TOWARDS A REAL-TIME IMPLEMENTATION OF LOUDNESS ENHANCEMENT
ALGORITHMS ON A MOTOROLA DSP 56600

By

ADNAN H. SABUWALA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002
ACKNOWLEDGMENTS

I would like to thank a myriad of persons without whose help this thesis would not have been possible. Of all these, I am immensely indebted to my professor and advisor, Dr. Harris, without whose help I would not be writing this thesis. He is an epitome of friendliness and generosity. He has been not only a mentor, but also has offered me a helping hand during the course of my graduate studies. Words are not enough to describe his emotional, financial and technical support and I would like to express my sincere gratitude towards him for the same.

I would like to express my sincere thanks to Dr. Principe and Dr. Rangarajan for agreeing to be on my thesis committee and providing me with helpful hints at every stage of my thesis.

I would also like to thank Mark, Bill and Kaustubh for their valuable hints. I would also like to thank Marc for providing me with a valuable research topic and for his thoughts and suggestions on the same.

And finally, I would like to thank my parents, Hatim and Fatema Sabuwala, for their unparalleled love and affection and for the belief they have shown in me.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ........................................... ii

ABSTRACT .................................................. v

CHAPTERS

1 INTRODUCTION .............................................. 1
  1.1 What Is a DSP? ........................................... 1
  1.2 Inside a Digital Cell Phone ............................... 2
  1.3 Loudness Enhancement Algorithms .......................... 4
    1.3.1 Critical Band Concept ............................. 5
    1.3.2 Warped Filter Implementation ......................... 7
    1.3.3 Vowels .......................................... 8
  1.4 Chapter Organization and Structure ......................... 9

2 DSP ARCHITECTURAL DETAILS ............................ 10
  2.1 Overview ............................................. 10
  2.2 Central Architecture .................................. 12
  2.3 Data Arithmetic Logic Unit ............................ 16
    2.3.1 Data ALU architecture ............................. 16
    2.3.2 Data ALU Registers ................................ 17
    2.3.3 MAC Unit ....................................... 18
    2.3.4 Data ALU Accumulator Registers ...................... 19
    2.3.5 Accumulator Shifter ................................ 19
    2.3.6 Bit Field Unit ................................... 20
    2.3.7 Data Shifter/Limiter ................................ 20
    2.3.8 Data ALU Arithmetic ................................ 22
  2.4 Address Generation Unit ................................ 23
  2.5 Program Control Unit .................................. 25
  2.6 Program Patch Logic ................................... 26
  2.7 PLL and Clock Oscillator ................................ 26
  2.8 Expansion Port (Port A) ................................ 27
  2.9 JTAG Test Access Port and On-Chip Emulator (OnCE) ......... 27
  2.10 On-Chip Memory ...................................... 27
  2.11 Peripherals .......................................... 28
  2.12 Summary ............................................. 28
3 BUILDING BLOCKS AND IMPLEMENTATION ISSUES ................. 29
  3.1 Basic Block Diagram .................................. 29
    3.1.1 Introduction to Linear Prediction .................. 29
    3.1.2 Bandwidth Expansion ............................... 32
  3.2 Autocorrelation ...................................... 33
  3.3 Levinson-Durbin Recursion ............................... 34
  3.4 FIR and IIR Filters ................................... 36
  3.5 Scaling FIR Coefficients ................................ 37
  3.6 Warped Filter Implementation ............................. 41

4 LMS: THE SOLUTION TO IMPLEMENTATION ISSUES .................. 45
  4.1 The Solution ......................................... 45
  4.2 Least Mean Squares (LMS) Algorithm ..................... 46
  4.3 Linear Prediction Using LMS ............................ 49
  4.4 Experimental Results .................................. 49

5 CONCLUSIONS AND FUTURE WORK .............................. 58

APPENDICES
A ASSEMBLY CODE FOR LEVINSON-DURBIN ......................... 60
B ASSEMBLY CODE FOR IIR AND FIR FILTERS ..................... 65
C ASSEMBLY CODE FOR LMS ALGORITHM .......................... 69
D ASSEMBLY CODE FOR AUTOCORRELATION ......................... 75
E ASSEMBLY CODE FOR MODIFIED SIGNED LMS ALGORITHM ........ 79
REFERENCES .................................................. 86
BIOGRAPHICAL SKETCH ........................................ 87
Towards a Real-Time Implementation of Loudness Enhancement Algorithms on a Motorola DSP 56600

By

Adnan H. Sabuwala

December 2002

Chairman: John G. Harris
Major Department: Electrical and Computer Engineering

Most of the cellular phone companies with audio speaker capabilities focus on reducing the current drain to extend battery life. None of these companies concentrate on modifying the speech signal itself to make it sound louder in noisy listener environments without adding additional energy. Such algorithms have been described in literature by Boillot and form the backbone of this thesis. The current project focusses on taking a step towards running these algorithms in real-time on a 16-bit fixed point Motorola DSP 56600. Implementation of the autocorrelation, Levinson-Durbin, FIR, and IIR filters in assembly for the Motorola DSP 56600 has been investigated in the thesis. The challenges and alternate solutions to circumvent the challenges have been described, and experimental results have been presented. Results indicate that the modified signed LMS algorithm, which can be considered to be a blend between the LMS and signed LMS algorithms, turns out to be an elegant solution to circumvent the challenges in implementing the Levinson-Durbin recursion.
CHAPTER 1
INTRODUCTION

In this thesis we provide a real-time implementation of algorithms that increase the perceived loudness of a cellular phone in noisy listener environments. The work presented in this thesis is a step towards achieving a real-time working product based on the loudness enhancement algorithms which have been earlier developed and simulated in MATLAB by Marc Boillot [2]. This algorithm widens the bandwidth of the formant of voiced phonemes in speech. This work, which has been funded by Motorola Inc., involves the use of a DSP simulator to simulate the algorithms on a MOTDSP56600 which is a ROM-based 16-bit fixed point CMOS Digital Signal Processor (DSP).

This chapter briefly describes the basic structure of a digital cell phone and some basics about digital signal processors (DSPs). Later in the chapter we discuss the loudness enhancement algorithms and present the structure of the remaining chapters. The loudness algorithms present the motivation behind this thesis.

1.1 What Is a DSP?

A digital signal processor (DSP) can be classified as a variant of a microprocessor—one that is fast and powerful enough to process data in real time. The real-time capability of the DSP makes it ideal for applications where delays must be minimized.

Vocoders attempt to achieve high data compression while simultaneously maintaining the signal quality. Most vocoders use psychoacoustic criteria incorporated in their bit compression schemes. Bandwidth expansion techniques have been used in vocoders to alleviate quantization noise and residual effects of the vocoding process. Two of the typically defined internal vocoder filtering operations which use this technique are the perceptual noise spectral shaping and the adaptive post-filtering. The Code Excited Linear Prediction (CELP) algorithm has been shown
to achieve high quality speech coding at low bit rates [4]. The perceptual noise weighting filter effectively generates a noise error function which retains the formant pole locations and elevates the allowable noise in tonal regions. Applied to the excitation signal it alters the flat spectrum of the excitation to that of human hearing sensitivity. The adaptive post filtering operation typically attempts to suppress quantization noise in the valley regions by amplification of the less sensitive formant regions. It consists of four basic filtering steps: long-term filtering, short-term filtering, tilt compensation, and adaptive gain control. Of these, the short-term filter is used to improve the overall quality of the synthesized speech and to alleviate quantization noise effects.

Older analog cellular phones usually suffered from poor speech quality and annoying echoes. However, in newer digital cell phones, the DSP takes a real-world signal, like speech, and performs mathematical computations on it to improve the sound (referred to as “speech enhancement”). The DSP compresses the data (one’s voice), removes the background noise (referred to as “noise cancellation”) and eliminates the echoes (referred to as “echo cancellation”) so that one’s voice travels at a faster rate. The result is a clear sound, with no annoying echoes.

One task that a DSP does is to take a digital signal and process it to improve the signal. The improvement is in the form of clearer speech, sharper images, or faster data. This ability to process signals without requiring additional energy can make new breakthroughs in cellular phone technology where a longer battery life is one of the prime concerns of the consumer.

In the next section we look at the internal structure of a digital cellular phone and try to explain the functionalities of each of the components that make up the phone.

1.2 Inside a Digital Cell Phone

Considering some of the most intricate devices based on a “complexity per cubic inch” scale, cell phones are one of the most such devices that people handle on a daily basis. Modern digital cell phones can process millions of calculations per second in order to compress and decompress the voice stream. They can transmit and receive on hundreds of channels, switching channels in sync with the base stations as the phone moves between cells.
The following individual parts can be clearly identified on taking a cell phone apart:

1. A miniature microphone

2. A speaker

3. An LCD or plasma display

4. A keyboard similar to the one on a TV remote control

5. An antenna

6. A battery

7. A circuit board containing the guts of the phone

The circuit board is the heart of the phone system. Figure 1.1 shows one of such circuit boards from a typical digital phone.

Figure 1.1 identifies the several components of a digital cellular phone. On the left, we see the Analog-to-Digital and Digital-to-Analog conversion chips. We also see a “Digital Signal Processor” chip which is a highly customizable processor designed to perform massive signal manipulations at fast speeds. The DSP chip used for the current research is a maximum 60 MIPS (Million
Instructions Per Second) chip and the vocoder handles all the compression and decompression of speech signals. The microprocessor and memory units handle the housekeeping chores for the keyboard and display, deal with command and control signalling with the base station and also coordinate the rest of the functions on the board. The RF and power section handles issues of power management and recharging and also deals with the hundreds of channels. And finally, the RF transmitter and receiver amplifiers handle the signals coming in and out of the antenna.

In the following section, we present and describe the “loudness enhancement algorithms” that form the backbone of this thesis report. In this section, a few technical terms which might be of interest in the remainder of this thesis have been described.

1.3 Loudness Enhancement Algorithms

There is a vast world market for small hand-held devices and cellular phones. The major concern of any such manufacturing company is that of designing low-cost devices with limited power consumption. Battery life can be considerably increased by saving on power consumption. However, a majority of these companies focus on building devices with better speaker design and also using more efficient power amplifiers which can reduce the current drain and as such increase the battery life considerably. None of these companies have tried to address energy conservation schemes which operate directly on the speech signal which serves as the input and output to the modern digital cellular phones. Therefore, an important step towards power savings can be addressing this issue and focussing towards designing algorithms which operate directly on the speech signal and try to increase loudness perception over a cellular phone without actually increasing the signal energy. This section describes the basis of such loudness enhancement algorithms and also presents the development of these algorithms and defines basic terminology associated with them. As we shall see in the following sections, these algorithms exploit the psychoacoustic nature of the human auditory system to achieve loudness enhancement and a novel warped filter implementation of the same is developed. Its critical that these algorithms be implementable in real-time so that a full working product can be made realizable.
1.3.1 Critical Band Concept

In this section, we present the basis of the “loudness enhancement algorithms.” A brief overview of the critical band concept and its significance towards loudness enhancement is presented.

Loudness

Loudness can be defined as the human perception of the intensity of the speech signal. Loudness is a function of the sound intensity and so also the frequency and quality of the speech signal. Loudness can evaluated based on the ISO-532B standard which is a graphical evaluation procedure for calculating the loudness of a complex sound. Loudness models have also been developed in the literature, first by Zwicker and then further improved by Moore and Glasberg. Moore’s and Zwicker’s models are very similar for moderate and normal sound levels. However, at lower frequencies and at sounds close to quiet level, Moore’s model outperforms Zwicker’s model. These algorithms which have been developed are based on Moore’s model which uses excitation patterns obtained from auditory filters.

The loudness level was introduced as a mechanism for loudness measurement procedure. By definition, loudness level is the pressure level of the sound of a 1-KHz tone which is equally loud as the sound being tested. The loudness level is measured with a unit called the “phon.” Sounds with equal phon levels are at equal loudness and a contour of such curves are known as the equal loudness curves and are shown in Figure 1.2.

The phon, however, does not provide a measure of the loudness scale and hence, another unit called the “sone” was introduced. A sone value of 1 corresponds to the loudness exhibited by a 1-KHz tone at an intensity of 40dB sound pressure level. A 10 phon increase is approximately equivalent to a doubling of the sone value.

