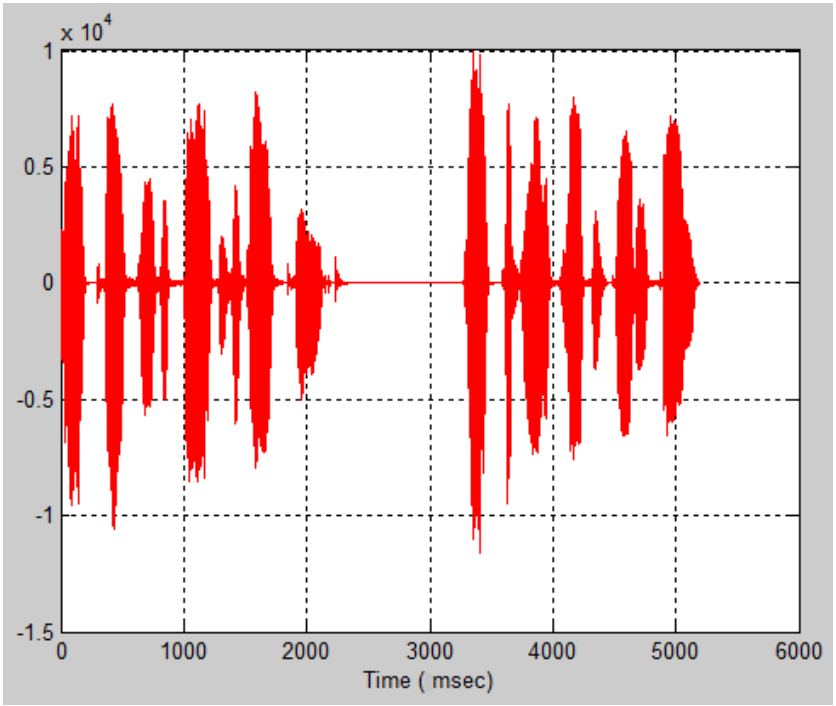


Figure 2.19: *Time domain waveform for the second female speech sample*

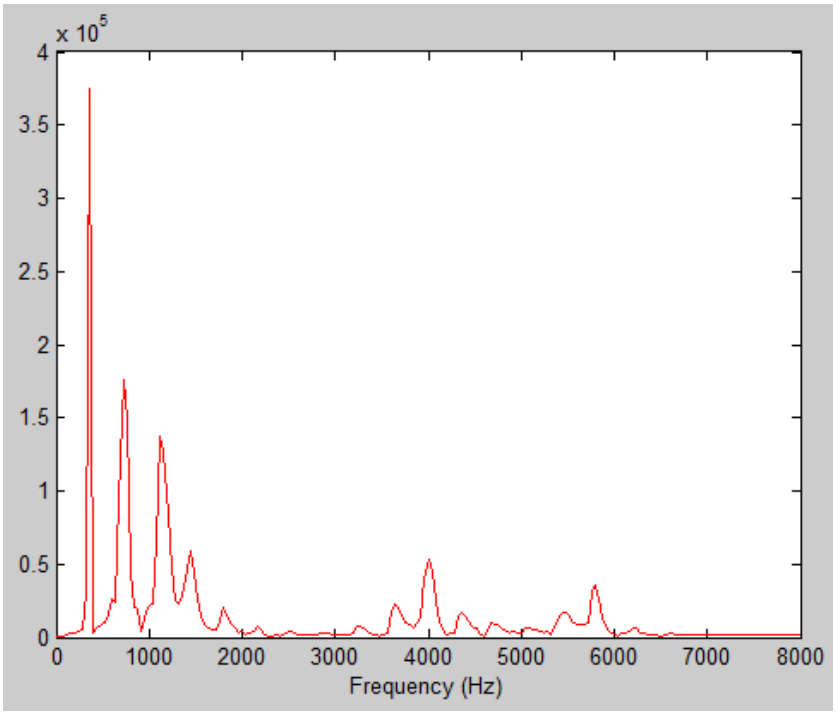


Figure 2.20: *Time domain waveform for the second female speech sample*

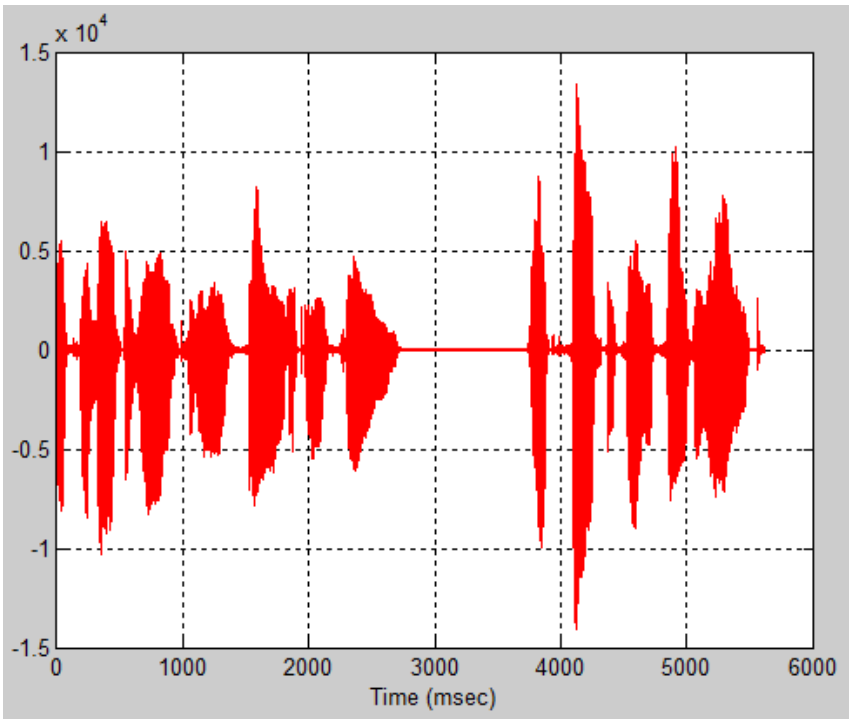


Figure 2.21: *Time domain waveform for the third female speech sample*

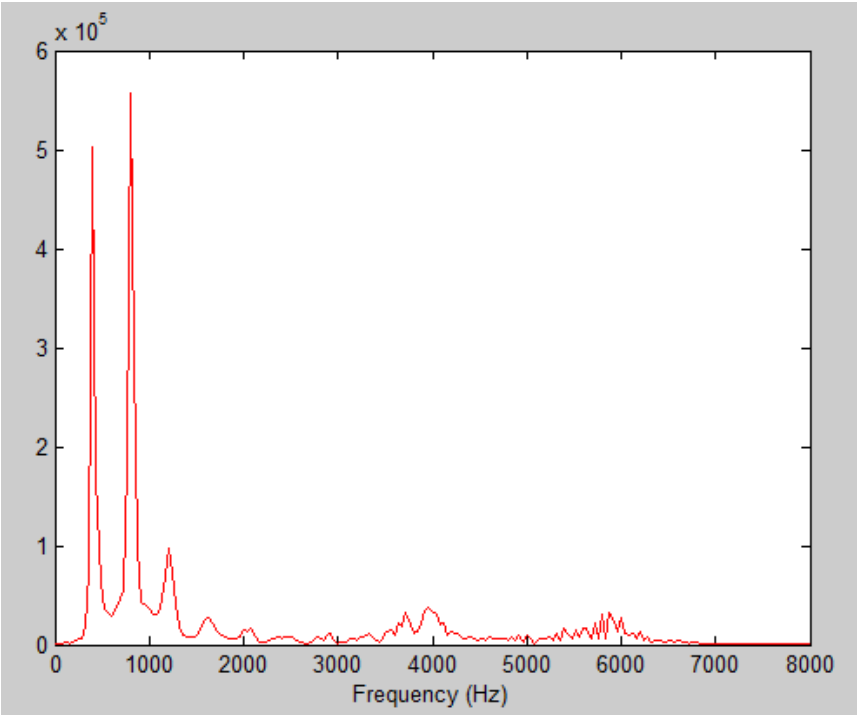


Figure 2.22: *Frequency domain waveform for the third female speech sample*

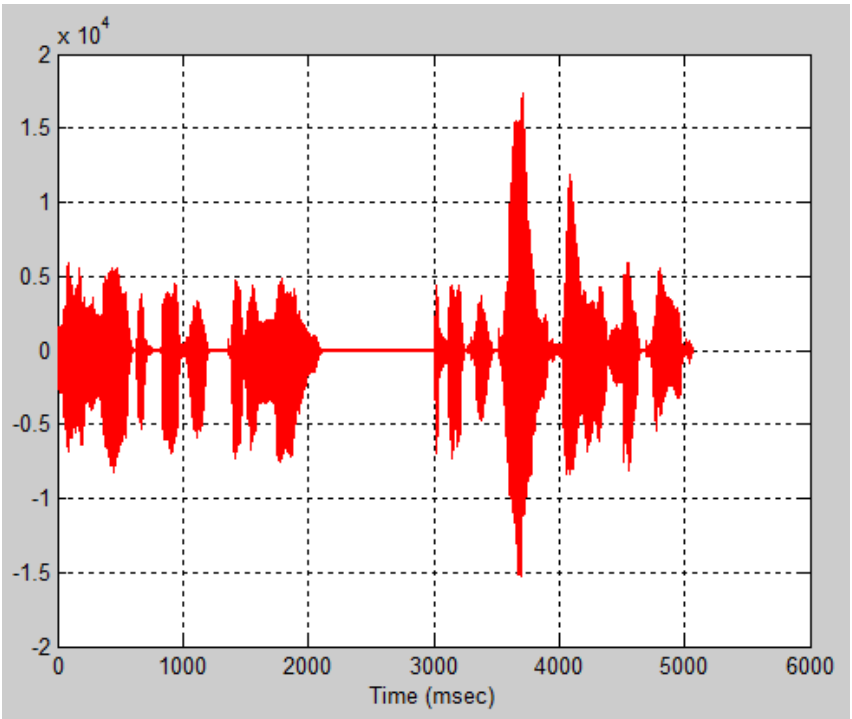


Figure 2.23: *Time domain waveform for the fourth female speech sample*

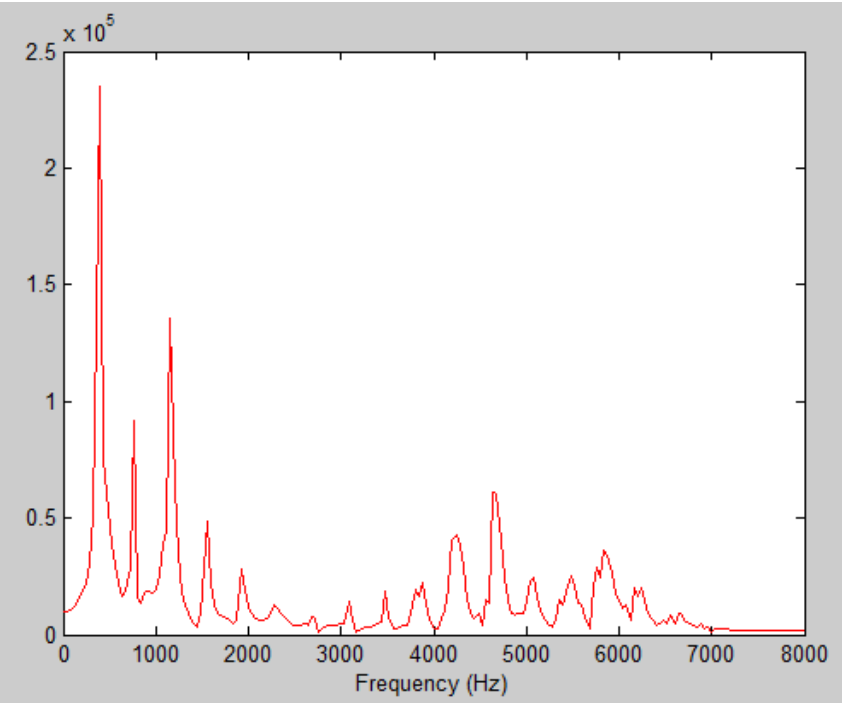


Figure 2.24: *Frequency domain waveform for the fourth female speech sample*

The conclusions from testing the selected speech codecs, through the software platform, were:

- The different quality of the output (decoded) speech among different codecs. Obviously, the bit-rate plays a very important role on that. For example, G.726 has better results on its normal mode (32kbps) than on the mode of 16kbps *or* G.723.1 has much poorer result than the other codecs.
- Wideband speech codec, G.722, has the best of the decoding results.
- G.711 and G.726 have almost the same quality of decoding speech.
- G.723.1 seems to have better results for female speech.

3 Speech Codecs Profiling

3.1 Introduction

After the platform, for the software mapping of speech codecs, was created, the next, and very principal, stage of this thesis was profiling of the arithmetic operations that are performed during the executing of the selected speech codecs algorithms. In the following sections of this chapter, the different steps and approaches that were held, in order to reach to the conclusion of which arithmetic unit should have been implemented, are described.

The general approach was to create a C-library named *profiler.c* in which profiling functions were about to be developed. These functions were called during the executing, and in specific points, from the main C-code in order to perform a task like changing the value of a counter or printing a message to a .txt file. The input speech samples, in order to evaluate the profiling result, were those referred in paragraph *2.3.2*.

Firstly, the whole procedure was developed for *G.711* and *G.726*, because they are not so complicated as *G.722* and *G.723.1*. After the method of profiling reached to a final form, it was applied to the other two codecs as well.

3.2 Functions and arithmetic operations

The first approach was characterized by an effort to keep track of the path that data (input voice) followed. For this reason the functions that were developed, in this step, had two purposes:

1. tracking of the function calls that take place during execution and of the arguments that get involved in these function calls
2. tracking of the arithmetic operations that take place during execution (this was a first step towards arithmetic profiling)

For the second purpose a data structure was created named ***operation_element*** in order to store information for each arithmetic operation. From the elements of this structure, a list named ***operations*** was constructed. ***Figure 3.1*** shows the declaration of this structure.

```
/*structure dedicated to an operand (argument), named "args" and having two fields:
 1) name of the operand
 2) bits of the operand
*/
struct arg {
     char name[12]; /*name of the argument*/
     short bits;/* (1) for 16 bits, (2) for 32 bits, (3) for 40 bits*/
};
typedef struct arg
        args;

/*structure dedicated to an arithmetic operation,
named "operation_element" and having six fields:
1) No_C (explained below)
2) No_C (explained below)
3) symbol, which operation takes place
4) operands, an array of two "args", the two operands of the arithmetic operation
5) result, an element of "args" that stores the information about the result of the operation
6) A pointer to the next "operation_element" in order to create a list
with all the operations that take place during executing
*/

struct Array2 {
     long No_C; /*Number of calling (in which function is this operation performed)*/
     long No_op; /*Number of operation (in the order of operations)*/
     char symbol;
     args operands[2]; /*Arguments*/
     args result;
     struct Array2 *next;
};
typedef struct Array2
        operation_element;
```

Figure 3.1: *Declaration of the structure* ***operation_element***

It has to be mentioned that through the whole profiling procedure (and not only during this step) **logical operations (like and, or, xor etc), complements and shifts were *not* taken into consideration**. For example, operations like $(a \ll 2) + b$ or $(a \,\&2) + b$ were considered the same as $a + b$. This approach was due to the fact that the purpose of this thesis was to implement an efficient arithmetic unit and as a result no attention was given to other kinds of operations.

Table 3.1 shows the characteristics of the developed functions. The first and the second function are dedicated to purpose 1 (track function calls), while the other four functions to purpose 2 (track arithmetic operations).

Figure 3.2 and **Figure 3.3**, that follow, show two snapshots, of .txt file *Function_Calls* after executing of G.711 encoding-decoding and of the and *Arith_operations* after executing of G.726 encoding-decoding, respectively. *Function_Calsl.txt* and *Arith_operations.txt* are described below in *Table 3.1*.

Also, ***Figure 3.4*** provides an overview of the way that this part of profiling was performed.