Critical Bands

The critical band is an extremely important concept in auditory theory. A critical band can be regarded as the bandwidth in which sudden perceptual changes can be noticed [5]. Critical
bands are independent processing channels which work on the spectral components of signals falling under the same channel. Fletcher performed experiments on masking phenomena and his results show that audibility is affected only by the amount of noise inside a critical band. Since noise outside a certain bandwidth does not affect the tone detection threshold, it suggested the existence of an auditory filter. The region beyond which the noise does not affect detection thresholds is called the **critical bandwidth**. Experiments have shown that the critical bandwidth increases with increasing frequency [14].

The critical band concept is an important concept for hearing sensations, especially loudness. For a fixed intensity sound, loudness remains constant as long as the bandwidth does not increase beyond the critical bandwidth. However, once the critical bandwidth is exceeded, the loudness perception increases. This increase in loudness takes place even when the energy of the speech signal remains constant.
In view of this the critical band concept forms the basis of the loudness enhancement algorithm. The overall loudness of a speech signal is obtained by summing up the loudness over each of the critical bands.

1.3.2 Warped Filter Implementation

In this section, we outline the implementation of a warped filter structure which was used in achieving loudness enhancement of the input speech signal without actually increasing its energy. However, we reserve the majority of the development of this structure for future chapters outlined in this thesis.

Loudness enhancement involves increasing the bandwidth beyond a critical bandwidth as indicated in the previous section. However, to achieve this bandwidth expansion without actually changing the formant locations of the spectral envelope of speech, we had to make use of the DSP fundamentals of evaluating the spectral response over a circle with radius larger than 1. We know from DSP that when the impulse response of a stable system is evaluated over a circle having a radius larger than 1, then the resulting system is not only stable, but also since the poles get pulled farther apart from the circle boundary, the formant locations get bandwidth expanded. This type of evaluation over a circle with radius larger than unity corresponds to an equivalent power scaling of the coefficients of the system with a radius term. This provides a fixed bandwidth increase independent of formant frequency. However, we know that the critical bandwidth increases with frequency and as such would like to achieve a non-linear expansion of bandwidth with frequency to account for the same. This non-linear expansion is achieved by the use of a warped filter structure. The details of this implementation shall be reserved for the future chapters.

It should be noted, that in Boillot [2], the loudness enhancement algorithms which were presented and developed in MATLAB worked only on the “voiced” sections of speech. A brief section on vowels is as such presented next to illustrate the significance of working with voiced speech for these algorithms.
1.3.3 Vowels

Vowels can be characterized typically using the first three formant locations. The important acoustic cues associated with vowels are the formant frequency locations, their bandwidths, amplitude and duration. The formant hypothesis (resulting from the classic study of Peterson and Barney, as cited in Hillenbrand et al. [7]) states that the first two to three formant locations provide for vowel discrimination while the second and third formants help in discerning vowel intelligibility. Thus, in essence, vowel formant locations are an acoustic cue in vowel perception. Furthermore, studies have shown that the change in formant locations can considerably affect the phonetic quality of vowels, however, the change in formant bandwidths or spectral tilt will not affect phonetic quality [1]. This is a useful feature of vowel characteristics, since in the current project we are interested in altering the formant bandwidths and not the formant locations to improve the loudness of the speech signal.

Vowels are typically spectrally smooth and have a high energy. The majority of the signal power $\approx 80\%$ is contained in them and a vast majority of it is unmasked. Also, as indicated in the previous section, the alteration of formant bandwidths does not degrade vowel identification and intelligibility. Loudness analysis indicates that the peak loudness is produced by vowels in speech [14]. Moreover, the intelligibility of speech is determined by the vowel-consonant-vowel transitions rather than the steady state region of vowels. These observations suggest that a loudness enhancement scheme which preserves energy would best work on vowel sections of speech. In view of these observations, a bandwidth expansion technique which increases the bandwidth of vowel regions of speech moderately should lead to an increase in loudness perception without actually degrading the signal. Such a technique is called formant expansion. Results have shown that increasing the bandwidth on a linear scale will increase loudness [2].

Thus, in this section we saw that the vowels have the majority of the speech signal power, are in abundance in any typical sentence, have smooth spectral shapes, have broad bandwidths which increase with increasing frequency and finally can be sufficiently bandwidth expanded
without degrading the intelligibility. Therefore, they form ideal candidates on which the loudness enhancement techniques can be performed.

1.4 Chapter Organization and Structure

In this section, we present the chapter structure and a brief description of each chapter in the remainder of this thesis.

In chapter 2, we shall discuss the architecture of the Motorola DSP 56600 which has been used for implementation of the developed algorithms in real-time. The chapter shall discuss the details of the processor and also provide some applications of the processor. In chapter 3, the basic building blocks for the implementation of the loudness enhancement algorithms are described. These include the autocorrelation, LPC, FIR and IIR filters respectively and shall be discussed in greater detail. We will also talk about the warped version implementation in this chapter. We will present the difficulties and challenges encountered in the implementation of the algorithm in real-time and also provide a basic description of the FIR scaling from a binary mathematical point of view. In chapter 4, we will present an alternate method for circumventing the challenges we encountered in implementation of the loudness enhancement algorithms in real-time. Experimental results are also described in the chapter and the total time taken for the algorithms to run in real world is tabulated. Chapter 5 shall be the final chapter of the current thesis and shall bring out the conclusions of the experimental results. It shall focus on providing a step towards future work that can be done to make this product completely realizable in real-time.
CHAPTER 2
DSP ARCHITECTURAL DETAILS

In this chapter, we will discuss the architectural details and programmable modes of the 16-bit DSP 56600 which has been used for the implementation of the loudness enhancement algorithms. The chapter begins with a small overview of the DSP followed by the various architectural components of the DSP chip.

2.1 Overview

The current project involved the use of the DSP 56600 family chip. The DSP56600 family of 16-bit high performance Digital Signal Processors (DSPs) is designed specifically for low-power digital handset cellular applications. These chips are capable of performing a wide variety of fixed-point DSP algorithms. Each DSP in the family architecture contains a central processing module which is common to various other family members. Besides this, a variety of other highly integrated and cost-effective DSP devices can be built around this core based upon a library of modules containing memories and peripherals. The main advantage of this DSP is that it can provide very high execution speeds in a real-time, Input/Output (I/O) intensive environment which most of the state-of-the-art DSP applications require.

Digital Signal Processing is the arithmetic processing of real-time signals which have been sampled at regular intervals and digitized. Examples of such type of processing includes the following:

- Filtering signals
- Convolution
- Correlation–comparing two signals
- Rectifying, amplifying and/or transforming a signal
All of the above functions have traditionally been performed using analog circuits. With recent developments in the semiconductor industry, it has been possible to obtain the processing power necessary to perform these and other functions using DSPs. Figure 2.1 shows an example of analog signal processing. The circuit in the diagram shows a filter implementation for controlling an actuator. Since an ideal filter is impossible to design, an engineer has to design it for an acceptable response considering temperature variations, component aging, fluctuations in power supply, and component accuracy. The resultant circuit has low noise immunity, requires adjustments and is difficult to modify.

The equivalent circuit using a DSP is shown in Figure 2.2. The application requires the use of an Analog-to-Digital (A/D) and Digital-to-Analog (D/A) converter in addition to the DSP. However, even with these additional parts, the total component count can be much lower using a DSP than the analog counterpart. This is mainly due to the high integration of components available with the use of a DSP.

In summary, the advantages of using DSPs as compared to analog-only circuits, include the following:
In the following sections, we describe the architecture of the Motorola DSP 56600 and also detail each of the components in the architecture.

2.2 Central Architecture

This section describes the DSP 56600 core, a member of Motorola’s family of programmable CMOS DSPs. Low power dissipation, low cost, high performance and high integration are the design priorities for this DSP core. Some of the major core features are the following [12]:

- 60 Million Instructions Per Second (MIPS) with a 60-MHz clock at 2.7V
- Fully pipelined 16×16-bit parallel Multiplier-Accumulator (MAC)
- 40-bit parallel barrel shifter
- Highly parallel instruction set
• Position Independent Code (PIC) support

• Unique DSP addressing modes

• Nested hardware DO loops

• Fast auto-return interrupts

• On-chip 16-stage hardware stack with stack extension

• On-chip support for software patching and enhancements

• On-chip PLL

• On-Chip Emulation (OnCE) module

• Address tracing for debugging

• JTAG port compatible with the IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1)

Low-power features of the DSP 56600 core include the following:

• Very low power CMOS design (<0.7mA/MIP, 2.7V @70 MIPS and <0.5mA/MIP, 1.8V @60 MIPS)

• Low power Wait standby mode

• Ultra-low power Stop mode

• Power management units for further power reduction

• Fully static logic, with operation frequency down to DC

The DSP core provides the following functional blocks:

• Data Arithmetic Logic Unit (Data ALU)
• Address Generation Unit (AGU)

• Program Control Unit (PCU)

• Program Patch Logic

• PLL and Clock Oscillator

• Expansion Port (Port A)

• JTAG Test Access Port and On-Chip Emulation (OnCE) module

• Memory

Besides this, each member of the DSP 56600 family provides its own set of on-chip peripherals for enhanced functionality. The following buses have been implemented for providing data exchange between the blocks of the DSP core:

• Peripheral I/O Expansion Bus (PIO_{EB}) to peripherals

• Program Memory Expansion Bus (PM_{EB}) to Program ROM

• X Memory Expansion Bus (XM_{EB}) to X Memory

• Y Memory Expansion Bus (YM_{EB}) to Y Memory

• Global Data Bus (GDB) between Program Control Unit and other core structures

• Program Data Bus (PDB) for carrying program data throughout the core

• X Memory Data Bus (XDB) for carrying X data throughout the core

• Y Memory Data Bus (YDB) for carrying Y data throughout the core

• Program Address Bus (PAB) for carrying program memory addresses throughout the core

• X Memory Address Bus (XAB) for carrying X memory addresses throughout the core
- Y Memory Address Bus (YAB) for carrying Y memory addresses throughout the core

Excepting the Program Data Bus (PDB), all internal buses on the DSP 56600 core are 16-bit buses. The PDB is a 24-bit bus. The block diagram of the DSP 56603 which is a member of the DSP 56600 family of DSPs is shown in Figure 2.3. It illustrates the core blocks of the DSP 56600 and also shows representative peripherals for the chip implementation.

![Figure 2.3: DSP 56603 Block Diagram](image)

In the following sections, we describe each of the functional blocks of the DSP 56600 core. A brief description of blocks that are not relevant to this project is also provided.
2.3 Data Arithmetic Logic Unit

This section presents the operation and architecture of the Data ALU, which is the heart of the arithmetic and logical operations of the DSP core. In addition, it also presents the arithmetic and rounding performed by the Data ALU.

2.3.1 Data ALU architecture

The Data ALU is primarily responsible for performing the arithmetic and logical operations on data operands in the DSP core. The Data ALU registers can be read over the X Data Bus (XDB) and the Y Data Bus (YDB) either as 16-bit or 32-bit operands. The source operands are always the Data ALU registers themselves and can be either 16, 32, or 40 bits. The results are stored in an accumulator. The operations are performed in 2 clock cycles in a pipeline fashion so that a new instruction can be initiated in every clock thereby yielding an effective execution rate of 1 clock cycle per instruction. Also another feature is that the destination register can be used as a source for the next instruction without any conflicts. The major components of the Data ALU which is shown in Figure 2.4 are as follows:

- Four 16-bit input registers
- A parallel, fully pipelined MAC
- Two 32-bit accumulator registers
- Two 8-bit accumulator extension registers
- A Bit Field Unit (BFU) with a 40-bit barrel shifter
- An accumulator shifter
- Two data bus shifter/limiter circuits
2.3.2 Data ALU Registers

X1, X0, Y1 and Y0 are four 16-bit general purpose data registers. These registers can either be viewed as four separate 16-bit registers or two 32-bit registers formed by the concatenation of X1:X0 and Y1:Y0, respectively. X1 is the most significant word in X and similarly Y1 is the most significant word in Y. As can be seen in Figure 2.4 these registers serve as input buffers between the XDB or YDB and the MAC unit or barrel shifter. These registers are used as Data ALU source operands allowing new operands to be loaded for the next instruction while the register
contents are used by the current instruction. Besides this, they can also be used to read back out onto the XDB or YDB.

2.3.3 MAC Unit

The heart of the arithmetic processing unit of the DSP 56600 core is the “Multiplier Accumulator (MAC).” It performs all the calculations on data operands. In the case of arithmetic operations, it accepts as many as 3 inputs and outputs one 40-bit result of the form Extension:Most Significant Product:Least Significant Product (EXT:MSP:LSP). The MAC unit operates independent of and in parallel with the XDB and YDB activity. It executes 16-bit ×16-bit, parallel, fractional multiplies, between two’s complement, signed, unsigned or mixed operands. The 32-bit product is right justified and added to the 40-bit contents of either the A or B accumulator.