Function Name	Input Parameters	Output Parameters	Function Description
void **keep_track()**	*char* ***fCalled**, *int* **No1**	Writes on **Function_calls.txt**	When there is a function call, it prints on *Function_calls.txt* the number of calling (*No1)* followed by the name of the Function that is called
void **f_arguments()**	*char* ***name**, *char* ***nature**	Writes on **Function_calls.txt**	After any *keep_track()*, it is called *n*-times (where *n* is the number of arguments) and it prints on *Function_calls.txt* the name and nature (type) of each argument
void **track_oper()**	*char* **symbol**, *long* **No_C**, *long* **No_op**	Writes on **Arith_operations.txt**	It is called when an arithmetic operation is performed and it prints on *Arith_operations.txt*: 1. the number of Function in which the operation is performed (*No_C*) 2. the number that the operation has in the general operation order (*No_op*) 3. the *symbol* of the operation
void **update_operations_type()**	*char* **symbol**, *long* **No_C**, *long* **No_op**	Creates and updates a new *operation_element* of the list *operations*	It is called inside the function *track_oper()* and creates a new element of the list *operations*. It fills the fields of *symbol*, *No_C* and *No_op*
void **f_operands()**	*short* **num**, *short* **result**, *char* ***name** ,*short* **bits**	Writes on **Arith_operations.txt**	After any *track_oper()*, it is called *3* times (2 for the operands and one for the result) and it prints on *Arith_operations.txt* the name and the number of bits of each operand (or of the result)
void **update_operations_operands()**	*short* **num**, *short* **result**, *char* ***name** ,*short* **bits**	updates the *operation_element* that the previous *update_operations_type()* created	It is called inside the function *f_operands()* and it fills the fields of *name* and *bits* for the first operand (when *num* = 0), the second operand (when *num* = 1), the result (when *result* = 1)

Table 3.1: *Describing of the profiling functions*

```
1. main
2.alaw_compress,    Arguments of alaw_compress: frame   (int),  inp_buf (array of -frame- 16-bit),  tmp_buf (array of -frame- 16-bit),
3. Return to main
4.alaw_expand,  Arguments of alaw_expand:   frame   (int),  tmp_buf (array of -frame- 8-bit),   out_buf (array of -frame- 8-bit),
5. Return to main
```

Figure 3.2: *Function_calls.txt after G.711 encoding-decoding*

```
Calling:35608,  Operation:3233651,  type:+, Operands:   srnexp  (32),   anexp   (32),   Result: wanexp  (32)
Calling:35608,  Operation:3233652,  type:*, Operands:   srnmant (32),   anmant  (32),   Result: temp1   (32)
Calling:35608,  Operation:3233653,  type:+, Operands:   temp1   (32),   48  (16),   Result: wanmant (32)
Calling:35608,  Operation:3233654,  type:-, Operands:   26  (16),   wanexp  (32),   Result: wanmag  (32)
Calling:35608,  Operation:3233655,  type:-, Operands:   65536   (32),    wanmag (32),   Result: wan (32)
Calling:35608,  Operation:3233656,  type:-, Operands:   16384   (16),   an  (32),   Result: anmag   (32)
Calling:35608,  Operation:3233657,  type:+, Operands:   srnexp  (32),   anexp   (32),   Result: wanexp  (32)
Calling:35608,  Operation:3233658,  type:*, Operands:   srnmant (32),   anmant  (32),   Result: temp1   (32)
Calling:35608,  Operation:3233659,  type:+, Operands:   temp1   (32),   48  (16),   Result: wanmant (32)
Calling:35608,  Operation:3233660,  type:-, Operands:   26  (16),   wanexp  (32),   Result: wanmag  (32)
Calling:35608,  Operation:3233661,  type:+, Operands:   srnexp  (32),   anexp   (32),   Result: wanexp  (32)
Calling:35608,  Operation:3233662,  type:*, Operands:   srnmant (32),   anmant  (32),   Result: temp1   (32)
Calling:35608,  Operation:3233663,  type:+, Operands:   temp1   (32),   48  (16),   Result: wanmant (32)
Calling:35608,  Operation:3233664,  type:-, Operands:   26  (16),   wanexp  (32),   Result: wanmag  (32)
Calling:35608,  Operation:3233665,  type:-, Operands:   65536   (32),    wanmag (32),   Result: wan (32)
Calling:35608,  Operation:3233666,  type:-, Operands:   16384   (16),   an  (32),   Result: anmag   (32)
Calling:35608,  Operation:3233667,  type:+, Operands:   srnexp  (32),   anexp   (32),   Result: wanexp  (32)
Calling:35608,  Operation:3233668,  type:*, Operands:   srnmant (32),   anmant  (32),   Result: temp1   (32)
Calling:35608,  Operation:3233669,  type:+, Operands:   temp1   (32),   48  (16),   Result: wanmant (32)
Calling:35608,  Operation:3233670,  type:-, Operands:   26  (16),   wanexp  (32),   Result: wanmag  (32)
Calling:35608,  Operation:3233671,  type:-, Operands:   65536   (32),    wanmag (32),   Result: wan (32)
Calling:35608,  Operation:3233672,  type:+, Operands:   wb11    (32),   wb21    (32),   Result: temp1   (32)
Calling:35608,  Operation:3233673,  type:+, Operands:   temp1   (32),   wb31    (32),   Result: temp2   (32)
Calling:35608,  Operation:3233674,  type:+, Operands:   temp2   (32),   wb41    (32),   Result: temp3   (32)
```

Figure 3.3: *Snapshot from Arith_operations.txt after G.726 encoding-decoding*

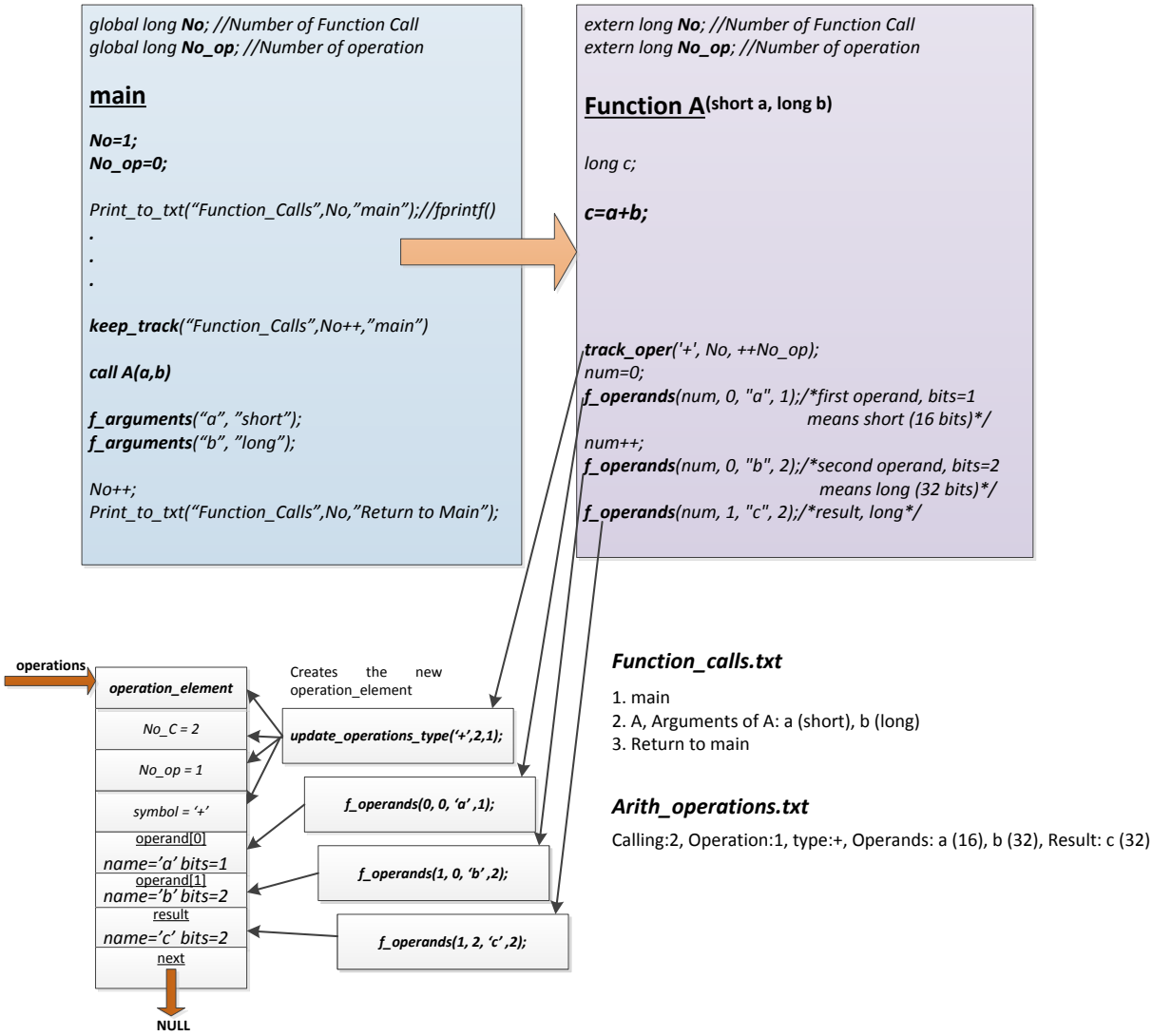


Figure 3.4: *Overview of performing the profiling functions*

This version of the aforementioned profiling functions is not exactly the form they had at their very first development. For example, within the framework of *track_sequence()* and *f_arguments()*, a list of *function_elements* was created while executing. But, then, this feature was removed as it did not provide any useful information and *Function_calls.txt* was enough.

The conclusion, from this profiling attempt, was the need to focus on the *operations* list and on the data that could be extracted from it and *not* to concentrate anymore on the procedure of tracking the function calls.

3.3 Sequences of arithmetic operations

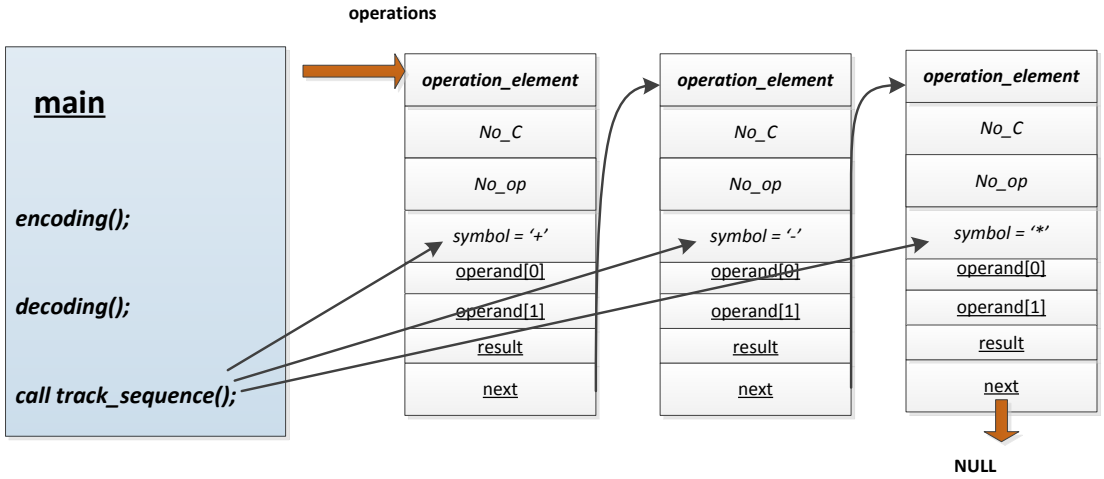
As a result of the aforementioned conclusion, the next step of profiling was developing a new function named *track_sequence()*. Its characteristics are described in ***Table 3.2*** and ***Figure 3.5***, ***Figure 3.6*** clarify its operation.

Function Name	Input Parameters	Output Parameters	Function Description
void **track_sequence()**	Processes the list *operations*	Writes on **sequence.txt**	It is called at the end of the *main* function and it accomplishes the following tasks: • It accesses the field *symbol* of all the nodes of the list *operations* • It counts sequential multiplications and additions or subtractions • It makes blocks of the counted operations • It prints the blocks on *sequence.txt*

Table 3.2: *Describing of the **track_sequence()** profiling function*

```
37690:  add or sub  28014-add   9676-sub
1:  mult
3:  add or sub  2-add   1-sub
1:  mult
3:  add or sub  2-add   1-sub
1:  mult
3:  add or sub  2-add   1-sub
1:  mult
3:  add or sub  2-add   1-sub
1:  mult
3:  add or sub  2-add   1-sub
1:  mult
3:  add or sub  2-add   1-sub
1:  mult
3:  add or sub  2-add   1-sub
1:  mult
3:  add or sub  2-add   1-sub
1:  mult
13: add or sub  10-add  3-sub
1:  mult
7:  add or sub  4-add   3-sub
1:  mult
74: add or sub  48-add  26-sub
1:  mult
5:  add or sub  2-add   3-sub
1:  mult
4:  add or sub  2-add   2-sub
1:  mult
```

Figure 3.5: *Snapshot from sequence.txt after G.726 encoding-decoding*



sequence.txt

2: add or sub 1-add 1-sub
1: mult

Figure 3.6: *An overview of performing track_sequence() for three sequential arithmetic operations*

This new profiling function (*track_sequence()*) provided a clearer view of the operations that are performed during the encoding-decoding procedure. But, obviously, this approach was not efficient enough, as data dependencies information for these operations was needed.

3.4 Data dependencies of arithmetic operations

In order to extract the required information of data dependencies among the performed arithmetic operations, a new approach was developed without any further process of the list *operations*. This approach consisted of the two following steps:

- Tracking of the loops performed while executing
- Creating a list of operation blocks

3.4.1 Tracking of loops

Within the framework of this profiling step, the C-codes of G.711 and G.726 were searched for loops. The purpose of this searching was to discover loops that are performed for a remarkable number of iterations and they include a sequence of operations with data dependencies. This could, possibly, lead to a compound operation, worthy to be mapped on hardware.

As a result, a simple function named *cnt_loops()* was developed and added to the profiling platform. Its description is available in ***Table 3.3*** that follows and ***Figure 3.7***, ***Figure 3.8*** clarify its operation.

Function Name	Input Parameters	Output Parameters	Function Description
void **cnt_loops()**	*long* **cnt**, *char* ***tree**	Writes on **num_of_iterations.txt**	It is called inside any algorithmic loop. It prints on *num_of_iterations.txt* the number of iterations (*cnt*) that the loop was performed followed by the operations of the loop. The number of bits that the operands are composed of is, also, noted.