The resultant 40-bit sum is stored back in the same accumulator. The MAC operation is fully pipelined and takes 2 clock cycles to complete. In the first clock cycle, the multiply operation is performed and the product is stored in the pipeline register. In the second clock cycle, the accumulator is added or subtracted. In the case of a pure multiply operation (MPY) being specified, the MAC clears the contents of the accumulator and adds the content of the product to it thereafter during the second clock cycle. A 40-bit result can also be stored as a 16-bit operand. In such a case, the LSP can either be truncated or rounded into the MSP. Rounding is performed if specified in the DSP instruction (e.g MACR). The rounding can be either convergent rounding (round-to-nearest-even) or two’s complement rounding. The type of rounding is specified by the Rounding Mode bit (RM) in the Status Register (SR). The bit in the accumulator that is rounded is specified by the Scaling Mode bits (S0 and S1) in the SR. It is possible to saturate the arithmetic unit’s result going into the accumulator so that we can fit it in 32 bits (MSP:LSP). This process is called “saturation.” It is activated by the Arithmetic Saturation Mode (SM) bit in the SR. This type of mode is typically used for algorithms which cannot take advantage of the Extension Accumulator (EXT).
2.3.4 Data ALU Accumulator Registers

There are six Data ALU registers viz. A2, A1, A0, B2, B1 and B0. Taken together they form two general purpose 40-bit accumulators A and B, with each one of them having three concatenated registers, A2:A1:A0 and B2:B1:B0, respectively. A1 or B1 stores the 16-bit MSP, A0 or B0 stores the 16-bit LSP while the 8-bit EXT is stored in A2 or B2.

Reading the A or B accumulators over the XDB or YDB buses is protected against overflow by substituting a limiting constant for the data that is being transferred. The content of A or B is not affected if limiting occurs. Only the value that is transferred over the XDB or YDB is limited. This process is commonly referred to as transfer saturation and is different from the Arithmetic Saturation mode that was described in Section 2.3.3.

The overflow protection is performed after the contents of the accumulator have been shifted according to the scaling mode. Shifting and limiting are performed only when the entire 40-bit accumulator is specified as the source for a parallel data move over the XDB or YDB. Shifting and limiting are not used when only an individual register within an accumulator (A1, A0, A2, B1, B0 or B2) is specified ad the source for a parallel data move. The A and B accumulators serve as buffer registers between the Arithmetic Unit and the XDB or YDB buses. These registers can be used as both Data ALU source and destination operands.

2.3.5 Accumulator Shifter

The accumulator shifter is an asynchronous parallel shifter with a 40-bit input and a 40-bit output that is implemented immediately before the MAC accumulator input. The source accumulator shifting operations are:

- No shift (Unmodified)
- 16-bit Right Shift (Arithmetic) for DMAC
- Force to zero
2.3.6  **Bit Field Unit**

The Bit Field Unit (BFU) contains a 40-bit parallel bidirectional shifter with a 40-bit input and a 40-bit output mask generation unit, and logic unit. The BFU is used in the following operations:

- Multibit Left Shift (Arithmetic or Logical) for ASL, LSL
- Multibit Right Shift (Arithmetic or Logical) for ASR, LSR
- 1-bit Rotate (Right or Left) for ROR, ROL
- Bit Field Merge, Insert, and Extract for MERGE, INSERT, EXTRACT, and EXTRACTU
- Count Leading Bits for CLB
- Fast Normalization for NORMF
- Logical operations for AND, OR, EOR, and NOT

2.3.7  **Data Shifter/Limiter**

The data shifter/limiter circuits provide special post-processing on data read from the A and B accumulators out to the XDB or YDB buses. There are two independent shifter/limiter circuits, one for the XDB bus and the other for the YDB bus. Each consists of a shifter followed by a limiter circuit.

**Scaling**

The data shifters in the shifters/limiters unit can perform the following data shift operations:

- Scale up—shift data one bit to the left
- Scale down—shift data one bit to the right
- No scaling—pass the data unshifted
Each data shifter has a 16-bit output with overflow indication. These shifters permit dynamic scaling of fixed-point data without modifying the program code. The data shifters are controlled using the Scaling Mode bits (S0 and S1) in the SR.

**Limiting**

In the DSP 56600 core, the Data ALU accumulators A and B have eight extension bits. Limiting occurs when the extension bits are in use and either A or B is the source being read over the XDB or YDB. The limiters in the DSP 56600 core place a shifted and limited value on XDB or YDB without changing the contents of the A or B registers. Having two limiters allows two-word operands to be limited independently in the same instruction cycle. The two data limiters can also be combined to form one 32-bit data limiter for long-word operands.

If the contents of the selected source accumulator can be represented without overflow in the destination operand size (i.e., the signed integer portion of the accumulator is not in use), the data limiter is disabled, and the operand is not modified. However, if the contents of the selected source accumulator cannot be represented without overflow in the destination operand size, the data limiter substitutes a limited data value having maximum magnitude (saturated) and having the same sign as the source accumulator contents:

- \$7FFF for 16-bit positive numbers
- \$7FFF FFFF for 32-bit positive numbers
- \$8000 for 16-bit negative numbers
- \$8000 0000 for 32-bit negative numbers

This process is called transfer saturation. The value in the accumulator register is not shifted or limited and can be reused within the Data ALU. When limiting does occur, a flag is set and latched in the SR.
### 2.3.8 Data ALU Arithmetic

The DSP 56600 core uses a fractional data representation for all Data ALU operations. The decimal points are all aligned and are left-justified.

The most negative number that can be represented is -1.0. The internal representation is $8000$ for words and $8000 0000$ for long-words. The most positive word is $87FF$ or $1 - 2^{-15}$ and the most positive long word is $87FF FFFF$ or $1 - 2^{-31}$. These limitations apply to all data stored in memory and to data stored in the Data ALU input buffer registers. The extension registers associated with the accumulators allow for word growth so that the most positive number with word growth that can be used is 256 and the most negative number with word growth is -256.

To maintain alignment of the binary point, when a word operand is written to accumulator A or B, the operand is written to the most significant accumulator register (A1 or B1), and its MSB is automatically sign extended through the accumulation extension register (A2 or B2). The least significant accumulator register (A0 or B0) is automatically cleared. When a long-word operand is written to an accumulator, the least significant word of the operand is written to the least significant accumulator register. The number representation for integers is between $2^{(N-1)}$. The fractional representation is limited to numbers between ±1. To convert from an integer to a fractional number, the integer must be multiplied by a scaling factor so that the result will always be between ±1. The representation of integer and fractional numbers is the same if the numbers are added or subtracted, but is different when the numbers are multiplied or divided. The key difference is in the alignment of the $2N - 1$ bit product. In fractional multiplication, the $2N - 1$ significant product bits should be left-aligned, and a 0 filled in the LSB to maintain fractional representation. In integer multiplication, the $2N - 1$ significant product bits should be right-aligned, and the sign bit duplicated to maintain integer representation. Since the DSP 56600 core incorporates a fractional array multiplier, it always aligns the $2N - 1$ significant product bits to the left. Besides these, the DSP 56600 core uses two types of rounding modes.
viz. “convergent-rounding” and “two’s-complement rounding.” The type of rounding is selected by the Rounding Mode (RM) bit in the SR.

2.4 Address Generation Unit

The Address Generation Unit (AGU) performs the effective address calculations using integer arithmetic necessary to address the data operands in memory and contains the registers used to generate these addresses. It implements four types of arithmetic: linear, modulo, multiple wrap-around modulo, and reverse-carry. The AGU operates in parallel with other chip-resources to minimize address-generation overhead.

The AGU is divided into two halves, each with its own Address Arithmetic Logic Unit (Address ALU). Each Address ALU has four sets of register triplets, and each register triplet is composed of an address register, an offset register, and a modifier register. The two Address ALUs are identical. Each contains a 16-bit full adder (called an offset adder).

A second full adder (called a modulo adder) adds the result of the first full adder to a modulo value that is stored in its respective modifier register. A third full adder (called a reverse-carry adder) is also provided. The offset adder and reverse-carry adder are in parallel and share common inputs. The only difference between them is that they carry propagates in opposite directions. Test logic determines which of the three summed results of the full adders is the output.

Each Address ALU can update one address register from its respective address register file during one instruction cycle. The contents of the associated modifier register specifies the type of arithmetic to be used in the address register update calculation. The modifier value is decoded in the Address ALU.

Since the modulo-addressing modifier type has been used in the current project, a brief description of the same is provided below.
**Modulo Modifier**

In this type of modifier arithmetic mode, address modification is performed modulo M, where M ranges from 2 to +32,768. Modulo M arithmetic causes the address register value to remain within an address range of size M, defined by a lower and upper address boundary.

The value $m = M - 1$ is stored in the modifier register. The lower boundary (base address) value must have zeros in the k LSBs, where $2^k \geq M$, and therefore must be a multiple of $2^k$. The upper boundary is the lower boundary plus the modulo size minus one (base address+$M - 1$). Since $M \leq 2^k$, once M is chosen, a sequential series of memory blocks, each of length $2^k$, is created where these circular buffers can be located. If $M < 2^k$, there is a space of $2^k - M$ between sequential buffers.

The address pointer is not required to start at the lower address boundary or to end on the upper address boundary; it can initially point anywhere within the defined modulo address range. Neither the lower nor the upper boundary of the modulo region is stored; only the size of the modulo region is stored in Mn. The boundaries are determined by the contents of Rn. Assuming the (Rn)+ indirect addressing mode is used, if the address register pointer increments past the upper boundary of the buffer (base address+$M - 1$), it wraps around through the base address (lower boundary). Alternatively, assuming that the (Rn)- addressing mode is used, if the address decrements past the lower boundary (base address), it wraps around through the base address+$M - 1$ (upper boundary).

If an offset, Nn, is used in the address calculations, the 16-bit absolute value, $|Nn|$, must be less than or equal to M for proper modulo addressing. If $Nn > M$, the result is data dependent and unpredictable, except for the special case where $Nn = P \times 2^k$, a multiple of the block size where P is a positive integer. For this special case, when using the (Rn)+Nn addressing mode, the pointer, Rn, jumps linearly to the same relative address in a new buffer, which is P blocks forward in memory. Similarly, for (Rn)−Nn, the pointer jumps P blocks backward in memory.

This technique is useful in sequentially processing multiple tables or N-dimensional arrays. The range of values for Nn is −32,768 to +32,767. The modulo arithmetic unit automatically
wraps around the address pointer by the required amount. This type of address modification is useful for creating circular buffers for FIFO queues, delay lines and sample buffers up to 32,767 words long, as well as for decimation, interpolation, and waveform generation. The special case of \((RN)\pm Nn \mod M\) with \(Nn = P \times 2^k\) is useful for performing the same algorithm on multiple blocks of data in memory, for example, when performing parallel Infinite Impulse Response (IIR) filtering.

### 2.5 Program Control Unit

The Program Control Unit (PCU) performs instruction prefetch, instruction decoding, hardware DO loop control and exception processing. The PCU implements a seven-stage pipeline and controls the different processing states of the DSP 56600 core. The PCU consists of three hardware blocks:

- Program Decode Controller (PDC)
- Program Address Generator (PAG)
- Program Interrupt Controller (PIC)

The PDC decodes the 24-bit instruction loaded into the instruction latch and generates all signals necessary for pipeline control. The PAG contains all the hardware needed for program address generation, system stack and loop control. The PIC arbitrates among all interrupt requests (internal interrupts as well as the five external requests IRQA, IRQB, IRQC, IRQD, and NMI), and generates the appropriate interrupt vector addresses.

The PCU implements its functions using the following registers:

- PC–Program Counter Register
- SR–Status Register
- LA–Loop Address Register
- LC–Loop Counter Register
• VBA–Vector Base Address Register
• SZ–Size Register
• SP–Stack Pointer
• OMR–Operating Mode Register
• SC–Stack Counter Register

The PCU also includes a hardware System Stack (SS).