Table 3.3: *Describing of the **cnt_loops()** profiling function*

```
[# of iterations] <operand1> arithmetic <operand2>....


[1] iexp(16)+1
[2] iexp(16)+1
[2] iexp(16)+1
[2] iexp(16)+1
[2] iexp(16)+1
[2] iexp(16)+1
[3] iexp(16)+1
[3] iexp(16)+1
[3] iexp(16)+1
[3] iexp(16)+1
[3] iexp(16)+1
[3] iexp(16)+1
```

Figure 3.7: *Snapshot from num_of_operations.txt after G.726 encoding-decoding*

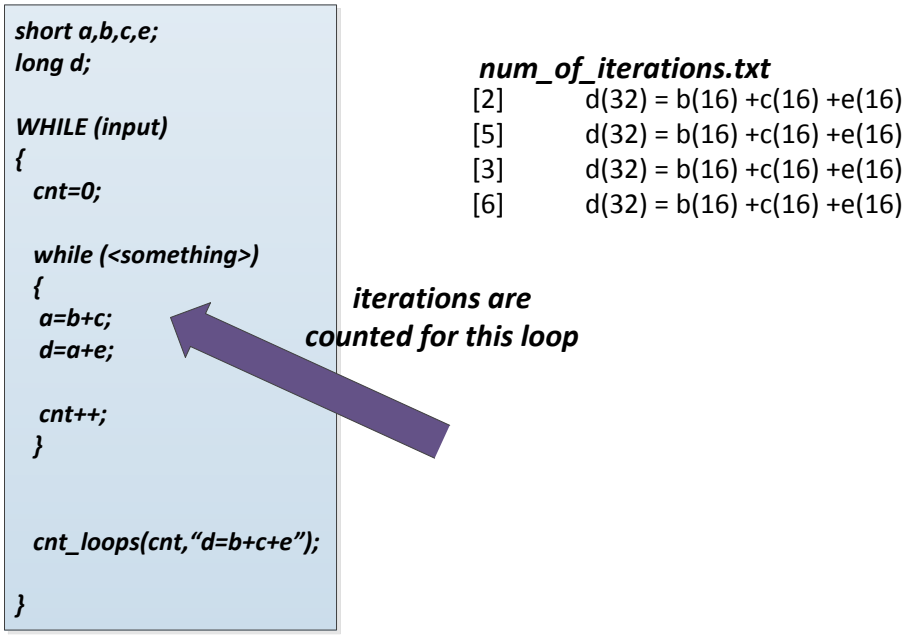


Figure 3.8: *An overview of performing **cnt_loops()***

The drawback of using *cnt_loops* was, initially, the fact that there are not a lot of algorithmic loops in the C-code (at least for G.711 and G.726 that were the basis of profiler's developing) and as a result, most of the arithmetic operations were not represented. But, the main problem was the importance of discovering data loops and not only algorithmic loops. There

was a need for finding remarkable arithmetic patterns that can be detected to a lot of points of the encoding-decoding procedure.

3.4.2 Listing of the operation blocks

The aforementioned conclusion led to a thorough listing of the operation blocks. The whole code was scanned and a .txt file named ***operation_blocks*** was created. This file included every operation that takes place during the encoding-decoding procedure and, also, it kept the information for the number of bits that compose every operand. Obviously, the data dependencies could be detected and a clearer view was provided.

For this profiling approach there were not any new profiling functions developed. The whole code (for each speech codec) was scanned and there was only use of the *fprintf()* C-function. When, in the code, an *if-then-else* structure was detected, counters were placed in any different branch of the if-structure; in order to count how many times each different operation block was executed. Also, extra information is provided like the name of the function in which a particular operation block is executed or the fact that a variable participates in an operation after a *shifting* or a *xor*. For example, an operation like $= b + \hat{}c$, means that variable *c* participates in the addition after it has been *shift*-ed or *xor*-ed.

This profiling method was the first that it was also applied to the two other selected codecs (G.722, G.723.1). ***Figure 3.9*** to ***Figure 3.12*** show a small part of the ***operation_blocks.txt*** file for each of the speech codecs.

```
        ENCODING_u_law....

[16901] MACRO1

        END OF ENCODING_u_law

input: linbuf(16), output: logbuf(16)

MACRO1:
    temp1=linbuf+33
    temp2=0x0008-[max times=6](accum: segno(| linbuf)+1)
    temp3=0x000F - low_nibble(temp1,segno)
    logbuf= ^temp2 | temp3

    (3+max)adds


        DECODING_u_law....

[16901] MACRO2

        END OF DECODING_u_law

input: logbuf(16), output: linbuf(16)

    MACRO2:
    temp1=^logbuf+1
    temp2=constant(temp1)
    linbuf =temp2*logbuf+constant(logbuf)+^temp2-132

    1add,1mult,3adds
```

Figure 3.9: Part of *operation_blocks.txt* for *G.711 (μ_law)*

```
G726_accum(inputs[all 16]: wa1, wa2, wb1, wb2, wb3, wb4, wb5, wb6[all 16])
G726_accum(outputs:[all 16]: se, sez)
    sez=wb11+wb21+ wb31+wb41+wb51+wb61
    se=sez+wa21+wa11

    7adds

G726_expand(inputs: s(16), law, output: sl(16)):
    [4876]  temp1=^(s-128-^s)+constant(| s)   -or-
    [12024]  temp1=[^(s-128-^s)+33]*constant(s)-33+constant(| s)

    sl = 16384-temp1

    [4876] 4adds -or-  [12024] 3adds,1mult,3adds

G726_subta(inputs: sl(16), se(16),outputs: d(16)):
    d = (sl+49152)+65536-(se+32768)

    4adds

G726_mix(inputs: al(16), yu(16), yl(32),output: y(16)):
    temp1=yu+16384-^yl
    temp2=(16384 - temp1)*al
    y = (16384-temp2)+^yl

    3adds,1mul,2adds

G726_log(input: d(16), outputs: dl(16), ds(16)):
    temp1=65536-d
    ds=^d
    dl=^temp1+constant(| temp1)
```

if-structure **counters were used to measure how many times each operation block was executed**

Figure 3.10: Part of *operation_blocks.txt* for *G.726*

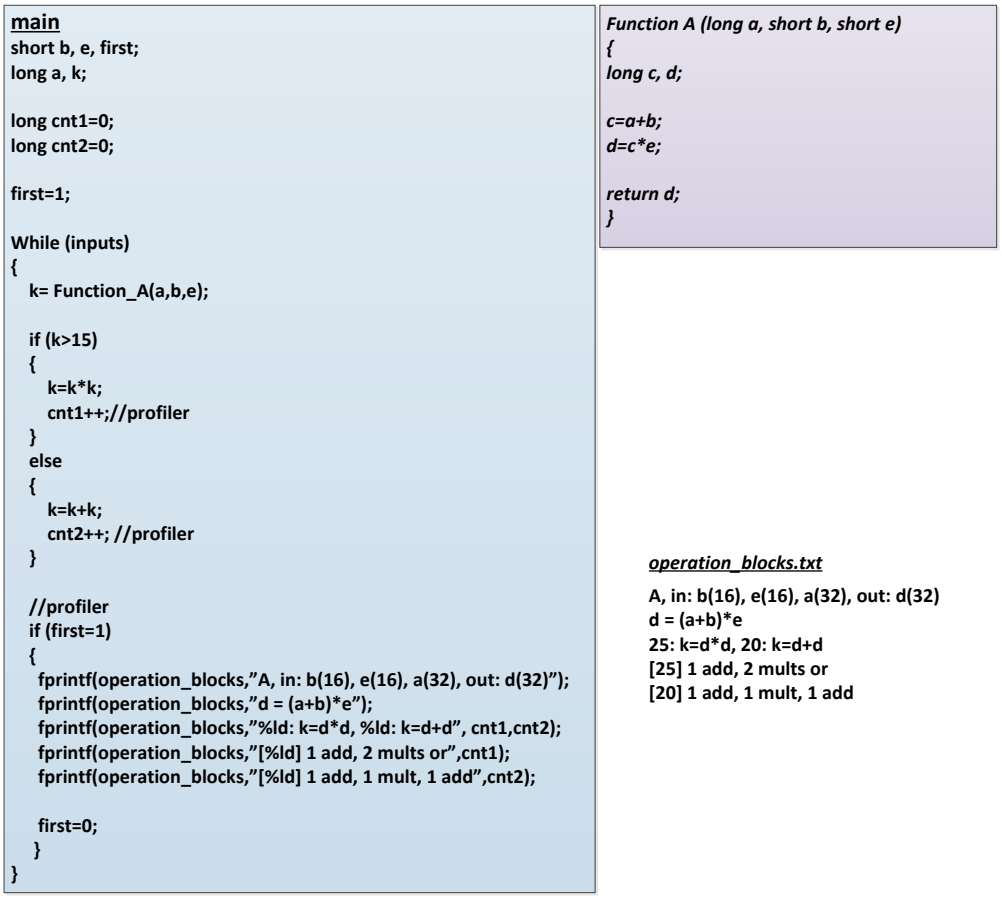
```
--uppol2 (inout(16): AL[], input(16): PLT[]):
    temp1'= 0+const(AL[1])
    (| PLT[0]-PLT[1])
    temp1'= 0-const(AL[1])
    (| PLT[0]-PLT[2])
    temp2'=(temp1'+const)+AL[2]*32512
    (| temp2' - 12288)
    (| temp2' + 12288)
    AL[2]=temp2'

    5adds, 1mult, 3adds


--uppol1 (inouts(16): AL[], PLT[]):
    (| PLT[0]-PLT[1])
    temp1'=const+AL[1]*32640
    temp2'= 15360-AL[2]
    (| temp2'-temp1')
    (| temp2'+temp1')
    AL[1]=temp2'

    2adds, 1mult, 3adds


--filtez (inputs(16): DLT[], BL[], output: SZL(16)):
    SZL=[x6] SZL+BL[]*(DLT[]+DLT[])


--filtep (inputs(16): RLT[], AL[], output: SPL(16)):
    SPL=[x2]SPL+AL[]*(RLT[]+RLT[])
```

Figure 3.11: Part of *operation_blocks.txt* for *G.722*

```
Lsp_Qnt(LspVect[] (16),CodStat.PrevLsp[](16), output: Rez(32) )
[x18]Wvect[]=LspVect[]-LspVect

[x10] div:Wvect[]=0x0020/Wvect(16)=>[x15] 1sub(32),1add(16)
[x10] LspVect=LspVect-const, PrevLsp=PrevLsp-const, 2 subs
[x10] Tmp0(16)=PrevLsp*12888+16384, LspVect=LspVect-Tmp0, [x10] 1mult,1add,1sub
[x10] PrevLsp=PrevLsp+const

--Lsp_Svq(inputs:LspVect,Wvect, output: Rez(32))

[x768]{
    [x3]Tmp[](16)=Wvect*const+16384
    [x3]Acc0(32)=^(LspVect*Tmp)+Acc0
    [x4]Acc0(32)=Acc0-const*Tmp+Acc0
    const=const+const1
}

[x3] Rez=Rez+cont

Line.LspId(32)=Rez

-------------------------------------------------------------------


Wght_Lpc(input: UnqLpc[](16),output: PerLpc[](16))
[x60] PerLpc[]=UnqLpc*const+16384

[x60] 1mult,1add
```

Figure 3.12: Part of *operation_blocks.txt* for *G.723.1*

It can be noticed, also, that, usually, after an operation block a summary is following that describes the number and the sequence of the block's operations. ***Figure 3.13*** that follows, shows an overview of the profiling method of this section. Scanning of operations is needed only once and for this reason a flag (named *first*) is used.



```
main
short b, e, first;
long a, k;

long cnt1=0;
long cnt2=0;

first=1;

While (inputs)
{
   k= Function_A(a,b,e);

   if (k>15)
   {
      k=k*k;
      cnt1++;//profiler
   }
   else
   {
      k=k+k;
      cnt2++; //profiler
   }

   //profiler
   if (first=1)
   {
    fprintf(operation_blocks,"A, in: b(16), e(16), a(32), out: d(32)");
    fprintf(operation_blocks,"d = (a+b)*e");
    fprintf(operation_blocks,"%ld: k=d*d, %ld: k=d+d", cnt1,cnt2);
    fprintf(operation_blocks,"[%ld] 1 add, 2 mults or",cnt1);
    fprintf(operation_blocks,"[%ld] 1 add, 1 mult, 1 add",cnt2);

    first=0;
   }
}
```

```
Function A (long a, short b, short e)
{
long c, d;

c=a+b;
d=c*e;

return d;
}
```

operation_blocks.txt

```
A, in: b(16), e(16), a(32), out: d(32)
d = (a+b)*e
25: k=d*d, 20: k=d+d
[25] 1 add, 2 mults or
[20] 1 add, 1 mult, 1 add
```

Figure 3.13: *Overview of the operation blocks profiling method*

Then, after all the operation blocks were available, a careful studying followed in order to detect sequential operation with data dependencies, that could lead to an appropriate arithmetic pattern, worthy to be implemented. This studying was mainly focused on the operation blocks of G.722 and G.723.1 that are more complicated and rich in operations.

The dominate operation, of course, is that of multiply-accumulation (MAC) as, especially in G.722 and G.723.1, there are a lot of different filters that are developed and this is performed, mainly, through this operation (MAC). But, except from multiply-accumulation, also, other operation schemes were noticed to can be applied, like:

- ***mac – add*** $[\ a * b + c + d\ ]$
- ***add – mac*** $[(a + b) * c + d]$
- ***add – mac – add*** $[(a + b) * c + d + e]$
- ***add – add – add*** $[a + b + c + d]$, especially for G.726 that includes a lot of sequential additions

For this reason, there was a need of finding a way to measure the efficiency of these different arithmetic schemes. The method that was followed for this measuring is described in the next paragraph.

3.5 Modeling based on operation factoring

In order to measure the efficiency of the aforementioned arithmetic schemes, there was a focus on G.722 and G.723.1. The reasons for that were, firstly, the complexity of these two speech codecs and, as a result, the interesting operation blocks they include, but, mainly, the existence of a library (named *basop.c*), for each of them, that includes functions for any arithmetic operation.

Actually, in the main C-codes of G.722 and G.723.1 there is no arithmetic operations like $a = b - c$, but $a = sub(b, c)$, where *sub(a,b)* is a function from the library *basop.c*, which is included in the main C-code. Also, other more complicated operations, like MAC, are performed through the basic functions (*add*, *sub*, *mult*). ***Figure 3.14*** that follows, provides a clearer view.

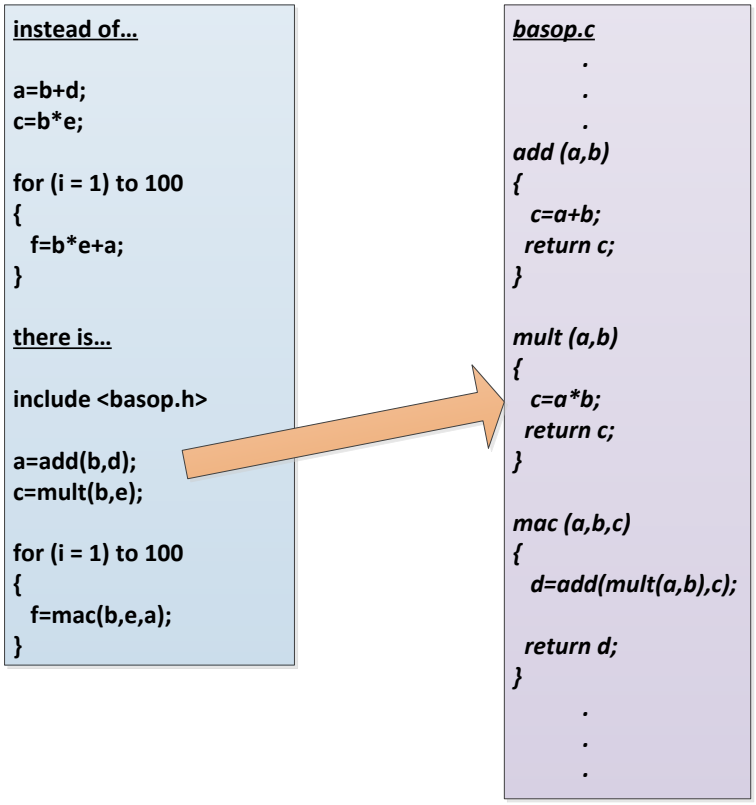


Figure 3.14: *Using arithmetic functions from the library* ***basop.c***

In *basop.c* there are arithmetic functions of any kind and even for numbers of different size. For example, there are two functions for addition, one for numbers composed of 16 bits (shorts), named *add()*, and one for numbers composed of 32 bits (longs), named *L_add()*.

Therefore, there was a thought of developing a list, very similar to the previously developed list *operations* but simpler, in order to keep a record only of the operations performed and the type of numbers that participate. The name of this list was ***benchmark*** and it consisted of ***bench*** elements that were composed only of three fields:

- *char* **symbol**, in order to store the type of the operation
- *short* **bits**, in order to store the type of the operands
- *bench* ***next,** in order to create the next element of the list

In order to create this list, a function named ***benchmarking()*** was developed. ***Table 3.4*** contains its description and ***Figure 3.15*** shows the easy way that it was applied through the functions of ***basop.c***. As it is depicted in this figure, with *benchmarking()* there was no reason to detect all the arithmetic operations of the C-code, as it happened during the

previous approach of the list *operations*. Also, it has to be noted that **additions and subtractions were, both, stored as additions because their hardware implementations is the same**.