2.6 Program Patch Logic

The Program Patch Logic (PPL) block provides the core user a way to fix the program code in the on-chip ROM without generating a new mask. Implementing the code correction is done by replacing a piece of ROM-based code with a patch program stored in RAM. The PPL consists of four Patch Address Registers (PAR1-PAR4) and four patch address comparators. Each PAR points to a starting location in the ROM code where the program flow is to be changed. The PC register in the PCU is compared to each PAR. When an address of a fetched instruction is identical to an address stored in one of the PARs, the Program Data Bus (PDB) is forced to a corresponding JMP instruction, replacing the instruction that otherwise would have been fetched from the ROM.

2.7 PLL and Clock Oscillator

The DSP 56600 core features a Phase Locked Loop (PLL) clock oscillator in its central processing module. The PLL allows the processor to operate at a high internal clock frequency using a low frequency clock input. The clock generator in the core is composed of two main blocks: the PLL, which performs the clock input division, frequency multiplication, and skew elimination, and the Clock Generator (CLKGEN), which performs low power division and clock pulse generation.
2.8 Expansion Port (Port A)

Port A is the memory expansion port and is used for both program and data memory. It provides an easy to use, low part-count connection with fast or slow static memories and with I/O devices. The Port A data bus is 24 bits wide with a separate 16-bit address bus capable of a sustained rate of one memory access per two clock cycles. External memory can be as large as $64 \times 24$-bit program memory space, depending on chip configuration. An internal wait state generator can be programmed to insert as many as thirty-one wait states if access to slower memory or I/O device is required. For power-sensitive applications and applications that do not require external memory, Port A can be fully disabled.

2.9 JTAG Test Access Port and On-Chip Emulator (OnCE)

The DSP 56600 core provides a dedicated user-accessible Test Access Port (TAP) that is fully compatible with the IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1). The test logic includes a Test Access Port consisting of four dedicated signal pins, a 16-state controller, and three test data registers. A boundary scan register links all device signal pins into a single shift register. The test logic, implemented using static logic design, is independent of the device system logic. The On-Chip Emulation (OnCE) module provides a means of interacting with the DSP 56600 core and its peripherals non-intrusively so that a user can examine registers, memory, or on-chip peripherals. This facilitates hardware and software development on the core processor.

2.10 On-Chip Memory

The memory space of the DSP 56600 core is partitioned into program memory space, X data memory space, and Y data memory space. The data memory space is divided into X data memory and to Y data memory in order to work with the two Address ALUs and to feed two operands simultaneously to the Data ALU. Memory space typically includes internal RAM and ROM and can be expanded off-chip under software control. Both internal and external memory configuration is specific to each member of the DSP 56600 family. The total on-chip and external
memory for the DSP 56602 and DSP 56603 which belong to the DSP 56600 family is tabulated in Table 2.1.

Table 2.1: On-Chip and External Memory

<table>
<thead>
<tr>
<th>Device</th>
<th>On-chip Data Memory</th>
<th>On-chip Program Memory</th>
<th>External Data/Program Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP 56602</td>
<td>25K x 16-bit X-RAM</td>
<td>5K x 24-bit RAM</td>
<td>64K x 24-bit</td>
</tr>
<tr>
<td></td>
<td>6K x 16-bit X-ROM</td>
<td>34K x 24-bit ROM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25K x 16-bit Y-RAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8K x 16-bit Y-ROM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSP 56603</td>
<td>8K x 16-bit X-RAM</td>
<td>16K x 24-bit RAM</td>
<td>64K x 24-bit</td>
</tr>
<tr>
<td></td>
<td>8K x 16-bit Y-RAM</td>
<td>3K x 24-bit ROM</td>
<td></td>
</tr>
</tbody>
</table>

2.11 Peripherals

Each member of the DSP 56600 family can be configured with its own set of on-chip peripherals for communicating with external devices or memory, as well as for providing additional on-chip functionality.

2.12 Summary

In this chapter, we have presented a description of the architectural details of the DSP 56600 core and have also discussed sections relevant to this project in greater detail. The next chapter outlines the basic building blocks that form the backbone of the loudness enhancement algorithms.
This chapter describes in greater detail the basic building blocks of the loudness enhancement algorithm implementation on the Motorola DSP 56600. A block diagram representation of the system setup is shown and thereafter each of the blocks in the diagram is described in greater detail. We also discuss the warped filter implementation of the loudness enhancement algorithms. In the current project, we have not implemented the warped filter structure for DSP simulations.

3.1 Basic Block Diagram

In this section, we shall present the fundamentals behind linear prediction which is the most important step in the bandwidth expansion technique for the loudness enhancement algorithms. This section will motivate the basic block diagram that can be used to represent the building blocks of the loudness enhancement algorithms. A brief section on the warped linear prediction is also presented.

3.1.1 Introduction to Linear Prediction

Linear prediction is the most well-known technique for modelling acoustical speech behavior [9]. Linear prediction makes use of the fact that speech varies very slowly with time with fairly stationary characteristics, that is, it is quasi-stationary. Linear prediction developed from models of speech production based on linear mathematical principles.

The linear model assumes that a glottal excitation source stimulates a vocal tract model which in turn passes through a lip radiation model. The overall model can be represented by the following equation

\[ S(z) = E(z)G(z)V(z)L(z) \] (3.1)
where $E(z)$ represents the excitation, $G(z)$ represents the glottal shaping, $V(z)$ represents the vocal tract model, and $L(z)$ represents the lip radiation model. The glottal excitation is the quasi-periodic pulse train of air generated by the vibration of vocal chords in response to air flow from the lungs. An all pole filter can be used to represent the linear speech production model, and is represented by the following equation

$$G(z)V(z)L(z) = \frac{1}{A(z)} = \frac{1}{1 - \sum_{k=1}^{p} a_k z^{-k}}$$

(3.2)

The all zero filter $A(z)$ is referred to as the inverse filter (sometimes also called as the analysis filter). This filter is used in the analysis model $E(z) = S(z)A(z)$. The reciprocal of $A(z)$ is called the all-pole model and is used in the all-pole speech synthesis $S(z) = E(z)A^{-1}(z)$.

Linear prediction of speech is based on the concept that the parameters of the speech production model vary very slowly over time and that in any interval of long enough duration, the speech waveform can be represented by a linear combination of its past values. The Linear Predictive Coding (LPC) model has been well understood since the early 1970’s and can be described by the following equation

$$s(n) = \sum_{k=1}^{p} a_k s(n - k) + Gu(n)$$

(3.3)

where $u(n)$ is the normalized glottal excitation and $G$ is the excitation gain. Eq. 3.3 leads to the following transfer function

$$H(z) = \frac{S(z)}{GU(z)} = \frac{1}{1 - \sum_{k=1}^{p} a_k z^{-k}} = \frac{1}{A(z)}$$

(3.4)

The LPC analysis equations provide a means of evaluating the prediction error. The prediction error is used as a minimization criterion in finding the optimal filter coefficients $a_k$ which best represent the speech signal in a mean squared error sense. The prediction error is basically a measurement criterion which indicates how close the synthetic representation of speech is to
the true speech signal. Let us define $\tilde{s}(n)$ as the synthetic representation of speech. Thus, $\tilde{s}(n)$ represents a linear combination of previous speech samples.

$$
\tilde{s}(n) = a_1 s(n - 1) + a_2 s(n - 2) + \ldots + a_p s(n - p)
$$

(3.5)

The prediction error is then given as

$$
e(n) = s(n) - \tilde{s}(n) = s(n) - \sum_{k=1}^{p} a_k s(n - k)
$$

(3.6)

which leads to the following transfer function

$$
A(z) = \frac{E(z)}{S(z)} = 1 - \sum_{k=1}^{p} a_k z^{-k}
$$

(3.7)

In the case when the speech $s(n)$ is actually generated using Eq. 3.5, the prediction error $e(n)$ equals the scaled glottal excitation $Gu(n)$. The main purpose of linear prediction is then to find a set of optimal coefficients $a_k$ which minimize the mean squared error. These set of equations which need to be solved in order to determine the optimal set of predictor coefficients are known as the set of normal equations and are given as

$$
\phi_n(i, 0) = \sum_{k=1}^{p} \tilde{a}_k \phi(i, k)
$$

(3.8)

where $\phi(i, k)$ represents the short-term covariances of the speech signal. These equations can be solved using the autocorrelation method shown below

$$
\sum_{k=1}^{p} r_n(|i - k|) \tilde{a}_k = r_n(i), \quad 1 \leq i \leq p
$$

(3.9)

where $r_k$ is the autocorrelation at lag $k$.

It is imperative to recall that the coefficients $\tilde{a}_k$ are related to the predictor coefficients $a_k$ by the following relation
\[ \tilde{a}_k = -a_k \quad \text{for} \quad k = 1, 2, \ldots, p \]  
\[ \text{(3.10)} \]

3.1.2 Bandwidth Expansion

As described earlier in Chapter 1, an LPC technique for loudness enhancement is to alter the formant bandwidths. Such a technique can be described by the following equation

\[ A(\tilde{z})|_{\tilde{z}=re^{jw}} = \sum_{k=0}^{p} (a_k r^{-k}) e^{-jwk} \]  
\[ \text{(3.11)} \]

This procedure is based on McCandless procedure [11] and provides us with a way of evaluating the \( z \)-transform over a circle with radius larger than or less than the unit circle \( r = 1 \). For the case \( 0 < r < 1 \), the evaluation is on a circle with radius smaller than unity. The poles are therefore closer to the circle than before and the contribution of the poles effectively increases. Also, stability is a concern for the inverse filter \( 1/A(\tilde{z}) \), since the analytic expression for the same may not have poles lying inside the circle of radius \( r \).

For the case of \( r > 1 \), the evaluation of the \( z \)-transform is on a circle farther away from the unit circle. The contribution of the poles decreases leading to a decrease in pole resonance peaks and also a corresponding expansion of pole bandwidths. Moreover, the analytic expression for the inverse filter has all its poles guaranteed to lie within the circle of radius \( r \) and hence, stability is not a concern. Translating the evaluation of the \( z \)-transform on a circle with radius \( r > 1 \) back into the filter coefficients terms, we find that this method of bandwidth expansion simply requires a scaling of the LPC coefficients by a power series of \( r \). The bandwidth broadening technique can be put in the following filter form

\[ H(z) = \frac{A(z/\gamma)}{A(z/\beta)} \]  
\[ \text{(3.12)} \]

where the bandwidth expansion factors \( \gamma \) and \( \beta \) set the level of bandwidth adjustment. Results have shown that the optimal values for \( \gamma \) and \( \beta \) are \( \gamma = 0.8 \) and \( \beta = 0.4 \) [2].
Eq. 3.12 suggests the use of FIR and IIR filter structures for the computation of the bandwidth expanded, loudness enhanced speech output. The numerator corresponds to an FIR analysis filter structure whose coefficients are the LPC coefficients scaled by a power series with common ratio $\gamma$. The denominator corresponds to an IIR synthesis filter structure whose coefficients are the LPC coefficients scaled by a power series with common ratio $\beta$.

Thus, in the computation of the bandwidth expanded, loudness enhanced speech from the original speech samples, we need to perform four basic steps:

1. Compute autocorrelation coefficients
2. Use autocorrelation coefficients to compute LPC (using Levinson-Durbin recursion)
3. Use LPC coefficients and $\gamma$ to build the FIR analysis structure and filter original speech using it
4. Use LPC coefficients and $\beta$ to build the IIR synthesis structure and filter the output from previous stage using it

These steps can be more clearly elicited in Figure 3.1.

In the next few sections, we shall describe each of the blocks in Figure 3.1 in further detail. Also, we shall present some results which show that the assembly output matches the MATLAB output for these blocks.

### 3.2 Autocorrelation

In this section, we describe the first block of Figure 3.1 which computes the autocorrelation of the input speech samples. This is an important step towards computing the linear predictive coefficients.

The input speech has been sampled at 16KHz and the autocorrelation block operates on 180 sample windows (which corresponds to 5.625ms of speech samples with 50% overlap). For the current project, speech samples from the TIMIT database are chosen for evaluation purposes.
The autocorrelation for the 180 sample window is computed using the assembly code listed in Appendix D.

The autocorrelation of the speech sample window is then used in the subsequent LPC block to compute the linear predictive coefficients using the Levinson-Durbin recursion.