Function Name	Input Parameters	Output Parameters	Function Description
void **benchmarking()**	*char* **symbol**, *short* **bits**	Adds an element to the list ***benchmark***	It is called only inside the basic arithmetic functions of addition, subtraction and multiplication of *basop.c*. It adds a *bench* element to the list *benchmark* and it completes the fields of the element.

Table 3.4: *Describing of the **benchmarking()** profiling function*

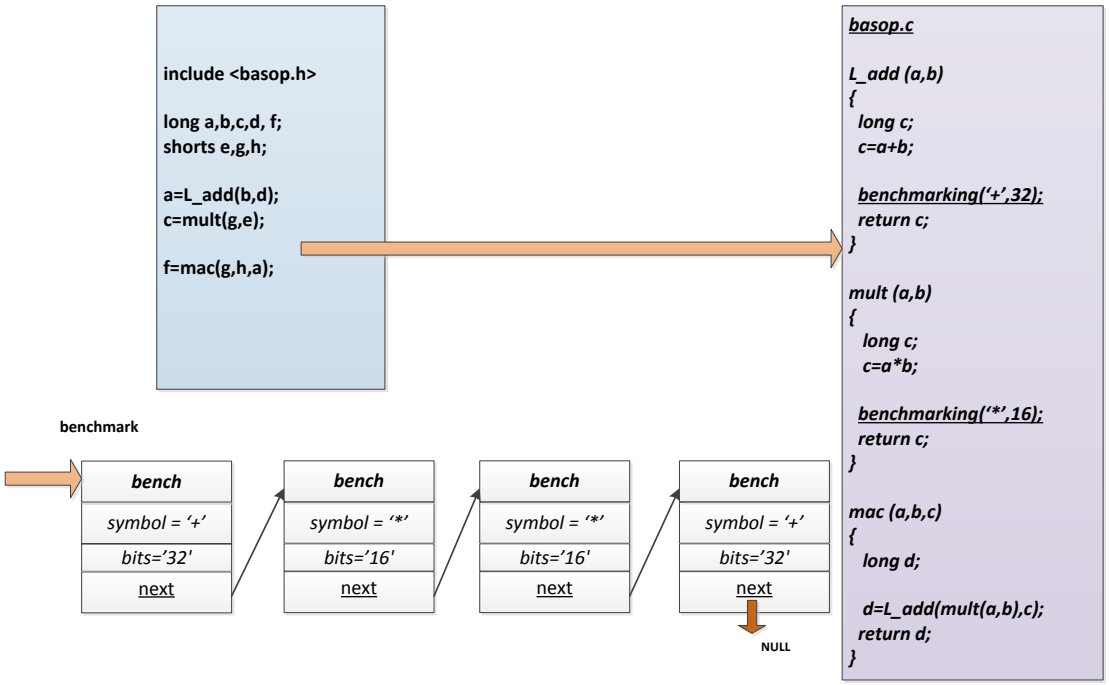


Figure 3.15: *Overview of the profiling function **benchmarking()***

After the list *benchmark* was available and in order to measure the efficiency of the different proposed arithmetic schemes, some final profiling functions were developed. The general

idea was: scanning of the list ***benchmark*** and count the times that the sequence of operations, corresponding to each operation scheme, appears.

Actually, each function models a circuit with the following arithmetic units:

1. **a 16-bit adder**
2. **a 32-bit adder**
3. **a 16-bit multiplier (with the result in 32-bit representation)**
4. **one of the following units:**
    - none (simple version)
    - a mac unit $(a * b + c)$
    - a mac-add unit $(a * b + c + d)$
    - an add-mac unit $((a + b) * c + d)$
    - an add-mac-add unit $((a + b) * c + d + e)$
    - an add-add-add unit $(a + b + c + d)$

Also, within the framework of this model, it was supposed that any unit needed only a cycle to perform its operation, but, on every cycle only one of the four arithmetic units could be used. As a result, **each function scanned the list *benchmark* and counted the times that its special arithmetic unit can be applied to the sequence of the operations**. The purpose was to have a measure of how efficient is any of these arithmetic units for the encoding-decoding procedure for different inputs of speech. Obviously, **this procedure included an error factor as the data dependencies were not taken into consideration**, but that could be evaluated from the studying of the available *operation_blocks.txt* files.

Finally, functions **consecutively** printed their results to another file, named ***counters.txt***, in order to provide a general view. The method was based in a central function named ***weighting()***, which called all the other functions. ***Table 3.5*** contains the descriptions of all these functions and ***Figure 3.16*** gives an overview of the whole procedure.

Function Name	Input Parameters	Output Parameters	Function Description
void **weighting()**	*No Input*	*No Output*	It is called in the end of *main* function and it calls all the other functions.
void **simple_version()**	*The list* ***benchmark***	Writes on **counters.txt**	It models the case there are *only* the basic arithmetic units (**1. – 3.**)
void **mac_version()**	*The list* ***benchmark***	Writes on **counters.txt**	It models the case that, except from the basic arithmetic units (**1. – 3.**), there is also a *mac* unit (it is applied to any ***,+** sequence).
void **mac_add_version()**	*The list* ***benchmark***	Writes on **counters.txt**	It models the case that, except from the basic arithmetic units (**1. – 3.**), there is also a *mac-add* unit (it is applied to any ***,+,+** sequence).
void **add_mac_version()**	*The list* ***benchmark***	Writes on **counters.txt**	It models the case that, except from the basic arithmetic units (**1. – 3.**), there is also an *add-mac* unit (it is applied to any **+,*,+** sequence).
void **add_mac_add_version()**	*The list* ***benchmark***	Writes on **counters.txt**	It models the case that, except from the basic arithmetic units (**1. – 3.**), there is also an *add-mac-add* unit(it is applied to any **+,*,+,+** sequence).
void **add_add_add_version()**	*The list* ***benchmark***	Writes on **counters.txt**	It models the case that, except from the basic arithmetic units (**1. – 3.**), there is also an *add-add-add* unit (it is applied to any **+,+,+** sequence).

Table 3.5: *Describing of the efficiency measuring profiling functions*

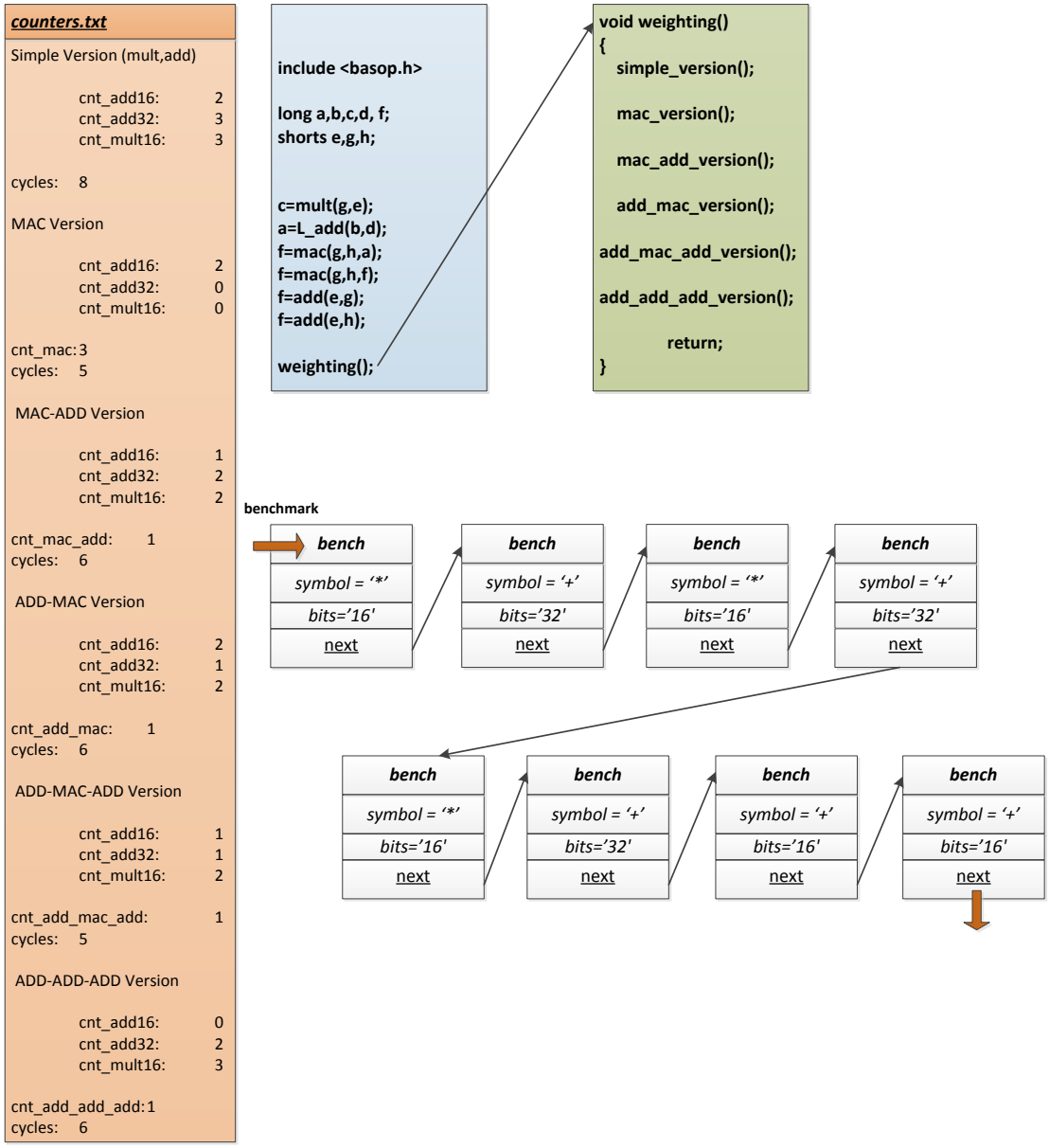


Figure 3.16: *Overview of the efficiency measuring profiling functions*

After all the available input speeches were used, in order to measure the efficiency of the different operation schemes for the speech codecs G.723.1, G.722 and G.726[1], the results were inserted in *Microsoft Excel*. The next procedure was to apply different weighting factors to each operation scheme according to how fast each of these arithmetic units is supposed to be. For example, a pairing of weighting factors could be $\left[adder_{16_{bits}}, adder_{32_{bits}}, mult_{16_{bits}}, mac, mac\_add\right] = [0.4, 0.6, 1.8, 2.2^{2}, 2.4]$,

For this procedure of weighting, results of previous research data were used in combination with the experience of the Lab's professor. Finally, for each speech codec, according to the described model, a table was created comparing the efficiency of implementation, from the aspect of *cycles* and *weighting*, to the efficiency of the *simple version*. The following figures (***Figure 3.17*** to ***Figure 3.20***) provide a view of the efficiency comparing and of the results.

	add-16 bits	***add-32 bits***	***mult-16 bits***	***mult-32 bits***	
weighting	**0,4**	**0,6**	**1,8**	**2,3**	
	mac	***mac - add***	***add - mac***	***add-mac-add***	***add-add-add***
weighting	2,15	2,2	2,45	2,9	1,2

Figure 3.17: *Overview of the applied weights*[3]

[1] G.711 was omitted in this procedure, because of the simplicity of the performed operations in its code. For G.726 that there is no *basop.c* library, the C-code was scanned and function ***benchmarking()*** was applied to every operation.

[2] For compound arithmetic units like *mac*, *mac_add* etc., a slightly lower weight was used than the addition of the parts that it is composed of. For example, for *mac* should have been $w_{mult_{16}} + w_{adder_{32}} = 1.8 + 0.6 = 2.4$, but a weight of 2.2 is applied.

[3] For the weights of the first row, the Microprocessors Lab's results were used. For the compound units the philosophy of footnote [2] was applied. It has, also, to be mentioned that implementation of *mac-add* was considered more efficient than that of *add-mac*. This can be explained from the efficient way that *mac-add* was implemented within the framework of this thesis (see *Chapter 4*).

	cycles	weight	cycles improvement	weight improvement
simple version (mult,add)	134562254	159264308,2	-	-
mac version	71450713	143608749,2	46,9014%	9,8299%
mac-add version	127276570	156539924,8	5,4144%	1,7106%
add-mac version	69661748	141709179,7	48,2308%	11,0226%
add-mac-add version	126198179	157481972,7	6,2158%	1,1191%
add-add-add version	132125768	158932089,6	1,8107%	0,2086%

Figure 3.18: *Comparing efficiency among different arithmetic schemes for G.723.1 (5.3 kbps)*

	cycles	weight	cycles improvement	weight improvement
simple version (mult,add)	31178160	26607413	-	-
mac version	22171425	25511156,25	28,8880%	4,1201%
mac-add version	19917498	24311241,4	36,1171%	8,6298%
add-mac version	17568690	24940782,75	43,6507%	6,2638%
add-mac-add version	19249053	26136377,7	38,2611%	1,7703%
add-add-add version	23552580	26519333,8	24,4581%	0,3310%

Figure 3.19: *Comparing efficiency among different arithmetic schemes for G.722 (64 kbps)*

	cycles	weight	cycles improvement	weight improvement
simple version (mult,add)	112641384	73696687,6	-	-
mac version	105186642	68421462,9	6,6181%	7,1580%
mac-add version	97731900	64467197,4	13,2362%	12,5236%
add-mac version	97731900	66251454,3	13,2362%	10,1025%
add-mac-add version	90277158	65279085,6	19,8544%	11,4220%
add-add-add version	46602134	60212358,2	58,6279%	18,2971%

Figure 3.20: *Comparing efficiency among different arithmetic schemes for G.726 (32 kbps – a-law)*

3.6 Conclusions

Obviously, in order to decide which operation scheme is the most appropriate to implement, a very careful study, of the *operation_blocks.txt* file for each speech codec, had to be realized. After this procedure, that was again, mainly, focused on G.722 and G.723.1, the final conclusions were:

1. **Multiply-accumulation (*MAC)* dominates**

   Efficiency of *MAC* could be easily understood from the comparing results of the previous paragraph. But, also, there are numerous points in the speech codecs' algorithms that it is met, especially in G.723.1 and G.722 for the construction of filters (see ***Figure 3.21, Figure 3.22***). This is a reason that this arithmetic unit is included in the most of the Digital Signal Processors (DSPs).

```
[x720] Acc0(32)=^(Temp[]*Temp[])+Acc0,1mac
[x120] Acc0=^(Temp[]*Temp[])+Acc0,1mac

DecCng.SidGain(16)=Acc0+0x8000

[x120] Acc0=^(Temp[]*Temp[])+Acc0,1mac
```

Figure 3.21: *A snapshot from **G_723_1_operation _blocks.txt*** with *three **MAC** operation blocks*

```
[x11] temp1= const(from_a_table)*const(from_a_table)+temp1, temp2=
const(from_a_table)*const(from_a_table)+temp2
```

Figure 3.22: *A snapshot from **G_722_operation _blocks.txt*** with *two **MAC** operation blocks*

2. ***ADD-MAC* is not as much efficient as it seems to be**

   From the ***Figure 3.18***, ***Figure 3.19*** a possible conclusion is that *ADD-MAC* is worthy to be implemented. But, studying of *operation_blocks* files showed that the arithmetic form of $[(a+b)*c+d]$ is very rare. An explanation of the reason that the *weighting* procedure presents this arithmetic unit to be efficient, could be the existence of numerous *MACs*.

For example, a loop of the form $[x4]\{ a * b + c\}$, in the benchmarking list would be applied in the sequence: *, +, *,+, *,+, *,+. As the function *add_mac_version()* would scan the sequence, would detect the following two sequences of its arithmetic scheme *, **+, *,+**, *,**+, *,+**. Obviously, these are not sequences of the *add_mac* scheme.

3. ***ADD-MAC-ADD* and *ADD-ADD-ADD are not efficient***

   The first scheme is, obviously, not efficient neither for the *weighting* procedure nor in the studying of operation blocks (as even *ADD-MAC* is a rare form).

   The second scheme is very efficient for G.726 as its algorithm consists of numerous sequential additions and, actually, this was the reason that it was selected as one of the modeling arithmetic units. But, it has not a very general structure in order to be efficient to the other basic speech codecs, also.

4. **Efficiency of *MAC-ADD***

   This arithmetic unit seems efficient to be implemented. Procedure of *weighting* indicated that and, also, it can be applied in the operation blocks of speech codecs (see ***Figure 3.23***, ***Figure 3.24***).

```
[x60]{

        Acc0=Acc0+^(Pf.ScGn*Temp),1mac
        DataBuff=Acc0+0x8000
}
```

Figure 3.23: *A snapshot from **G_723_1_operation _blocks.txt*** with a ***MAC-ADD** operation block*

```
temp2'=(temp1'+const)+AH[2]*32512
temp1' = NBH*32512
temp2' = temp1'+const(ih)
(| temp2'-22528)
```

Figure 3.24: *A snapshot from **G_722_operation _blocks.txt*** with two ***MAC-ADD** operation blocks*

5. **A combination of *MAC* and *MAC-ADD* units would be the most efficient solution**

   From the aforementioned conclusion it is obvious that a *MAC* unit with the ability of performing, also, *MAC-ADD*, would be the most efficient solution to be implemented.

4 Hardware Implementation

4.1 Introduction

In this section the *Speech Coding Arithmetic Unit* (**SCAU**) is described. The purpose of this arithmetic unit is the efficient implementation of two arithmetical modes/operations, according to the conclusions of the whole profiling method. The first mode (*MAC* mode) is the implementation of the multiply-accumulate operation for *n*-times $(\sum_{i=1}^{n} x_i \times a_i)$. The second mode (*MADD-ADD* mode) is the implementation of multiply-add-add operation $(x \times a + b + d)$.

4.2 Tools and Flow

The tools which were used for the hardware implementation of *SCAU* were:

- *ModelSim* (by *Mentor Graphics*), for the part of Verilog-HDL code simulation and verification.
- *Synopsys Software Tools*, for the part of synthesis and the estimation of delay, area and power.

Both of these tools were used within the framework of the *Flow* Tool, which has been developed in Microprocessor's and Digital Systems Lab. Finally, it has to be mentioned that for synthesis the *tsmc* library was used in a 90-nm technology.

4.3 Circuit Description

4.3.1 General Description

Figure 4.1 shows the pins of SCAU. It can be noticed that there are 7 pins as inputs and 2 pins as outputs. ***Table 4.1*** gives the description of any of these pins.

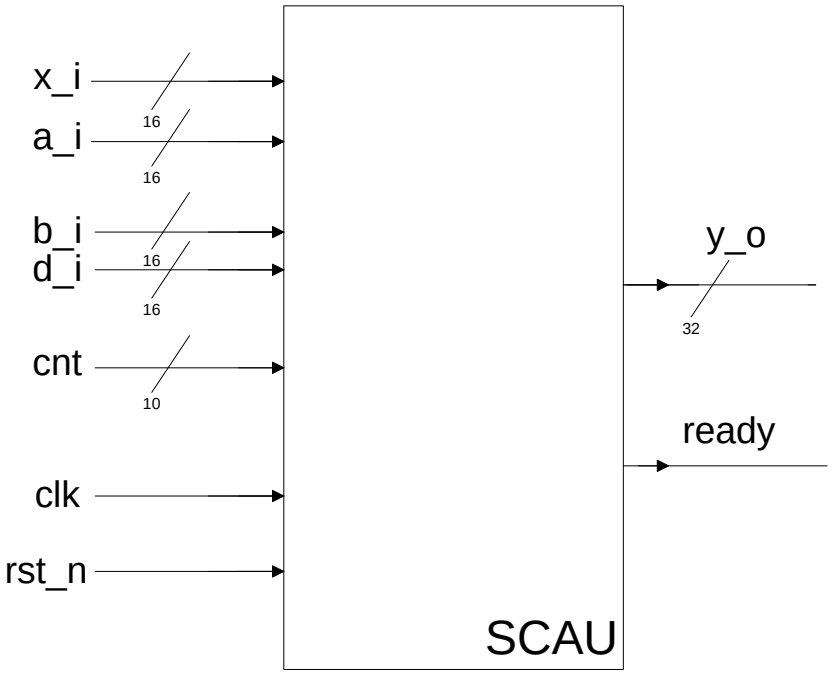


Figure 4.1: *Pinout of SCAU*

Name	***Type***	***Width (bits)***	***Description***
clk	input	-	clock signal
rst_n	input	1	reset signal (Active low)
x_i	input	16	multiplier
a_i	input	16	multiplicand
b_i	input	16	1st addend
d_i	input	16	2nd addend
cnt	input	10	counter (counts the number of iterations for the MAC operation)
y_o	output	32	the final result of each operation (it is available until a new final result is loaded)
ready	output	1	signal that is set HIGH **a cycle before** *y_o* has the final result (it notifies the CPU of data availability on the next cycle)

Table 4.1: *Description of SCAU pins*

The way that **SCAU** functions, is the following (it is supposed that *rst_n* is HIGH):

- ***MAC mode***

For the MAC mode, it has to be set, on the input pin *cnt*, the number of iterations that multiply-accumulation has to be performed. On the same clock positive-edge the first input pair (*x_i, a_i*) has to be ready on the pins *x_i*, *a_i* and after that, cycle-by-cycle, input pairs are inserted, according to the value of *cnt* that it was chosen. For example, for an operation of 100 multiply-accumulations (see ***Figure 4.2***), on the first positive edge, *cnt* would be 100 and x_i_1, a_i_1 loaded on the corresponding pins. On the second cycle, x_i_2, a_i_2 would be loaded, on the third cycle x_i_3, a_i_3, etc until the cycle 100 when the pair x_i_{100}, a_i_{100} should be loaded.

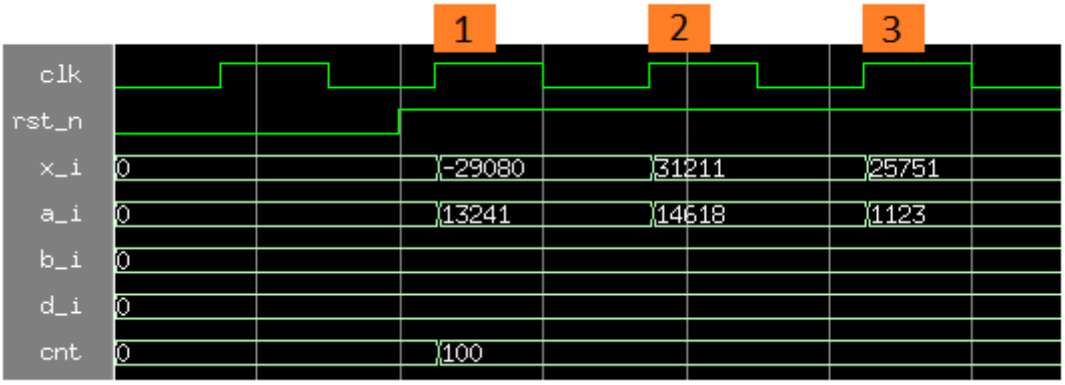


Figure 4.2: *Beginning of a MAC mode*

The final (proper) result of the whole MAC operation $(x_1 \times a_1 + x_2 \times a_2 + \cdots + x_{99} \times a_{99} + x_{100} \times a_{100})$ will be on the output pin *y_o* on the cycle 103. On the cycle 102 the output pin *ready* is set HIGH and notifies the CPU that on the next cycle (103), the final result will be available on *y_o* (see ***Figure 4.3***).

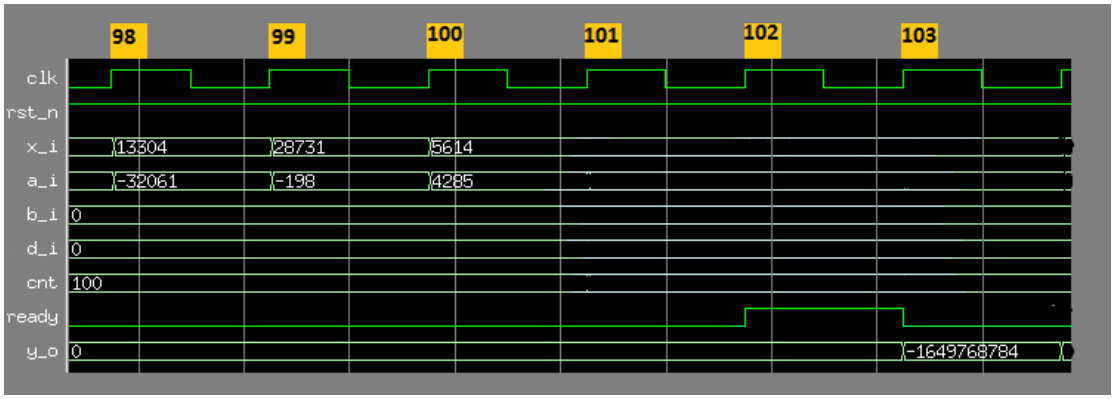


Figure 4.3: *Ending of a MAC mode*

On cycle 101 a new value for *cnt* can be loaded for a new sequence of input pairs and a new MAC operation. Also, something important is that, on MAC mode, inputs *b_i*, *d_i* must have the zero value, at least on the first cycle after the last input pair (cycle 101 of the previous example).

- ***MADD-ADD mode***

For the MADD-ADD mode, as it is expected, the value one (1) has to be loaded on the input pin *cnt*. On the same clock positive-edge the inputs *x_i, a_i, b_i, d_i* has to be ready on the corresponding pins.

The result of the MADD-ADD operation $(x_i \times a_i + b_i + d\_i)$ will be on the output pin *y_o* on the cycle 4 (if it is supposed that inputs are loaded on cycle 1). On the cycle 3 the output pin *ready* is set HIGH and notifies the CPU that on the next cycle (4), the final result will be available on *y_o*. ***Figure 4.4*** shows an example of MADD-ADD mode.

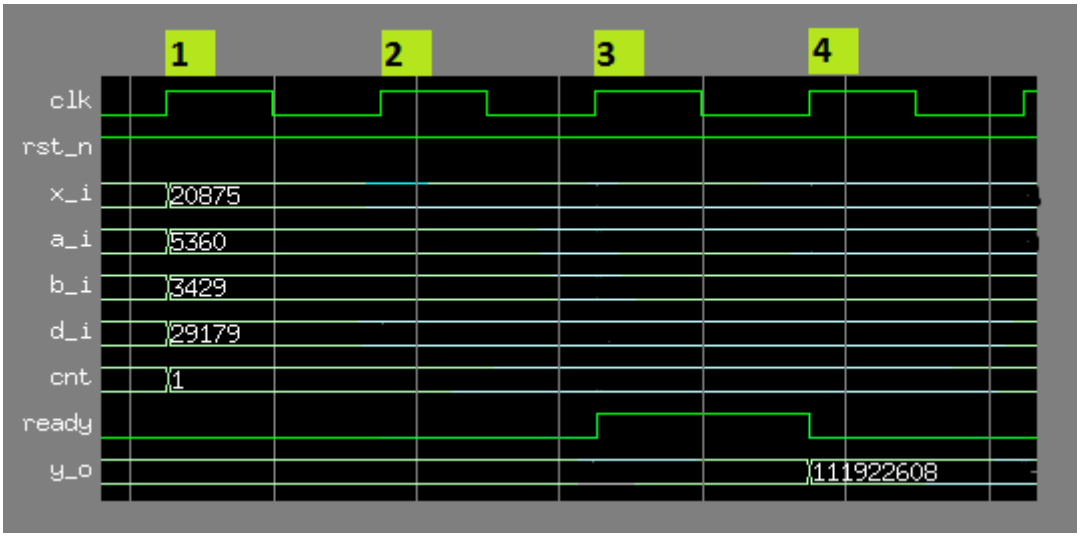


Figure 4.4: An *example of MADD-ADD mode*

On cycle 2 the value one (1) can be loaded on pin *cnt* and the rest of the inputs for a new MADD-ADD operation.

- ***Transition from one mode to the other***

From the aforementioned description of the two different arithmetic modes, is easy to understand that controlling of which mode **SCAU** operates, is a matter of which value is loaded on the input *cnt*. As a result, after the finish of inserting the inputs of each mode, transition to the other mode can happen by loading the proper *cnt* value.

For example, in the previous example of MAC mode, if it is needed to switch to MADD-ADD mode after the end of the whole MAC operation, the value one (1) has to be loaded, on cycle 101, on the input pin *cnt* and, on the same cycle (positive-edge), the inputs *x_i, a_i, b_i, d_i* has to be ready on the corresponding pins.

One the other hand, in the previous example of MADD-ADD mode, if it is needed to switch to MAC mode, the number of iterations of the multiply-accumulation has to be loaded, on cycle 2, on the input pin *cnt*. On the same clock cycle, the first input pair (*x_i, a_i*) has to be ready on the pins *x_i*, *a_i* and after that, cycle-by-cycle, input pairs are inserted, according to the value of *cnt* that it was chosen.

Figure 4.5 shows a transition from *MAC* to *MADD-ADD* mode.

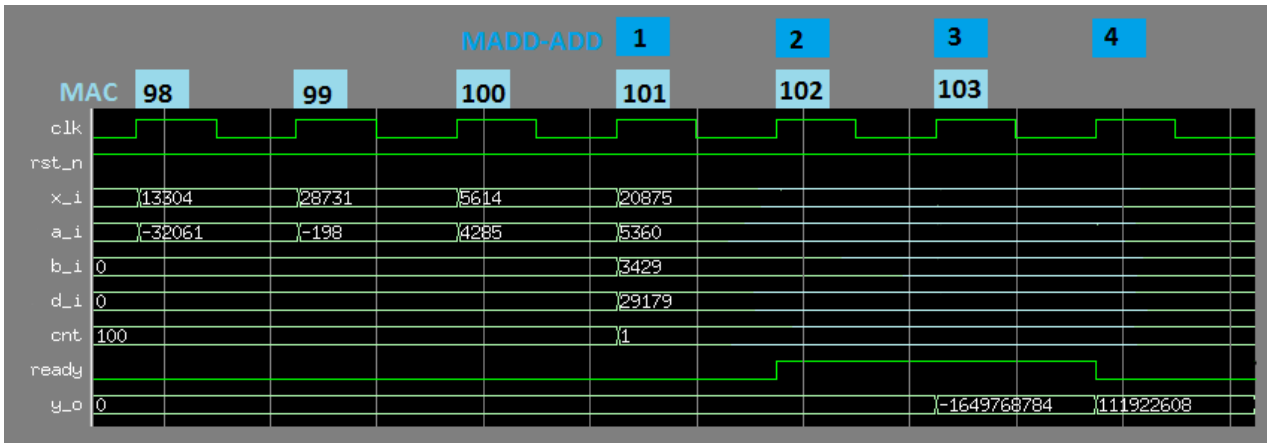


Figure 4.5: *Transition from MAC to MADD-ADD mode*

4.3.2 Detailed Description

Figure 4.6 shows the block diagram of SCAU and explains its internal function. ***Figure 4.7*** and ***Figure 4.8*** show the active data paths for each mode.