### 3.3 Levinson-Durbin Recursion

From Eq. 3.9, it is clear that the basic problem of finding the linear predictive coefficients is that of solving the matrix equation \( \mathbf{R}\tilde{\mathbf{a}} = \mathbf{r} \). Here, \( \mathbf{R} \) is the autocorrelation matrix, \( \tilde{\mathbf{a}} \) are related to the linear prediction coefficients by Eq. 3.10, and \( \mathbf{r} \) is the autocorrelation vector. In 1947, Levinson [8] published an algorithm for solving the problem \( \mathbf{Ax} = \mathbf{b} \) in which \( \mathbf{A} \) is Toeplitz, symmetric, and positive definite, and \( \mathbf{b} \) is arbitrary. The autocorrelation equation in Eq. 3.9 are of this form, with \( \mathbf{b} \) having a special relationship to the elements of the matrix \( \mathbf{A} \). In 1959, Durbin [3] published a slightly more efficient algorithm for this special case. This algorithm is referred to as the Levinson-Durbin recursion in speech processing.

The Levinson-Durbin recursion can be stated by the following set of equations [13]:

![Block Diagram for the Loudness Enhancement Algorithm](image)
First step consists of the initialization of the error term which is done in Eq. 3.13. Thereafter, the $i^{th}$ reflection coefficient is computed in Eq. 3.14. The next step involves the computation of the $i^{th}$ predictive coefficient and the previous coefficients (if any) are updated using the update rule defined by Eq. 3.16. Finally the last step involves the computation of the error term and the algorithm progresses recursively until all the linear prediction coefficients have been found.

As can be seen from the above set of equations, implementation of the Levinson-Durbin recursion in assembly for a fixed-point DSP can be a challenge. Eq. 3.14 which calculates the reflection coefficients needs a division to be performed in every recursion. The built-in division routine written for the Motorola DSP 56600 provides for 32-bit dividends and 16-bit divisors. As a result, the quotient is restricted to $[-1,1)$ range. However, it is impossible to guarantee that the numerator in Eq. 3.14 will always be less than or equal to the denominator. We, therefore, have to look for other ways of getting around the division step. One solution is to write a separate subroutine for the DSP which performs division in the conventional way of subtracting the dividend from the divisor until the difference is smaller than the divisor itself. The difference
can then be divided to compute the decimal part of the quotient and the number of times we need to subtract the divisor will give us the integer part of the quotient. However, this whole process needs a large number of memory registers (9 registers, 2 accumulators, 4 data ALU input registers) and also we have a trade-off between complexity and accuracy. Since the algorithm relies on the accuracy of the pole locations rather than the LPC coefficient values themselves it makes sense to consider an approximation algorithm for computing the coefficients which themselves may not be exact, however, the pole locations will still be very close to the original pole locations. Such an approximation algorithm has been dealt with in Chapter 4.

Listed in Appendix A is the assembly code for the Levinson-Durbin recursion:

### 3.4 FIR and IIR Filters

The linear prediction block as described in the previous section outputs the LPC coefficients which form the filter coefficients for the FIR analysis filter and IIR synthesis filter with proper scaling by corresponding power series. Building a filter (FIR or IIR) in assembly requires input scaling and other issues to be taken care of to avoid overflow and underflow problems. Such issues have been discussed in greater detail below. The FIR and IIR filters in assembly are found in Appendix B.

The most basic type of filter in DSP is the FIR filter. By definition, a filter is classified as FIR if it has the following transfer function

\[
H(z) = \frac{b_0 z^{N-1} + b_1 z^{N-2} + \ldots + b_{N-2} z + b_{N-1}}{z^{M-1}},
\]

where \( b_i \in \mathbb{R}, \quad N, M \in \mathbb{Z}, \quad N > 0, \quad z \in \mathbb{C} \)

This is referred to as an N-tap FIR filter. In general, an FIR filter can be either causal or non-causal. However, FIR filters are always stable and that is the chief reason they are widely used. The difference equation which results from the above transfer function when \( N = M \) is

\[
y(n) = b_0 x(n) + b_1 x(n-1) + \ldots + b_{N-2} x(n - N + 2) + b_{N-1} x(n - N + 1)
\]

(3.19)
This is the familiar result of discrete convolution of the filter with the input data. The equations above are the idealized, mathematical representations of an FIR filter because the arithmetic operations of addition, subtraction, multiplication, and division are performed over the field of real numbers \((\mathbb{R}, +, \times)\), i.e., in the real number system. In practice, both the data values and the coefficients are constrained to be fixed-point rationals. While this set is closed, it is not “bit-bounded”, i.e., the number of bits required to represent a value in the fixed-point rationals can be arbitrarily large. In a practical system, one is limited to a finite number of bits in the words used for the filter input, coefficients, and filter output. Most current DSPs provide ALUs and memory architectures to support 16-bit, 24-bit, or 32-bit wordlengths, however, one may implement arbitrarily long lengths by customizing the multiplications and additions in software and utilizing more processor cycles and memory. The final choices, however, are governed by many aspects of the design such as required speed, power consumption, SNR, cost and others.

There are generally two methods of operating on fixed-point data viz. integer and fractional. The integer method represents data as integers and performs integer arithmetic. The fractional method assumes the data are fixed-point rationals bounded between -1 and +1. Except for an extra left shift performed in fractional multiplies, these two methods can be considered equivalent.

### 3.5 Scaling FIR Coefficients

Consider an FIR filter with \(N\) coefficients \(b_0, b_1, \ldots, b_{N-1}\), \(b_i \in \mathbb{R}\). In fixed-point arithmetic, a binary word can be interpreted as an unsigned or signed fixed-point rational. Although there are a number of situations in which the filter coefficients could be the same sign and thus could be represented using unsigned values, let us assume that they are not and hence we must utilize signed fixed-point rationals for our coefficients. Thus, we must find a way of representing, or more accurately, of *estimating*, the filter coefficients using signed fixed-point rationals.

Since a signed fixed-point rational is of the form \(B_i/2^b\), where \(B_i\) and \(b\) are integers, \(-2^{M-1} \leq B_i \leq 2^{M-1} - 1\), and \(M\) is the wordlength used for the coefficients, we determine the estimate \(b'_i\)
of coefficient $b_i$ by choosing a value for $b$ and then determining $B_i$ as

$$B_i = \text{round}(b_i \ast 2^b)$$

(3.20)

Then

$$b'_i = B_i / 2^b$$

(3.21)

In general, $b'_i$ is only an estimate of $b_i$ because of the rounding operation. This approximation is called \textit{coefficient quantization}. The quantization error can be determined by the following

$$e_i = b'_i - b_i$$  
$$= B_i / 2^b - b_i$$  
$$= \frac{\text{round}(b_i \ast 2^b)}{2^b} - b_i$$

(3.22)

The question that arises then is how do we choose $b$? In order to answer this, note that the maximum error $e_{i\text{max}}$ a quantized coefficient can have will be one-half of the bit being rounded at, i.e.,

$$e_{i\text{max}} = 2^{-b} / 2$$  
$$= 2^{-b-1}$$

(3.23)

It is now easy to see that lacking any additional criteria, the ideal value for $b$ is the maximum it can be since that will result in the least amount of coefficient quantization error. However, $b$ is from the integers, and the integers can go to infinity. Again, considering the coefficient wordlength to be $M$ bits, the maximum magnitude a signed two’s complement value has is $2^{M-1} - 1$. Therefore, we must be careful not to choose a value for $b$ which will produce a $B_i$ that has a magnitude larger than $2^{M-1} - 1$. When a value becomes too large to be represented by the representation we have chosen, then we say that an \textit{overflow} has occurred. Thus to avoid
overflow, the value of $b$ that will not overflow the largest magnitude coefficient can be computed as

$$b = \lfloor \log_2((2^M - 1)/\max(|b_i|)) \rfloor \quad (3.24)$$

In summary, we see that the ideal value for $b$ is the maximum value which can be used without overflowing the coefficients since that provides the minimum coefficient quantization error. However, adding two $J$-bit values requires $J + 1$ bits in order to maintain precision and avoid overflow. This can be easily extended to a sum of multiple values and we find that the sum of $N$ $J$-bit values requires $J + \lceil \log_2 N \rceil$ bits to maintain precision and avoid overflow.

Let us consider an $N$-tap FIR filter which has $L$-bit data values and $M$-bit coefficients. Then using the above relations, the final $N$-term sum required at each time interval $n$,

$$y(n) = b'_0x(n) + b'_1x(n-1) + \ldots + b'_{N-1}x(n-N+1) \quad (3.25)$$

requires $L + M + \log_2 N$ bits in order to maintain precision and avoid overflow. Most processors and hardware components provide the ability to multiply two $M$-bit values together to form a $2M$-bit result. Most general purpose and some DSP processors provide an accumulator that is the same width as the multiplier output. Some DSP processors provide a $2M + G$-bit accumulator, where $G$ denotes “guard bits.” Therefore, another criteria in the design of FIR filters is that the final convolution sum fit within the accumulator. To put it algebraically, we require that

$$2M + \log_2 N \leq 2M + G \quad (3.26)$$

assuming that the coefficient wordlength and the data wordlength is the same ($M$ bits). The key point here is that the number of bits required for the filter output increases with the length of the filter. For situations where we don’t have guard bits ($G = 0$), we see that we immediately have problems even for a 2-tap filter. This is precisely why the guard bits are provided because they guard against overflow when performing summations. However, even though the accumulator
may have guard bits, it is still possible to overflow the accumulator if \( \log_2 N > G \), i.e., if we attempt to use a filter that is longer than \( 2^G \) taps. However, in the current project, we have 8 guard bits and are using a 4th order FIR filter. Therefore, we can be assured that the accumulator will not overflow despite the presence of guard bits.

Consider the convolution sum shown in Eq. 3.19. The signs of \( x(k) \) which will make the terms in \( b_i x(n - i) \) all positive will result in larger output. This occurs when \( \text{sgn}(x(n - i)) = \text{sgn}(b_i) \). Therefore the convolution sum can be rewritten as

\[
y(n) = \sum_{i=0}^{N-1} b_i x(n - i)
\]

\[
= \sum_{i=0}^{N-1} b_i |\text{sgn}(b_i)| |x(n - i)|
\]

\[
= \sum_{i=0}^{N-1} |b_i||x(n - i)|
\]

(3.27)

If we let \( x_{MAX} \) denote the maximum magnitude of \( x(n) \), then the maximum sum represented above would be

\[
y_{MAX} = \sum_{i=0}^{N-1} |b_i|x_{MAX}
\]

\[
= x_{MAX} \sum_{i=0}^{N-1} |b_i|
\]

\[
= \alpha x_{MAX}
\]

(3.28)

where \( \alpha = \sum_{i=0}^{N-1} |b_i| \) represents the coefficient area. Using scaled representation format, we have

\[
y_{MAX} = \frac{\alpha X_{MAX}}{2^b_x}
\]

(3.29)

Similarly using the scaled representation for \( y_{MAX} \), we have,

\[
Y_{MAX} = 2^b_x \alpha X_{MAX}
\]

(3.30)
For an $A$-bit accumulator for storing the output with $L$-bit data wordlength and coefficient area $\alpha$, the maximum value for the coefficient scale factor $b_b$ is

$$b_b = A - L - \left\lceil \log_2 \alpha \right\rceil$$ (3.31)

To summarize, we need to maximize $b_b$ to reduce quantization error, also we need to constrain $b_b$ so that the coefficient with the largest magnitude is representable, and finally we need to constrain $b_b$ so that overflows in the convolution sum are avoided. Taking these three criteria into consideration, the value of $b_b$ that we seek is given by

$$b_b = \min\left(\left\lfloor \log_2 \left(\frac{2^M - 1}{\max(|b_i|)}\right) \right\rfloor, A - L - \left\lceil \log_2 \alpha \right\rceil\right)$$ (3.32)

This section provided a binary mathematical point of view towards coefficient scaling to avoid overflows in FIR filters.

Figure 3.2 shows a zoomed in version of an overlay graph of the MATLAB and assembly output for the autocorrelation of 180 sample windows for the sentence “She had your dark suit in greasy wash water all year.” This sentence was taken from the TIMIT database. The blue solid line shows MATLAB output while the red dotted line shows the assembly output. The outputs match each other within the precision of the hardware and as such it difficult to discern the two plots from each other. Figure 3.3 shows another such overlay plot for the FIR output of a single phoneme (783 samples) being passed through an FIR analysis filter. Also the IIR output of the same phoneme passed through an IIR synthesis filter with bandwidth expansion factor $\gamma = 0.909$ does not match the MATLAB output exactly but is slightly off from it. The difference is so small that it is not of much significance.