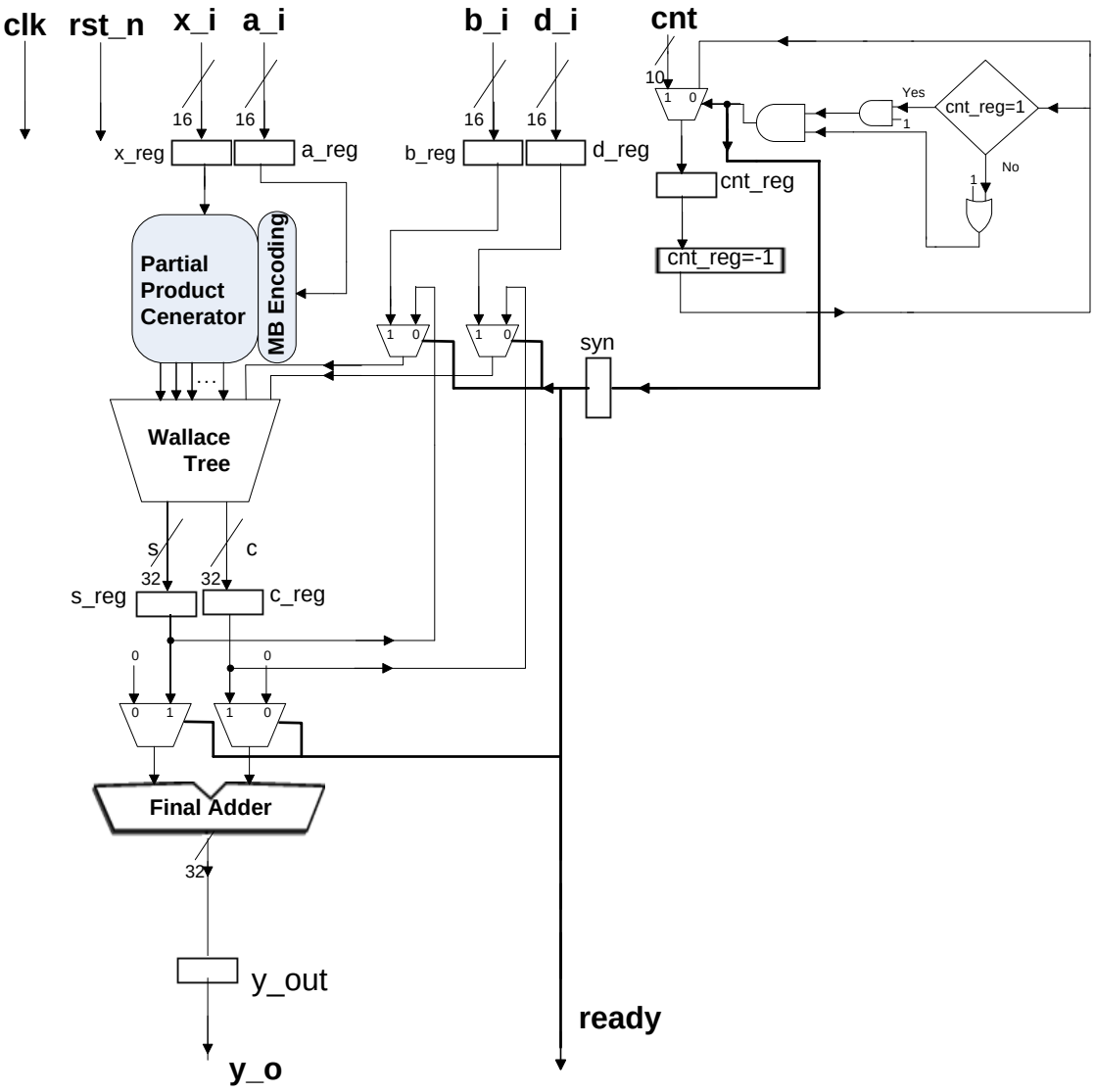


Figure 4.6: *Block diagram of SCAU*

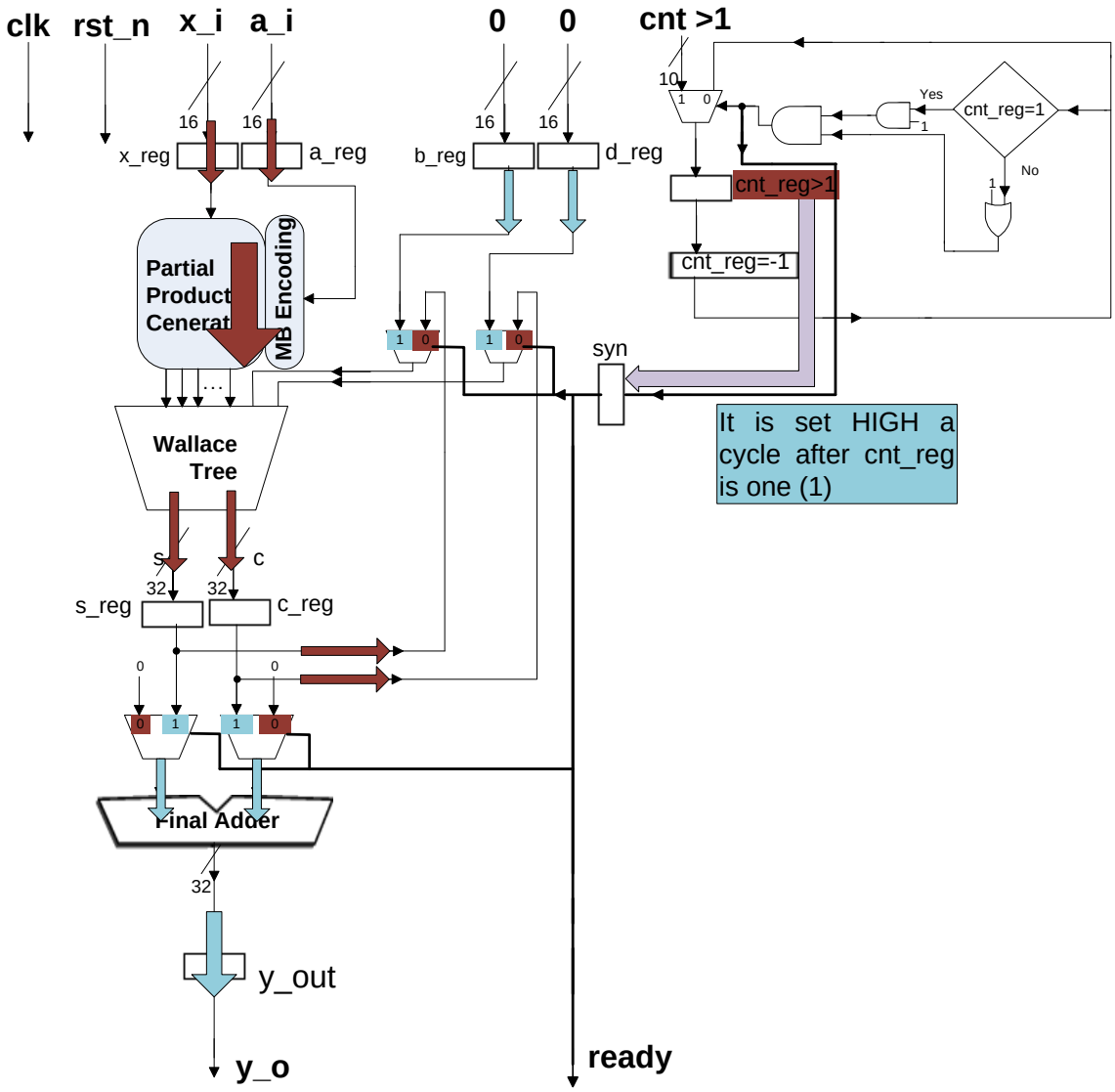


Figure 4.7: *Data path when in **MAC** mode*

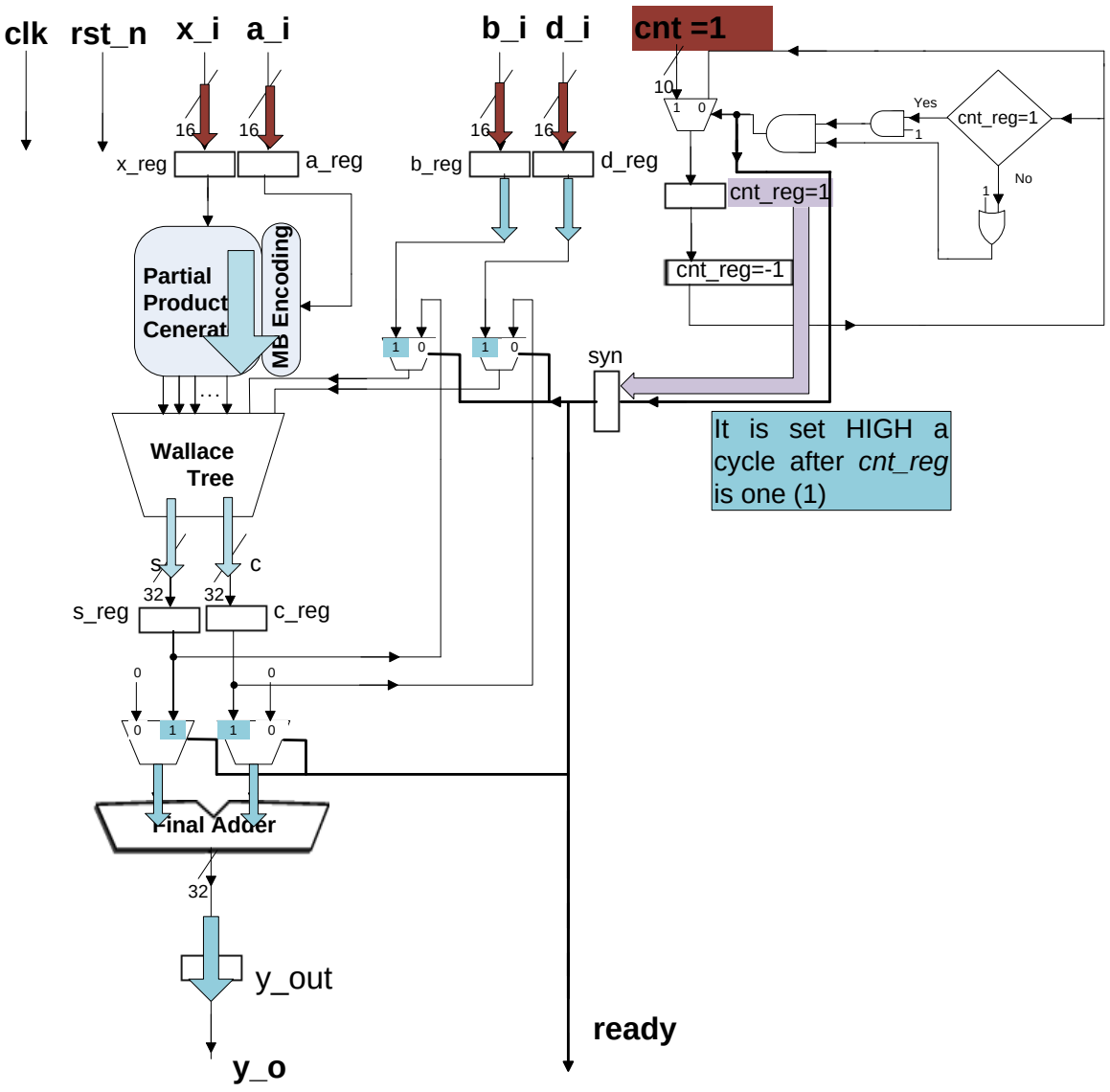


Figure 4.8: *Data path when in **MADD-ADD** mode*

SCAU consists of three parts:

- **a multiplier**
- **a final adder**
- **a counter control unit**

A more detailed view of these parts and of the way they co-operate, in order to produce the outputs from the inputs, is provided below.

- **Multiplier**

Multiplier consists of a **Partial Product Generator (PPG)** and a **Wallace Tree Adder**. Inputs *x_i*, *a_i* are inserted to *PPG*, through the input registers *x_reg* and *a_reg*. As ***Figure 4.6*** shows, input *a_i* is firstly inserted to a **Modified-Booth Encoder** and after that to *PPG*.

After PPG, partial products, that have been produced, are inserted to the Wallace Tree. Wallace Tree adds the partial products and gives a result in a carry-save representation. Finally, carry and save are stored to *c_reg* and *s_reg*, respectively.

In ***Figure 4.7*** it is shown that, after storage to *c_reg* and *s_reg,* carry and save return back to the Wallace Tree as two more inputs of it. This is, actually, the main point of the MAC mode. As new input pairs are inserted, cycle-by-cycle, and new partial products are produced, carry and save of every cycle is accumulated, via Wallace Tree, to the previous carry-and-save result. **In SCAU, accumulation happens before the Final Adder (carry-save representation) and *not* after the Final Adder (that the result is in normal representation).**

- **Final Adder**

Final Adder is a fast arithmetic unit that takes the ***final*** carry and save as inputs, adds them, and provides the final result of accumulation as an output. This output is stored on *y_out*.

In order to ensure that the Final Adder will be used *only* for the addition of the final carry and save, two multiplexers are used on its inputs, which are controlled from the Counter Control Unit (register *syn*). The same signal sets output *ready* HIGH, in order to notify the CPU that on the next cycle the final result will be available.

As a result, it is easily understood that the whole synchronization of SCAU is directed from the Counter Control Unit.

- **Counter Control Unit and synchronization**

The purpose of Counter Control Unit is to control, first of all, the inputs of Wallace Tree in order to ensure that SCAU works properly. Firstly, the double role of register *syn* will be described and, then, how *syn* is controlled by the input *cnt* and the Counter Control Unit.

In ***Figure 4.7*** can be noticed that the feedback of accumulation (two of the inputs of Wallace Tree) is the outputs of two multiplexers, which are controlled by register *syn*. If ***syn* is zero**, then the current carry and save are inserted to the Wallace Tree with the new partial products and they form the new carry and save. If ***syn* is one (1)**, *b_i* and *d_i*, and *not* carry and save, are inserted to the Wallace Tree to be added with the partial products.

So, during a MAC mode, register *syn* has to be loaded with zero in order to let the accumulation to take place. On the other hand, there are two situations (that actually happen in the same cycle as explained later in this section), in which inputs *b_i* and *d_i* need to be inserted in the Wallace Tree and, as a result, *syn* has to be loaded with one (1).

The first of them is when a MAC mode is finished and a second MAC mode is about to start. In this case, the example of paragraph *4.3.1* about MAC mode will help understanding. One hundred input pairs is assumed. Input pair 100 (x_i_{100}, a_i_{100}) is inserted and loaded to *x_reg*, *a_reg*, respectively, on **cycle 100**. They will be multiplied during **cycle 101** and the new (and last) pair of carry and save will be loaded on *c_reg* and *s_reg*. On the same cycle (101), a new MAC operation starts. New values of inputs (*x_i*, *a_i*, *cnt* and *b_i=d_i=0*) are loaded on the input registers (*x_reg*, *a_reg*, *cnt_reg*). On **cycle 102**, *syn* is set HIGH and three different things happen:

a) the final carry and save of the former MAC are inserted into the Final Adder,

b) the first input pair of the latter MAC operation is inserted into the Partial Product Generator,

c) carry and save from a) are not allowed to insert in the Wallace Tree (*syn* is one (1)), because a new accumulation starts and zero-values are inserted to the two inputs of Wallace Tree that come from the multiplexers, via *b_reg* and *d_reg*. This fact explains the note of paragraph *4.3.1* about MAC mode that on cycle 101, *b_i* and *d_i* must be zeroed and, as a result, on cycle 102 *b_reg* and *d_reg* will be zeroed.

The second situation in which *syn* has to be loaded with one (1) is when a MADD-ADD situation is about to start. For example, if for inputs (*x_i*, *a_i*, *b_i, d_i, cnt*=1) on cycle 1, then, on cycle 2, *syn* has to be one (1) in order to let *b_i*, *d_i* pass, through *b_reg* and *d_reg* and the multiplexers, in the Wallace Tree and be added with the partial products of the multiplication.

The combination of the above-mentioned descriptions clarify that these two cases happen in the same cycle (as it was noted) and can be noticed, regardless the mode that SCAU is about

to work in, ***syn* must be HIGH a cycle after the beginning of a new operation**. This conclusion does not hold in the case that the very first operation of SCAU is a MAC operation, because there are no previous carry and save from a previous operation that should be deterred to insert into the Wallace Tree.