3.6 Warped Filter Implementation

In this section, we present a brief overview of the warped filter structure implementation for the loudness enhancement algorithms. The warped filter technique is used to increase the
Figure 3.2: Overlay of MATLAB and Assembly output for autocorrelation

bandwidth on a critical band scale instead of a linear band scale. As we saw in Sec. 3.1.2, the
LPC pole placement technique leads to a linear fixed increase in bandwidth independent of the
frequency. However, as discussed in Chapter 1, the loudness enhancement technique involves
increasing the bandwidth on a critical band scale. This requires an additional degree of freedom
for bandwidth adjustment. The all-pass warping factor $\alpha$ provides this additional degree of
freedom.

Eq. 3.11 shows how the $z$-transform can be evaluated over a circle with radius $r$ for a given
set of LPC coefficients. The radius determines the amount of bandwidth expansion and this is
fixed over the entire frequency scale. However, it would be desirable to introduce some kind of
non-linearity in the bandwidth expansion based on the critical band concept for human auditory
system. This non-linearity is introduced by warping the frequency scale. Warping refers to alteration of the frequency scale. In simpler terms, it refers to a stretching or compression of the frequency scale. Warping can be represented by a functional one-to-one mapping of the unit circle onto itself. The mapping function itself lies in the $z$ domain and the following mappings define the relation between the $z$ domain and the warped $z$ (referred to as $\tilde{z}$) domain.

$$\tilde{z} = f(z)$$

(3.33)

$$z = g(\tilde{z})$$

(3.34)
The bilinear transform is one such one-to-one mapping which is easily invertible too. It corresponds to the first-order all-pass filter as shown below:

\[
\tilde{z}^{-1} = \frac{z^{-1} - \alpha}{1 - \alpha z^{-1}} \quad -1 < \alpha < 1
\] (3.35)

All-pass systems have a unit-magnitude response and passes all frequencies with unit magnitude. They are mainly used to compensate for group-delay distortions. In the case of warped filter structures, the ability of all-pass systems to distort the phase is used favorably to alter the frequency scale. \(\alpha\) is the dispersive delay element and sets the degree of frequency warping. The dispersive elements inject frequency dependence of digital filter outputs thereby resulting in a non-uniform frequency resolution. The \(z\)-transform in the warped domain with respect to the warped frequency scale is the same as the \(z\)-transform in the normal frequency domain. The warped filter structures can be found in greater detail in [2].

In the next chapter, we will discuss how we modified the original algorithm to overcome the challenges encountered in implementation of the Levinson-Durbin recursion.
In the previous chapter, we discussed the various building blocks to be implemented for the bandwidth expansion technique for loudness enhancement. These were the autocorrelation, LPC, FIR and IIR filter blocks respectively. We also presented some overlay graphs which showed that each of these blocks worked perfectly well. However, we had overflow problems in the division routine in the LPC block as was described in that chapter. The current chapter deals with the problem of finding a solution to this issue and so also implementing the solution in assembly for the Motorola DSP 56600.

4.1 The Solution

In the previous chapter, we saw that to compute the filter coefficients for the FIR and IIR filters, we needed to first compute the linear predictive coefficients (LPC). These coefficients then appropriately scaled by the radius terms constituted the filter coefficients. The Levinson-Durbin algorithm as shown in Eqs. 3.13 - 3.17 involved computation of the reflection coefficients ($k_i$'s). This involved the use of a division routine for the fixed-point Motorola DSP 56600. The main problem associated with the division routine was the overflow upon division which led to recursive errors as the Levinson-Durbin is a recursive method of computing the coefficients. This obviously affects the pole locations for the FIR and IIR filters and since these pole locations are crucial in determining the formant locations for the re-synthesized speech, it becomes necessary to be able to obtain the linear predictive coefficients with greater accuracy. Moreover, if we can avoid the division routine in the computation of the LPC coefficients, then we can guarantee the solution to converge without overflow problems. The LMS is one such elegant algorithm which makes use of a feedforward structure for estimating the filter coefficients. The details about
the LMS algorithm are provided in the coming sections and we shall show that this algorithm performs considerably better than the Levinson-Durbin on a fixed-point DSP.

### 4.2 Least Mean Squares (LMS) Algorithm

In 1960, Widrow and Hoff developed a widely used algorithm called the *Least Mean Square (LMS) algorithm*. The method of steepest descent uses a fixed gradient in the recursive computation of the Wiener filter for stochastic inputs. However, in contrast, the LMS uses a “stochastic gradient” in this computation and hence, the LMS is an important member of the *stochastic gradient algorithms* family. The most salient features of the LMS algorithm are its simplicity, non-requirement of pertinent correlation functions and so also no matrix inversion is needed. This simplicity has made the LMS algorithm a standard against other linear adaptive filtering algorithms which have been benchmarked.

The LMS algorithm consists of two basic steps:

1. A *filtering process* involving the computation of a filter output in response to a specified input and then generating an estimation error by computing the difference between a desired signal and the output of the filter.

2. A *adaptive process* wherein the parameters of the filter (filter coefficients) are automatically adjusted based on the estimation error.

These two steps together can be depicted by the feedback loop shown in Figure 4.1.

First, we have the transversal filter which is responsible for the filtering process and next we have an adaptive weight control block which performs an adaptive control mechanism on the filter coefficients. The transversal filter consists of an $M^{th}$ order feedforward structure with $M-1$ delay elements, $M$ tap inputs and $M$ weights for each of the inputs. During the filtering process, the desired response $d(n)$ is provided for processing besides the input vector $u(n)$. The transversal filter produces an estimate $d_{est}(n)$ for the desired signal. Based on this estimate we can compute an estimation error $e(n)$. The estimation error along with the input $u(n)$ are then applied to the adaptive control mechanism to estimate the new set of tap weights for the transversal filter.
The LMS algorithm uses the product of $u(n - k)e^*(k)$ as an estimate for the $k^{th}$ element in the gradient vector $\nabla J(n)$ that characterizes the method of steepest descent \[6\].

Stability might be a concern since the LMS filter involves feedback. In this context, a meaningful criterion is to require that

$$J(n) \to J(\infty) \quad \text{as} \quad n \to \infty \quad (4.1)$$

where $J(n)$ is the mean-square error produced by the LMS filter at time $n$ and its final value $J(\infty)$ is a constant. For the LMS algorithm to satisfy this criterion, the step-size parameter $\mu$ has to satisfy a certain condition related to the spectral content of the tap inputs.

The difference between the final value $J(\infty)$ and the minimum value $J_{min}$ attained by the Wiener-Hopf solution is called the *excess mean-square error* $J_{ex}(\infty)$. This difference represents the excess price paid for using the adaptive LMS approach for computing the filter weights as compared to a deterministic approach as in the method of steepest descent. The ratio of $J_{ex}(\infty)$
to $J_{\text{min}}$ is called the misadjustment. It is a measure of how far the solution of the LMS filter approach is away from the Wiener solution. However, the misadjustment can be controlled by the proper choice of the step-size parameter $\mu$. The misadjustment is related to the step-size parameter by

$$M = \frac{\mu}{2} \text{tr}[R] \quad (4.2)$$

where $R$ is the autocorrelation matrix.

The LMS filter is simple in implementation but at the same time is very strong in delivering high performance due to its ability to adapt to the external environment. However, we have to pay special attention to the proper choice of the step-size parameter $\mu$. The LMS algorithm can be derived from the steepest descent algorithm by replacing the gradient vector by its instantaneous estimate. The derivation can be found in greater detail in [6]. The LMS algorithm in its final form comprises of the following three set of equations

$$y(n) = \hat{w}^H(n)u(n) \quad (4.3)$$

$$e(n) = d(n) - y(n) \quad (4.4)$$

$$\hat{w}(n + 1) = \hat{w}(n) + \mu u(n)e^*(n) \quad (4.5)$$

Eq. 4.3 computes the filter output and represents the filtering process. In Eq. 4.4, the error is estimated on the basis of the current desired signal and finally the filter tap weights are updated in Eq. 4.5. These equations represent the LMS algorithm in its complex form. We find that the LMS algorithm requires $2M+1$ complex multiplications and $2M$ complex additions per iteration where $M$ is the number of tap weights used in the transversal filter. In other words, the computational complexity of the LMS algorithm is $O(M)$ which is much easier to implement.
in a DSP as opposed to the complex Levinson-Durbin algorithm as shown in Chapter 3. The LMS algorithm can be used for a wide variety of applications. Some of the most commonly used applications of the LMS are adaptive noise cancellation, adaptive beamforming, adaptive line enhancement and linear prediction. In the next section, we describe the application of the LMS algorithm in the determination of linear prediction coefficients of speech.

4.3 Linear Prediction Using LMS

Recalling from Chapter 3, linear prediction of speech involves the estimation of the current speech sample using previous speech samples. As shown in Eq. 3.5, \( \hat{s}(n) \) is the estimate for the current speech sample based on the previous speech samples \( s(n-1), s(n-2), \ldots, s(n-p) \) for a linear predictor of order \( p \). Based on the discussion in the previous section, if we delay the input sequence by one sample and feed the resulting vector \( (u(n-1)) \) as input to the transversal filter with a desired response \( d(n) = u(n) \), then the resulting tap weight vectors for a suitable step-size parameter \( \mu \) and an appropriate number of passes (which depends on the speed of the DSP) would closely match the true linear predictive coefficients. Clearly, this is an elegant approach to finding the linear predictive coefficients as compared to the Levinson-Durbin recursion within the limitations of the fixed-point DSP.

Figure 4.2 shows the transversal filter structure with a feedback loop for computing the linear predictive coefficients adaptively. The input \( u(n) \) is first delayed by one sample and then followed by the transversal feedforward structure. Also, the current input \( u(n) \) serves as the desired signal and the filter tap weights (linear predictive coefficients) are updated accordingly.

4.4 Experimental Results

In this section, we present the results obtained from simulating the LMS algorithm for computation of the linear predictive coefficients on the Motorola DSP 56600. The assembly code for the LMS algorithm can be found in Appendix C. Sentences were taken from the TIMIT database. Speech is sampled at 16KHz and sentences are broken into 180 sample windows with 50% overlap. A set of four linear predictive coefficients are computed for each frame. We use
the LMS algorithm with an initial step-size of $\mu = 0.2$ with an automatic update for $\mu$ for each pass of the frame. For each pass, $\mu$ is scaled down by a factor of 0.99. This ensures we have a smaller misadjustment towards the latter passes. The idea is to converge faster to the correct solution in the earlier passes and then to achieve better accuracy using a smaller step-size in the latter passes. We perform 20 passes for each frame which still leaves us with enough time to perform computations on the current frame before the next frame arrives for the Motorola DSP 56600 running at 60MHz clock cycle. Number of clock cycles for each of the Autocorrelation, Levinson-Durbin, LMS, Modified signed LMS, FIR and IIR filter blocks along with their durations for each frame of data are tabulated in Table 4.1.

Figure 4.3 shows the LPC value tracks (which are simply the negative of the weight values) for 260 frames of speech, each frame being 180 samples long, for the sentence “She had your dark suit in greasy wash water all year.”
These LPC values do not match the true LPC values exactly as is expected, however, if we look at the variation in pole locations with each pass and the final location of the poles, we see that the poles match the original poles very closely. This is highly desirable as the poles are crucial in determining the formant locations rather than the LPC values themselves. The pole tracks for the first pass for the first frame in the sentence under consideration are shown in Figure 4.4 followed by Figure 4.5 which shows the pole location variations in the second pass. It is clear from the figures that the poles begin to stabilize with increasing number of passes.

For the first few frames of speech (typically silence), the coefficient updates resulting from the LMS algorithm are small enough to be accurately represented by the limited 16-bit precision of the DSP. As such, for the first few frames, the coefficients stay at zero for the DSP as compared
to MATLAB which has a 64-bit floating point representation. Due to this underflow problem, the coefficients do not match exactly but are close to each other. Table 4.2 shows the mean and variance of the true pole locations and the assembly simulated pole locations.

An overlay plot for the true LPC pole locations and the assembly pole locations for all the 20 passes for the 201\textsuperscript{st} frame are shown in Figure 4.6.

It is clear from the above results that the LMS algorithm turns out to be an elegant approach to finding the linear predictive coefficients circumventing the division overflow problem which led to recursive errors in the computation of the coefficients. Besides, the LMS also requires fewer clock cycles per pass for each frame as compared to the Levinson-Durbin recursion. Moreover, we see that the whole algorithm can be completely executed on each frame of data in 5.62ms
Table 4.1: Clock cycles for Levinson-Durbin, LMS, FIR and IIR filter blocks

<table>
<thead>
<tr>
<th>Block</th>
<th>Number of Clock Cycles</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autocorrelation</td>
<td>105372</td>
<td>1.756ms</td>
</tr>
<tr>
<td>Levinson-Durbin</td>
<td>25075</td>
<td>0.418ms</td>
</tr>
<tr>
<td>LMS</td>
<td>310985</td>
<td>5.18ms</td>
</tr>
<tr>
<td>Modified signed LMS</td>
<td>371663</td>
<td>6.19ms</td>
</tr>
<tr>
<td>FIR</td>
<td>12423</td>
<td>0.21ms</td>
</tr>
<tr>
<td>IIR</td>
<td>13637</td>
<td>0.23ms</td>
</tr>
<tr>
<td>Overall using LMS</td>
<td>337045</td>
<td>5.62ms</td>
</tr>
<tr>
<td>Overall using modified signed LMS</td>
<td>397723</td>
<td>6.63ms</td>
</tr>
<tr>
<td>Overall using Levinson-Durbin</td>
<td>156507</td>
<td>2.61ms</td>
</tr>
</tbody>
</table>

which still leaves us with plenty of time to account for the external data interface operations for a 180 sample frame of speech being sampled at 16KHz.

However, from Table 4.2, we see that although the variances in the pole radii and angles are very small yet the mean pole locations using LMS are slightly off from the true pole locations. This suggests that the variances in pole locations is not a good measure of performance. Therefore, we looked at the root mean squared error values of the pole radii and angles which tell us how far away from the true pole locations are we when using the LMS algorithm to compute the linear prediction coefficients. Also since the variances in pole locations was so small, the updates in the weight values started to fall below the minimum machine precision of the hardware. This led us to the development of the modified signed LMS algorithm which is a blend between the LMS algorithm and the signed LMS algorithm. The modified signed LMS can be described by the following weight update equation:

\[
\hat{w}(n+1) = \hat{w}(n) + \text{sgn}(\mu(n)e^*(n)) \times \max(\text{eps}, \text{abs}(\mu(n)e^*(n)))
\]

(4.6)

As can be seen from this equation, the signed LMS (\text{eps}=0) algorithm is a special case of the modified signed LMS. The objective in using the modified signed LMS was to force an update
equal to the minimum precision of the hardware (\(\varepsilon\)) when an underflow occurred. Whenever
the update is large enough to be correctly represented using the 16-bit DSP, the modified signed
LMS algorithm switches back to the LMS algorithm. This algorithm turned out to be a very good
replacement for the Levinson-Durbin recursion and also exhibited very low root mean squared
error values for the pole locations as compared to the LMS algorithm. Table 4.3 shows the root
mean squared error values for the radius and angle locations for the 4 poles corresponding to the
4 LPC coefficients which have been computed in the current project for each frame of speech
samples. The table clearly shows that the modified signed LMS algorithm yields poles which are
closer to the true poles by almost a factor of 0.5. The assembly code for the modified signed
LMS is included in Appendix E.
Table 4.2: Mean and Variance of pole locations

<table>
<thead>
<tr>
<th>Pole</th>
<th>$\mu_r$</th>
<th>$\mu_\theta$ (in radians)</th>
<th>$\sigma_r^2$</th>
<th>$\sigma_\theta^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True1</td>
<td>0.536</td>
<td>0.7055</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>True2</td>
<td>0.536</td>
<td>-0.7055</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>True3</td>
<td>0.4346</td>
<td>2.1336</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>True4</td>
<td>0.4346</td>
<td>-2.1336</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Assembly1</td>
<td>0.5719</td>
<td>0.6835</td>
<td>$6.46 \times 10^{-6}$</td>
<td>$4.438 \times 10^{-6}$</td>
</tr>
<tr>
<td>Assembly2</td>
<td>0.5719</td>
<td>-0.6835</td>
<td>$6.46 \times 10^{-6}$</td>
<td>$4.438 \times 10^{-6}$</td>
</tr>
<tr>
<td>Assembly3</td>
<td>0.4799</td>
<td>2.1738</td>
<td>$1.51 \times 10^{-5}$</td>
<td>$5.395 \times 10^{-6}$</td>
</tr>
<tr>
<td>Assembly4</td>
<td>0.4799</td>
<td>-2.1738</td>
<td>$1.51 \times 10^{-5}$</td>
<td>$5.395 \times 10^{-6}$</td>
</tr>
</tbody>
</table>

Figure 4.7 shows the LPC value tracks obtained using the modified signed LMS for 260 frames of speech, each frame being 180 samples long, for the sentence “She had your dark suit in greasy wash water all year.” When compared to Figure 4.3, we see that the linear prediction coefficients from assembly match the true values from MATLAB more closely.

Table 4.4 illustrates a comparison of the root MSE for the modified signed LMS algorithm with the number of passes for each frame. The table indicates that with as low as 10 passes the modified signed LMS algorithm yields pole locations which are quite close to the true pole locations. This can lead to additional reduction in execution time.

These results clearly indicate that the modified signed LMS is a very good replacement for the Levinson-Durbin recursion for computing the linear prediction coefficients. The total execution time for the overall algorithm using the modified signed LMS in place of the Levinson-Durbin recursion is tabulated in Table 4.1.
Figure 4.6: Pole Value Tracks for all 20 passes for the 201st frame

Table 4.3: Comparison of Root MSE values of pole locations using LMS and modified signed LMS algorithms

<table>
<thead>
<tr>
<th>Value</th>
<th>Root MSE for LMS</th>
<th>Root MSE for modified signed LMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>0.0408</td>
<td>0.0205</td>
</tr>
<tr>
<td>$r_2$</td>
<td>0.0408</td>
<td>0.0205</td>
</tr>
<tr>
<td>$r_3$</td>
<td>0.04</td>
<td>0.0177</td>
</tr>
<tr>
<td>$r_4$</td>
<td>0.04</td>
<td>0.0177</td>
</tr>
<tr>
<td>$\theta_1$</td>
<td>0.0321</td>
<td>0.0184</td>
</tr>
<tr>
<td>$\theta_2$</td>
<td>0.0321</td>
<td>0.0184</td>
</tr>
<tr>
<td>$\theta_3$</td>
<td>0.0522</td>
<td>0.0388</td>
</tr>
<tr>
<td>$\theta_4$</td>
<td>0.0522</td>
<td>0.0388</td>
</tr>
</tbody>
</table>
Table 4.4: Comparison of Root MSE values of pole locations using modified signed LMS algorithms vs. number of passes

<table>
<thead>
<tr>
<th>Number of passes</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$\theta_1$</th>
<th>$\theta_2$</th>
<th>$\theta_3$</th>
<th>$\theta_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.0302</td>
<td>0.0302</td>
<td>0.0214</td>
<td>0.0214</td>
<td>0.5415</td>
<td>0.5415</td>
<td>0.5685</td>
<td>0.5685</td>
</tr>
<tr>
<td>10</td>
<td>0.0212</td>
<td>0.0212</td>
<td>0.0188</td>
<td>0.0188</td>
<td>0.0218</td>
<td>0.0218</td>
<td>0.0391</td>
<td>0.0391</td>
</tr>
<tr>
<td>15</td>
<td>0.0207</td>
<td>0.0207</td>
<td>0.0185</td>
<td>0.0185</td>
<td>0.0210</td>
<td>0.0210</td>
<td>0.0390</td>
<td>0.0390</td>
</tr>
<tr>
<td>20</td>
<td>0.0205</td>
<td>0.0205</td>
<td>0.0177</td>
<td>0.0177</td>
<td>0.0184</td>
<td>0.0184</td>
<td>0.0388</td>
<td>0.0388</td>
</tr>
</tbody>
</table>

Figure 4.7: LPC Value Tracks
CHAPTER 5
CONCLUSIONS AND FUTURE WORK

The primary goal of this thesis has been to take a step towards implementing the loudness enhancement techniques [2] in real-time on a Motorola DSP 56600 which is a 16-bit fixed point DSP. Experimental results revealed the challenges in implementing the Levinson-Durbin recursion in a fixed point DSP and the LMS algorithm was presented as an elegant solution to this problem. We also showed that the FIR and IIR filtering processes needed input scaling to prevent overflows and underflows and a binary mathematical discussion was presented. Results indicate that the LMS performs very well in comparison to the Levinson-Durbin recursion within the limitations of the underlying hardware. An analysis of the computation time in terms of number of clock cycles was also presented. This analysis is crucial to ensure that the implemented algorithm can run in real-time. Results showed that the bandwidth expansion technique which has been implemented leaves us with sufficient time to process a single frame before the next frame arrives. Sentences were picked up from the TIMIT database which is a testing standard for most of the speech enhancement and recognition systems. Input speech is sampled at 16KHz and is broken up into 180 sample windows with 50% overlap. These frames are then processed by the Motorola DSP 56600 running at 60MHz clock cycle taking a total of 5.62ms to process a single frame.

Since this has been a step towards implementing the whole loudness enhancement algorithm described in [2], we have focussed our efforts on implementing the preliminary form of a linear bandwidth expansion as developed in [2]. This leaves us with the scope of implementing the warped filter structure which incorporates the psychoacoustic nature of the human auditory system to achieve a non-linear bandwidth expansion. Besides this, we can also focus on improving the efficiency of the current algorithm to make it run faster on the DSP. Another area of research
that can be investigated is the comparison between the Levinson-Durbin recursion implementation as opposed to the LMS implementation when a separate custom subroutine to perform 40-bit division has been written for the current DSP. The two algorithms can be compared for the complexity of implementation versus the accuracy of the solution. Also, we can investigate writing dynamic algorithms for preventing underflow for the initial frames of the LMS when the coefficient updates are very small and also preventing overflow for the latter frames when the coefficients themselves increase beyond the [-1,1) range.

These algorithms running in real-time will form an important component of most state-of-the-art cellular phone technology in years to come. The most important advantage of these algorithms is the ability to increase loudness at the same energy level thereby saving considerably on battery life, which for a consumer is one of the most important factors to be considered in making a decision for buying a particular model vis-a-vis another.
This appendix contains the assembly Levinson-Durbin recursion code which we have implemented for the Motorola DSP 56600.

MATLAB code for Durbin recursion:

```matlab
function [ar1]=durbin_orig(rr,N);
    k = quant16(rr(2)/rr(1));
    a0=1; ar(2)= quant16(rr(2)/rr(1)*a0);
    e =quant16((1-(rr(2)/rr(1))^2)*rr(1));
    for i=3:N+1
        k = quant16((rr(i)-sum(ar(2:i-1)'.*rr(i-1:-1:2)))/e);
        ar(i)=k;
        ar(2:i-1) = quant16(ar(2:i-1)-k*fliplr(ar(2:i-1)));
        e=quant16((1-k^2)*e);
    end
    ar1=-ar;
    ar1(1)=1;
```

`durbin.asm`:

```assembly
move   #rr,r0
do    #nk+1,loadrr
```
move x:input_ptr,a
move a,x:(r0)+

loadrr
move #rr,r0
move #k,r1

; Begin Durbin Algorithm
move x:(r0)+,x0
move x:(r0)+,a
abs a a,b
eor x0,b #acoeffs+1,r4
andi #$fe,ccr
rep #16
div x0,a
jmi L1
neg a

L1 move a0,a
clr b a,x:(r1)+ a,y0
move #$7fff,b1
macr -y0,y0,b #2,r7
move b,x1 a,y:(r4)+
mpyr x1,x0,a #2,n7
move a,y1
move #-2,n5

; outer do loop (note: alpha = y1)
do #nk-1,L6
move r0,r2
move #acoeffs,r5
move (r0)+
clr a x:(r2)-,x0 y:(r5)+,y0
move x0,a
move a,x0
clr a

; inner do loop #1 (note: r7 = i)
doi r7,L2
mac x0,y0,a x:(r2)-,x0 y:(r5)+,y0
move x0,b
move b,x0
clr b

; back to outer do loop (note: error = a)
L2
do #>2,sd
asr a

sd
abs a a,b
eor y1,b #1,n6
andi #$fe,ccr
rep #16
div y1,a
jmi L3
neg a
L3
move a0,a
clr b a,x0 a,y:(r4)+
move a,x:(r1)+
move (r7)-
move  #$7fff,b1
macr  -x0,x0,b  r4,r6
move  b,x1
mpyr  x1,y1,b  (r6)-n6
move  b,y1
move  #acoeffs+1,r5
move  #anew+1,r3
move  y:(r6)-,y0
move  y:(r5)+,a