Furthermore, *syn* has one more very important role to play. It is connected with the output of SCAU. As already mentioned in the previous MAC example, *syn* is set HIGH on cycle **102** when the final carry and save are inserted into the Final Adder through the multiplexers. In ***Figure 4.6***, it can be noticed that output *ready* is directly connected with *syn* and, as a result, on cycle **102** *ready* is set HIGH and notifying that on cycle **103** the final result will be ready on the pin *y_o*.

As a result, except from the aforementioned conclusion, it can be noticed that from the point of view of the output, regardless the mode that SCAU worked in for the last operation, ***syn* must be HIGH two cycles after the last input of the last operation**.

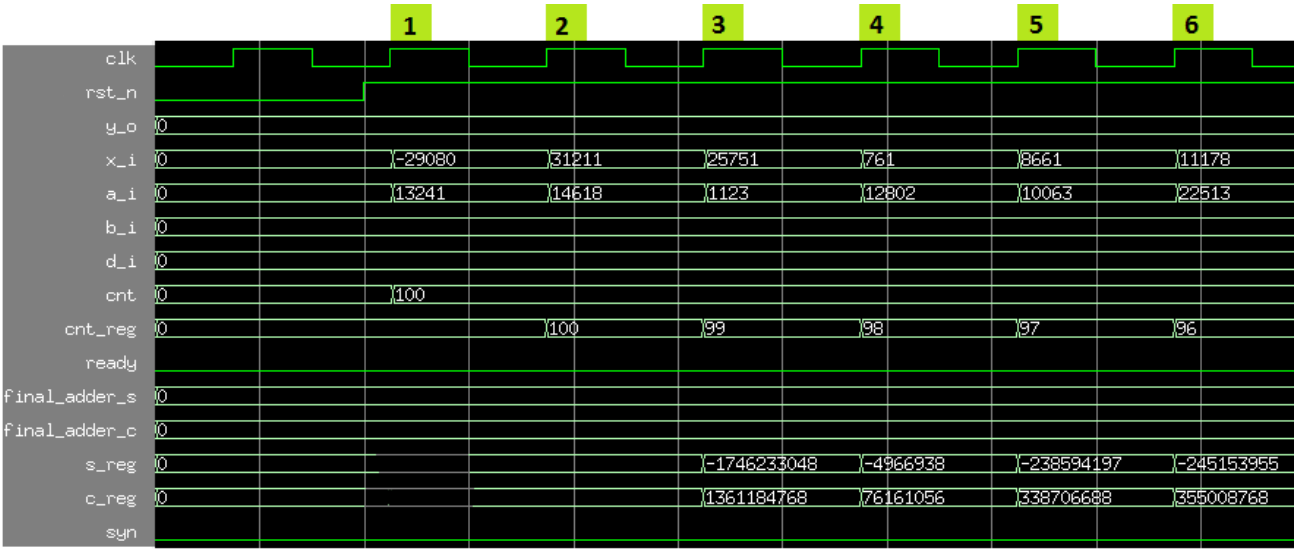


Figure 4.9: *Beginning of a MAC operation*

Figure 4.9 shows the beginning of a MAC operation. When *rst_n* is LOW and on the first clock's (clk) positive edge, first inputs *x_i*, *a_i, cnt* are loaded and everything else is zero. New input pairs are inserted on successive clock cycles, *cnt* does not change and only *cnt_reg* (which is loaded on cycle 2) is reduced by one each clock. Also, something that is important is that on cycle 3 the first carry and save results, that come from the first input pair can be noticed. On the other hand the inputs of the Full Adder (*final_adder_s*,

final_adder_c) and the output (*y_o)* are zeroed because of the multiplexers. The datapath clock cycle breakdown is as follows:

Cycle 1: input at registers *x_reg*, *a_reg*

Cycle 2: multiplication is done and result is stored at registers *s_reg*, *c_reg*

Cycle 3: carry and save is inserted back into the Wallace Tree

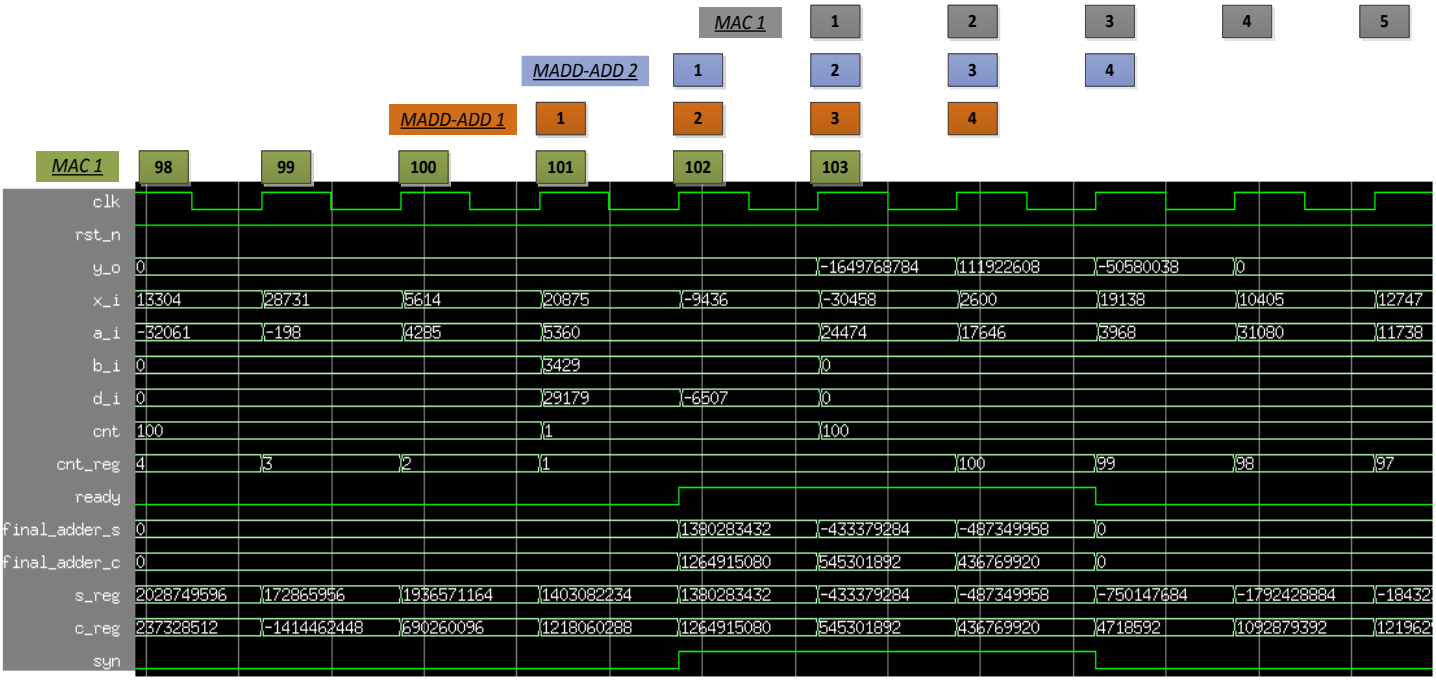


Figure 4.10: *End of the above-figured MAC operation, two sequential MADD-ADD operations and the beginning of a MAC operation*

Figure 4.10 shows the end of the MAC operation, two sequential MADD-ADD operations and the beginning a new MAC. A description of every cycle for any of these operations follows:

MAC 1 - MADD ADD 1 - MADD ADD 2 - MAC 2:

Cycle 100: Final input pair of MAC operation

Cycle 101 - Cycle 1: Final input pair is multiplied and accumulated with the previous carry and save. **-** Inputs *x_i*, *a_i, b_i*, *d_i, cnt=1* are loaded.

Cycle 102 - Cycle 2 - Cycle 1: Final carry and save are inserted in the Final Adder. *syn is HIGH* so carry and save does not get back on the Wallace Tree. *ready is HIGH* so on the next cycle

final result will be on *y_o*. **-** *syn is HIGH so b_i*, *d_i* are inserted in the Wallace Tree through *b_reg, d_reg* in order to be added with *x_i*, *a_i*. **-** Inputs *x_i*, *a_i (the same with MADD-ADD 1), b_i (the same with MADD-ADD 1)*, *d_i, cnt=1* are loaded.

Cycle 103 - Cycle 3 - Cycle 2 - Cycle 1: Final result is on *y_o*. **-** carry and save are inserted in the Final Adder. *syn is HIGH* so carry and save does not get back on the Wallace Tree. *ready is HIGH* so on the next cycle result will be on *y_o*. **-** *syn is HIGH so b_i*, *d_i* are inserted in the Wallace Tree through *b_reg, d_reg* in order to be added with *x_i*, *a_i*. **-** first inputs *x_i*, *a_i, cnt=100* are loaded.

null - Cycle 4 - Cycle 3 - Cycle 2: null **-** result is on *y_o* **-** carry and save are inserted in the Final Adder. *syn is HIGH* so carry and save does not get back on the Wallace Tree. *ready is HIGH* so on the next cycle result will be on *y_o*. **-** *syn is HIGH so b_i=d_i=0* are inserted in the Wallace Tree through *b_reg, d_reg* in order to start new accumulation. First carry and save are produced. New input pair is inserted.

null - null - Cycle 4 - Cycle 3: null **-** null **-** result is on *y_o* **-** *syn is **LOW*** so carry and save gets back on the Wallace Tree for being accumulated with next input pairs that have come.

Finally, operation of the **Counter Control Unit** can be clarified. Input *cnt* is inserted on every beginning of a new operation. On the next cycle, it is loaded on *cnt_reg* and if it is not one (see *Figure 2 – Yes = 0 and No = 1 =>multiplexer takes zero value as control signal)* it is reduced by one. This control happens in every cycle and if it is found that *cnt_reg* is one one (*Yes = 1 and No = 0 =>multiplexer takes one (1) as control signal)*, on the next cycle, *cnt* is loaded again and *syn* is set HIGH. ***Figure 4.11*** provide an overview of Counter Control Unit's waveforms.

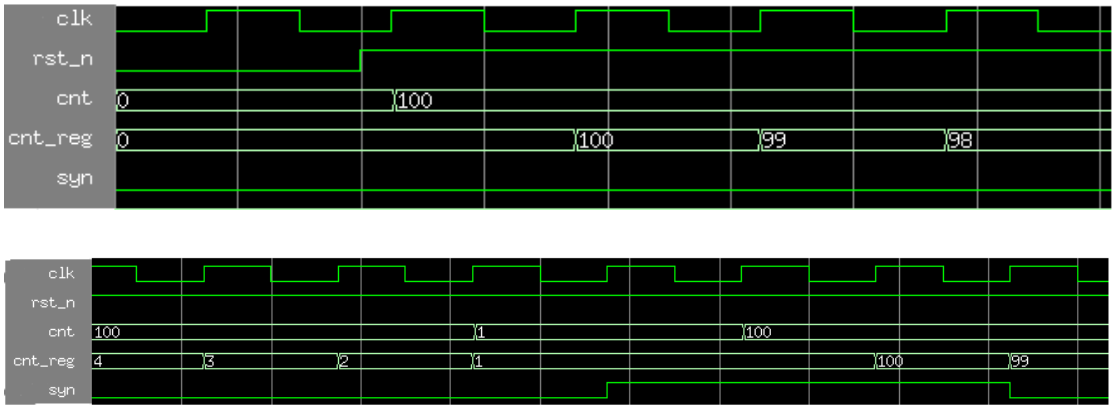


Figure 4.11: *Waveforms of Counter Control Unit*

4.4 Implementation Results

Figure 4.12 shows the implementation results for different values of clock cycle. The shortest clock cycle that could be met was this of *1.1 ns*. Also, ***Figure 4.13*** and ***Figure 4.14*** show the diagrams of *clock period – area* and *period – power*, respectively.

clock period (ns)	frequency (MHz)	critical path (ns)	area (μm²)	power (mW)
1,1	909	1,07	9956,016230	8,597
1,2	833	1,17	9141,048193	7,504
1,3	769	1,27	8827,761752	6,969
1,4	714	1,38	8575,156928	6,366
1,5	667	1,49	8630,899410	5,663
1,6	625	1,59	8412,868996	3,512

Figure 4.12: *Implementation results*

Obviously, in all cases critical paths exist in the part of *Multiplier*.

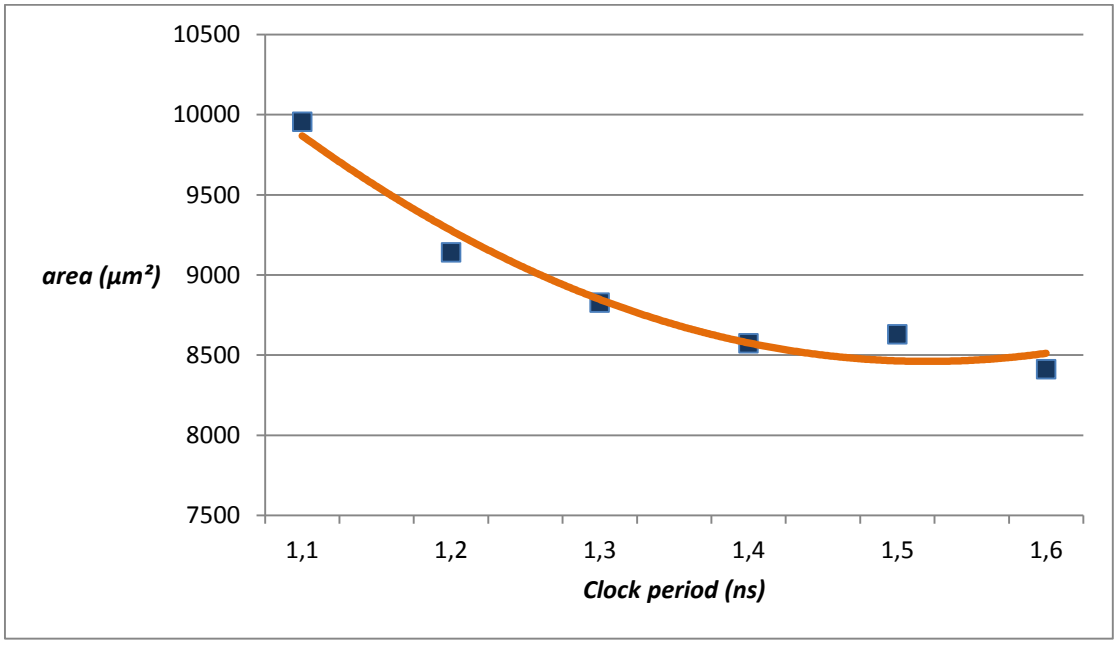


Figure 4.13: *Diagram of Clock Period – Area*

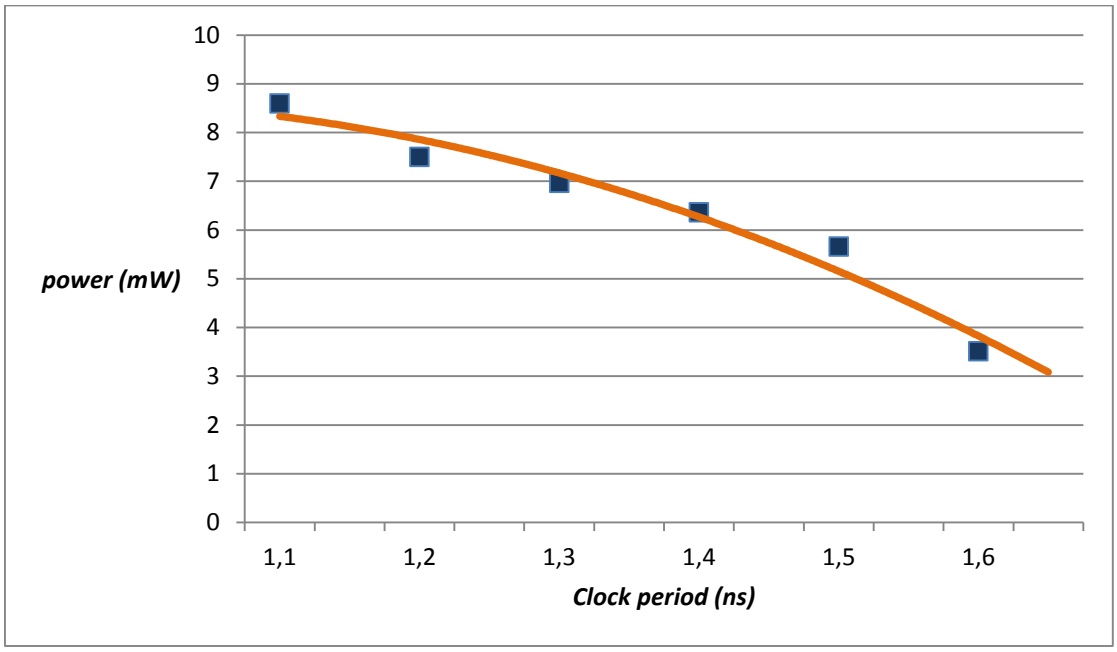


Figure 4.14: *Diagram of Clock Period – Power*

Furthermore, in order to measure the efficiency of using the *Final Adder* only for the extracting of the final result in order to save power, a slightly different circuit was implemented, which **has exactly the same behavior**, as far as the output pins are concerned, but it uses a multiplexer *after* the *Final Adder* and not before. As a result, the unit of the *Final Adder* is used in every cycle. ***Figure 4.15*** shows the block diagram of this different implementation of *SCAU* and ***Figure 4.16*** to ***Figure 4.18*** provide the corresponding information of the *Figure 4.12* to *Figure 4.14*.