;  inner do loop #2  (note: r7 = (i-1))
do  r7,L4
macr  x0,y0,a  y:(r6)-,y0
move  a,y:(r3)+
move  y:(r5)+,a

;  end of inner do loop #2  ;  inner do loop #3  (note: r7
;  := (i-1))
L4
move  x:(r3)-,x0  y:(r5)+n5,y0
do  r7,L5
move  y:(r3)-,a
move  a,y:(r5)-

;  end of inner do loop #3  ;  end of outer do loop
L5
move  (r7)+n7
L6
move  #acoeffs+1,r4
do  #nk,endsend
move y:(r4)+,x0
move x0,x:output_ptr
endsend
	nop
	nop
	nop

DURBIN_FINISH
APPENDIX B
ASSEMBLY CODE FOR IIR AND FIR FILTERS

This appendix lists the assembly code for the IIR and FIR filters that have been implemented for the Motorola DSP 56600.

fir.asm

FIR_start

move  #in,r4
move  #fcoeffs,r0
do  #N,loadf
move  x:input_ptr1,a
do  #$>3,scaledown
asr  a
scaledown
nop
move  a,x:(r0)+
loadf
do  #M,loadin
move  y:input_ptr2,a
nop
move  a,y:(r4)+
loadin
move    #fcoeffs,r0
move    #in,r4
do      #M,_FIRcomp
move    #M-1,m4
move    #N-1,m0
clr     a
move    x:(r0)+,x0
move    y:(r4)-,y0
rep     #N-1
mac     x0,y0,a x:(r0)+,x0 y:(r4)-,y0
macr    x0,y0,a
do      #M-N-1,_update
move    (r4)-
_update
nop
move    a,x:output_ptr
_FIRcomp
_end
FIR_finish

iir.asm
IIR_start

move    #fcoeffs,r0
move    #>0,r1
move    #>r,x0
do      #N,loadf
move x:input_ptr1,a

nop

move a,x1

do r1,_mul

mpy x0,x1,a

nop

move a,x1

_mul

nop

move a,x:output_ptr2

neg a

nop

move a,x:(r0)+

move (r1)+

loadf

move #fcoeffs,r0

move x:(r0)+,a

neg a

nop

move (r0)-

move a,x:(r0)+

move #fcoeffs,r0

IIR_break

do #>N,storef

move x:(r0)+,a

nop

move a,y:output_ptr1
storef

move #fcoeffs,r0
move #states,r4
do #M,IIRcomp
move #M-1,m4
move #N-1,m0
clr a
nop
move y:input_ptr2,a
do #2,scaledown2
asr a

scaledown2

move x:(r0)+,x0 y:(r4)-,y0
rep #N-1
mac x0,y0,a x:(r0)+,x0 y:(r4)-,y0
macr x0,y0,a
nop
do #M-N-1,_update
move (r4)-

_update

move (r4)-
move a,y:(r4)
move a,x:output_ptr
move (r4)+

IIRcomp

IIR_finish
APPENDIX C
ASSEMBLY CODE FOR LMS ALGORITHM

This section lists the assembly code for the LMS algorithm implemented for the Motorola DSP 56600.

; This program calculates the LPC coefficients using the LMS algorithm

nk equ 4
mu equ 0.2 ;0.2/4
scaler equ 0.99
win equ 176
len equ 180
npass equ 20
nframes equ 260

SECTION xram2

xdef input_ptr,input_ptr1,output_ptr
xdef acoeffs
xdef samp,newsamp,data,desired

org x:$0
samp ds nk
newsamp ds nk
data ds len
desired ds win
input_ptr dc 1
input_ptr1 dc 1
output_ptr dc 1

org y: $0
acoeffs ds nk

ENDSEC

SECTION mydurbin

xref input_ptr,input_ptr1,output_ptr
xref acoeffs
xref samp,newsamp,data,desired

org p:
lms_test

move #-1,n0
move n0,n4
move #nk-1,m0
move #len-1,m3
move #win-1,m6
move m0,m2
move m0,m4
move m0,m5
move #samp,r0
move #newsamp,r2
move #data,r3
move #desired,r6
move #acoeffs,r4
move r4,r5

do #nframes,_allframes

do #len,_readfile
move x:input_ptr,a
move a,x:(r3)+

_readfile

do #win,_readdesired
move x:input_ptr1,a
move a,x:(r6)+

_readdesired

do #npass,_morepasses

; move #>0,r1
move #mu,y1
do  #nk,_readdata
move  x:(r3)+,a
move  a,x:(r0)+

_readdata

  do  #win,_onepass
  clr  b  x:(r0)+,x0  y:(r4)+,y0
  move  #>scaler,x1

  rep  #nk-1
  mac  x0,y0,b  x:(r0)+,x0  y:(r4)+,y0

  macr  x0,y0,b
  clr  a
  move  x:(r6)+,a
  asr  a
  asr  a
  sub  b,a
  move  a,x0
  ; move  #mu,y1
  mpy  y1,x0,a
  mpy  y1,x1,b
  move  b,y1
  clr  b
  move  a,x1
  clr  a
move x:(r0)+,x0 y:(r4)+,a

do #nk,_cup

macr x0,x1,a x:(r0)+,x0 y:(r4)+,y0

move a,y:(r5)+

move y0,a

_cup

lua (r0)+n0,r0

lua (r4)+n4,r4

do #nk,_swone

move y:(r5)+,a

move a,y:(r4)+

_swone

move (r0)+

do #nk-1,_trfer

move x:(r0)+,a

move a,x:(r2)+

_trfer

move x:(r3)+,a

move a,x:(r2)+

do #nk,_swtwo

move x:(r2)+,a

move a,x:(r0)+

_swtwo

nop

_onepass

nop

mpy y1,x1,b
move  b,y1
clr   b

_morepasses

nop

do   #nk,_opcoeff
move  y:(r4)+,a
move  a,x:output_ptr

_opcoeff

nop

_allframes

nop

LMS_FINISH

rts

ENDSEC
APPENDIX D
ASSEMBLY CODE FOR AUTOCORRELATION

This section lists the assembly code for the autocorrelation of speech using Motorola DSP 56600.

**main.asm**

```assembly
move    #oldwin,r1
move    #newin,r2
do      #WINLEN,LoadInput
move    x:input_ptr,a1 ; input x:input_ptr boblib.io
move    a1,x:(r1)+ ; stated in command script
move    a1,x:(r2)+

LoadInput
jsr    acorr
jsr    storeop
nop
nop
nop
nop

mainLoop SOLA_FINISH
```

**acorr.asm**

```assembly
move    #newin,R2
```
move #>WINLEN,n1
move #>SCALER,x0
do n1,scale_down
move x:(r2),y1
mpy x0,y1,b
nop
move b1,x:(r2)+

scale_down
move #>WINLEN,a1
asl #1,a,a
sub #2,a ; length is 2*M-1 and 0 index
nop
move a1,n5
move x:xcorr_buffer_ptr,R5
lua (r5)+n5,r6
clr a ; clear sum
clr b ; clear sum
move #>0,r0
move #>WINLEN,n0
do n0,_MLoop ;outer loop
clr a
clr b
move (r0)+
move r0,a1
nop
move a1,n3
move n0,b
sub    a,b
nop
move   b1,n1
move   b1,n2
move   #oldwin,R1
move   #newin,R2
lua    (r1)+n1,r3
lua    (r2)+n2,r4
clr    a
clr    b
do     n3,_jLoop ;inner mac loop
move   x:(r1)+,x1 ; Second half of correlation
move   x:(r4)+,x0 ; Can be commented out since
mac    x0,x1,a    ; xcorrindex goes 1:WINLEN/2
move   x:(r2)+,y1 ; first half of correlation
move   x:(r3)+,y0
mac    y0,y1,b
    _jLoop
move   a,x:(r6)-
move   b,x:(r5)+
    _MLoop FINISH
nop
move   #newin,R2
move   #>WINLEN,n1
move   #>10,x1
do     n1, scale_up
move   x:(r2),y1
mpy x1,y1,b
nop
move b0,x:(r2)+

scale_up

storeop.asm

move #xcorr_buffer,r1 ; correlation vector returned by acorr()
move #WINLEN-1,n1
lua (r1)+n1,r1
do #WINLEN,STOREOP
move x:(r1)+,a
nop
move a,x:output_ptr

STOREOP
APPENDIX E
ASSEMBLY CODE FOR MODIFIED SIGNED LMS ALGORITHM

In this section, we present the assembly code for computing the linear prediction coefficients using the Modified Signed LMS algorithm.

; This program calculates the LPC coefficients using the modified signed LMS algorithm

nk    equ 4
mu    equ 0.2 ; 0.2/4
scaler equ 0.99
win    equ 176
len    equ 180
npass  equ 20
nframes equ 260

SECTION xram2

xdef input_ptr,input_ptr1,output_ptr
xdef acoeffs
xdef samp,newsamp,data,desired

org   x:$0
samp   ds nk
newsamp  ds  nk
data      ds  len
desired   ds  win
input_ptr dc  1
input_ptr1 dc  1
output_ptr dc  1

org y:$0
acoeffs  ds  nk

ENDSEC

SECTION mydurbin

xref input_ptr,input_ptr1,output_ptr
xref acoeffs
xref samp,newsamp,data,desired

org p:
lms_test

move  #-1,n0
move  n0,n4
move  #nk-1,m0
move  #len-1,m3
move  #win-1,m6
move m0,m2
move m0,m4
move m0,m5
move #samp,r0
move #newsamp,r2
move #data,r3
move #desired,r6
move #acoeffs,r4
move r4,r5

do #nframes,_allframes

do #len,_readfile
move x:input_ptr,a
move a,x:(r3)+
_readfile

do #win,_readdesired
move x:input_ptr1,a
move a,x:(r6)+
_readdesired

do #npass,_morepasses

move #mu,y1
do   #nk,_readdata
move x:(r3)+,a
move a,x:(r0)+

_readdata

do   #win,_onepass
clr  b    x:(r0)+,x0  y:(r4)+,y0
move  #$>scaler,x1

rep   #nk-1
mac  x0,y0,b    x:(r0)+,x0  y:(r4)+,y0

macr  x0,y0,b
clr  a
move x:(r6)+,a
asr  a
asr  a
sub  b,a
move a,x0
mpy  y1,x0,a
mpy  y1,x1,b
move b,y1
clr  b
move a,x1
clr  a

move x:(r0)+,x0  y:(r4)+,a
do      #nk, _cup

mpy    x0, x1, b
jmi    _negprod
cmp    #>$0001, b
jpl    _normal
add    #>$0001, a
move   x:(r0)+, x0   y:(r4)+, y0
jmp    _cup1

_negprod
    abs b
    cmp    #>$0001, b
    jpl    _normal
    sub    #>$0001, a
    move   x:(r0)+, x0   y:(r4)+, y0
    jmp    _cup1

_normal
    macr    x0, x1, a    x:(r0)+, x0   y:(r4)+, y0

_cup1
    move    a, y:(r5)+
    move    y0, a

_cup
    lua    (r0)+n0, r0
    lua    (r4)+n4, r4
do      #nk, _swone
move    y:(r5)+, a
move a, y:(r4)+

_swone
move (r0)+
do #nk-1, _trfer
move x:(r0)+, a
move a, x:(r2)+

_trfer
move x:(r3)+, a
move a, x:(r2)+
do #nk, _swtwo
move x:(r2)+, a
move a, x:(r0)+

_swtwo
nop

_onepass
nop

mpy y1, x1, b
move b, y1
clr b

_morepasses
nop

do #nk, _opcoeff
move y:(r4)+, a
move a, x:output_ptr

_opcoeff
n nop

_allframes

n nop

LMS_FINISH

rts

ENDSEC
REFERENCES


BIOGRAPHICAL SKETCH

Adnan H. Sabuwala was born in Bombay, India, on 18th November 1978. He completed his schooling from the Versova Welfare High School and joined the Sathaye College, Vile Parle, for his high school studies. In July of 1996 he was admitted to the Indian Institute of Technology, Bombay (IIT-B), to the Department of Electrical Engineering. He graduated with a B.Tech degree from the IIT in August 2000 and joined the Department of Electrical and Computer Engineering at the University of Florida in Fall 2000. Since January 2001, he has been working as a research assistant for Dr. John G. Harris in the Computational Neuro-Engineering Lab where he completed his master’s thesis on “Towards a Real-Time Implementation of Loudness Enhancement Algorithms on a Motorola DSP 56600.”