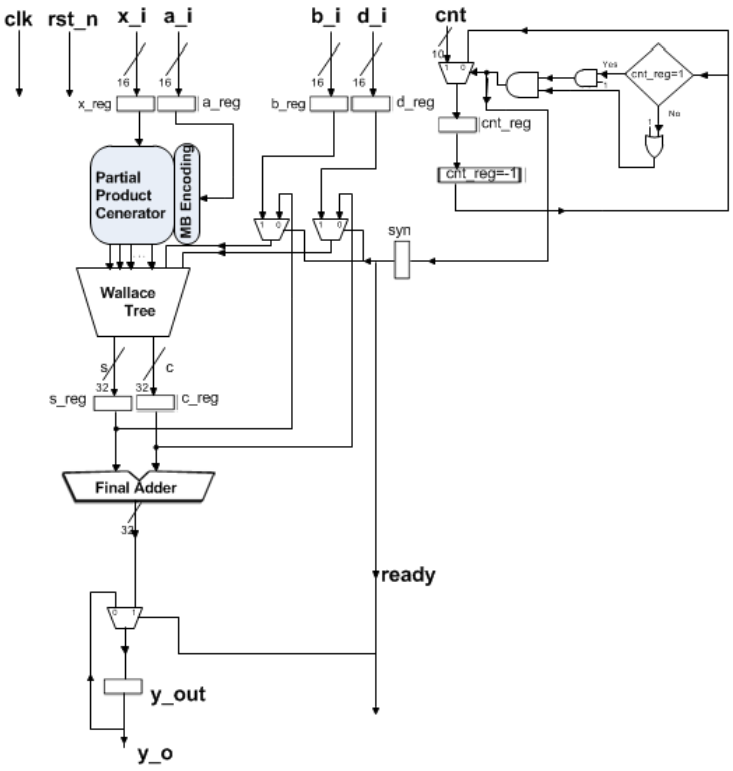


Figure 4.15: *A different implementation of SCAU*

clock period (ns)	frequency (MHz)	critical path (ns)	area (µm²)	power (mW)	area decrease (%)	power increase (%)
1,1	909	1,09	9859,349035	8,904	*0,971%*	*3,571%*
1,2	833	1,17	9094,478591	8,210	*0,509%*	*9,408%*
1,3	769	1,28	8793,892950	7,601	*0,384%*	*9,069%*
1,4	714	1,39	8531,409727	6,941	*0,510%*	*9,032%*
1,5	667	1,48	8570,923406	6,174	*0,695%*	*9,023%*
1,6	625	1,58	8371,944198	3,513	*0,486%*	*0,028%*

Figure 4.16: *Implementation results of the different implementation of SCAU and comparing with these of the normal implementation*

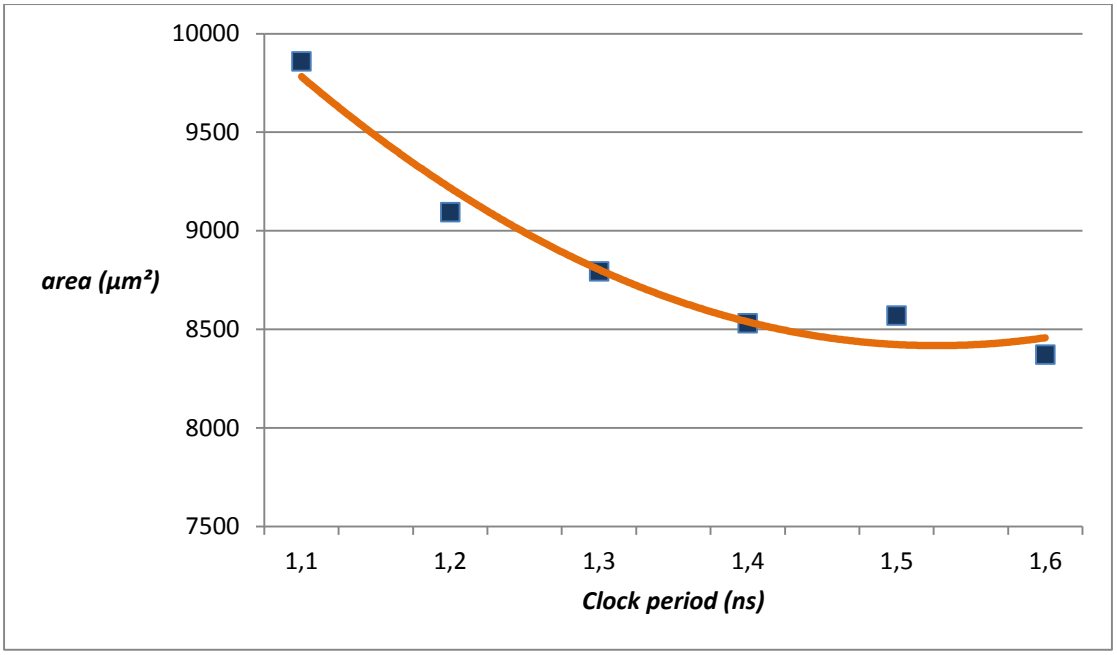


Figure 4.17: *Diagram of Clock Period – Area for the different implementation of SCAU*

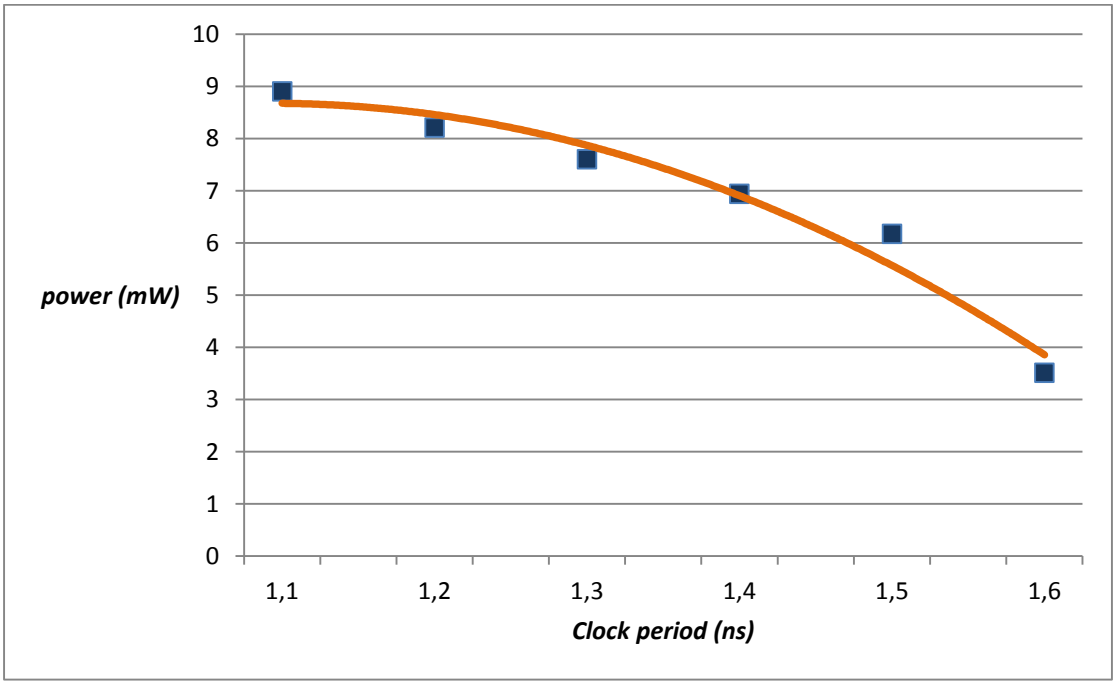


Figure 4.18: *Diagram of Clock Period – Power for the different implementation of SCAU*

From the implementation results shown in ***Figure 4.16***, the efficiency in power saving provided from the multiplexers of the normal version of SCAU can be noticed. Actually, with

these two multiplexers, SCAU takes advantage of its special feature of accumulation in carry-save representation.

5 Future work

Suggestions for future work could be made for all the different aspects of this diploma thesis. These aspects are:

- **Variety of Speech codecs**
  1. Studying of more speech codecs, in order to obtain a more general view, and, especially, of those applied in *VoIP* like *G.729a*, *iLBC* and *G.722.2* (this is not an open source speech codec).
  2. It would be very interesting to study the open-source multimode speech codecs *Opus* and *SILK* (speech codec of Skype).
- **Profiling method**
  1. Trying to apply a better, and less exposed to errors, profiling method. A good idea could be using a better C-processing of the list *operations* that keeps information for the operands.
  2. Applying *benchmarking* method not only for additions and multiplications, but keeping track also of MACs.
  3. Different arithmetic schemes should be tested for efficiency [e.g. *mult – mac* ($(a * b) * c + d$)].
- **SCAU**
  1. A more efficient implementation of the two operations of *MAC* and *MADD-ADD.*
  2. Test efficiency of SCAU for an operation of $a + b + c$ in comparison with other solutions (e.g. an adder of three inputs). In SCAU this operation could be implemented by performing a *MADD-ADD* mode in a way like: $x * 1 + b + d$.

Conclusion

In this diploma thesis an arithmetic unit for efficient hardware implementation of speech codecs was designed. The whole procedure that was followed, in order to fulfil this purpose, included a lot of different aspects of studying and the familiarization with a variety of tools.

Actually, it was a unique experience to use different parts of Electrical and Computer Engineering and come through all this way from digital signal processing and software implementations to the testing of different profiling methods and to end up in hardware design, synthesis and evaluation. It was a touch of how nowadays technology engineering develops all of these everyday applications, like *VoIP* conferences.

Obviously, researching never stops and there in nothing that can claim that it solved a problem. The more realistic view is to state that there was a try to find a better approach towards the problem or towards the covering of a need.

The same can be stated for this diploma thesis. It was an effort to find a better arithmetic unit for speech codecs and, obviously, there are a lot of aspects that need or can be improved (see *Future Work* section). Therefore, these improvements will be very valuable as they can lead to a better implementation of this great tool of human communication, the speech codecs.

List of Figures

List of Tables

References

[1] Wai C. Chu (2003), "*SPEECH CODING ALGORITHMS Foundation and Evolution of Standarized Coders".* WILEY-INTERSCIENCE, John Wiley & Sons, Inc., Hoboken, New Jersey.

[2] Lawrence R. Rabiner, Ronald W. Schafer (1978), "*Digital Processing of Speech Signals".* Ed. Alan V. Oppenheim. Prentice-Hall Signal Processing Series.

[3] K. Pekmestzi (2003), "*Ψηφιακά Συστήματα VLSI*". Athens

[4] A. Kanatas, F. Konstantinou, G. Pantos (2008), "*Συστήματα Κινητών Επικοινωνιών".* Athens: Papasotiriou Editions.

[5] ITU- T Recommendation G.191 (2010), "*Software tools for speech and audio coding standardization".*

[6] ITU- T Recommendation G.711 (1988), "*Pulse code modulation (PCM) of voice frequencies*". [Online], Available: http://www.itu.int/rec/T-REC-G.191/en [Accessed: Apr. 3, 2012]

[7] ITU- T Recommendation G.722 (1988), "*7 kHz audio-coding within 64 kbit/s*". [Online], Available: http://www.itu.int/rec/T-REC-G.722/en [Accessed: Apr. 3, 2012]

[8] ITU-T Recommendation G.723.1 (2006), "*Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s "*. [Online], Available: http://www.itu.int/rec/T-REC-G.723.1/en [Accessed: Apr. 3, 2012]

[9] ITU- T Recommendation G.726 (1990), "*40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM) "*. [Online], Available: http://www.itu.int/rec/T-REC-G.726/en [Accessed: Apr. 3, 2012]

[10] ITU- T Recommendation G.729 (2007), "*Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP) "*. [Online], Available: http://www.itu.int/rec/T-REC-G.729/en [Accessed: Apr. 3, 2012]

[11] Yao Wang (2006), "*Source Coding Basic and Speech Coding".* EE3414 Multimedia Communications System – I, Polytechnic University, Brooklyn, NY [Online]. Available: http://eeweb.poly.edu/~yao/EE3414/audio_coding.pdf [Accessed: Apr. 24, 2012]

[12] Cisco (2006), "*Understanding Delay in Packet Voice Networks*" [Online], Available: http://www.cisco.com/en/US/tech/tk652/tk698/technologies_white_paper09186a00800a8993.shtml [Accessed: Apr. 24, 2012]

[13] Cable Television Laboratories, Inc. (2009), "*PacketCable 2.0 Codec and Media Specification*", [Online]. Available: http://www.cablelabs.com/specifications/PKT-SP-CODEC-MEDIA-I07-090702.pdf [Accessed: Apr. 30, 2012]

[14] Nuntius, "*Wireline Communication Solutions"*, [Online], Available: http://www.nuntius.com/solutions11.html [Accessed: Apr. 30, 2012]

[15] Bishnu S. Atal & Nikil S. Jayant (1996), "Speech Coding", AT&T Bell Laboratories, Murray Hill, New Jersey, USA [Online], Available: http://www.cslu.ogi.edu/HLTsurvey/ch10node4.html [Accessed: Apr. 22, 2012]

[16] Nadeem Unuth, "*VoiP Codecs"*. [Online], Available: http://voip.about.com/od/voipbasics/a/voipcodecs.htm [Accessed: Apr. 20, 2012]

[17] Mathworks, Inc.,"Create MEX-Files". [Online], Available: http://www.mathworks.com/help/matlab/create-mex-files.html [Accessed: May. 15, 2012]

[18] VoIP Solutions.GR, "*VoiP Phones"*. [Online], Available: http://www.voipsolutions.gr/shop/index.php?category_id=230&target=categories [Accessed: Feb. 23, 2013]

[19] "*Opus Interactive Audio Codec"*, [Online], Available: http://opus-codec.org/ [Accessed: Feb. 25, 2013]

[20] "*Fax Over IP"*, [Online], Available: http://www.foip.org/ [Accessed: Feb. 25, 2013]

[21] Radio-Electronics.com, "*DECT Technology Tutorial"*, [Online], Available: http://www.radio-electronics.com/info/wireless/dect/dect_basics.php [Accessed: Feb. 25, 2013]

[22] VOCAL, "*Voice over IP (VoIP)"*, [Online], Available: http://www.vocal.com/voip-software/ [Accessed: Feb. 27, 2013]

[23] VoiceAge, "*Narrowband/Wideband Speech Comparison"*, [Online], Available: http://www.voiceage.com/listening_comparison.php [Accessed: Apr. 11, 2013]