AN MOS-LSI AUTOCORRELATION LINEAR PREDICTION SYSTEM

by

P. J. Hurst

Memorandum No. UCB/ERL M83/51

26 August 1983

(cover)

AN MOS-LSI AUTOCORRELATION LINEAR PREDICTION SYSTEM

by

Paul James Hurst

Memorandum No. UCB/ERL M83/51

26 August 1983

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

An MOS-LSI Autocorrelation Linear Prediction System

By

Paul James Hurst

B.S. (University of California) 1977
M.S. (University of California) 1979

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Engineering

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: Robert W. Brodersen 8/24/83
Chairman Date
David G. Messerschmitt 8/25/83
Jan A. Rabaey 26 Aug 1983.

.....

An MOS-LSI Autocorrelation Linear Prediction System

Paul James Hurst

Ph. D. EECS

Robert W. Brodersen
Chairman of Committee

ABSTRACT

Integration of an autocorrelation linear prediction system for speech analysis will be described. The system is composed of both analog, switched-capacitor and digital circuitry. A switched-capacitor autocorrelator is the key component of the system. Accuracy requirements both in the analog and digital processing domains are carefully considered for the speech application. Practical aspects of speech processing are discussed.

The autocorrelator was fabricated on an analog/digital compatible CMOS process. The design, layout, and fabrication of the integrated circuit are described. Experimental results are presented for the autocorrelator integrated circuit as well as the complete linear prediction system.

This research was sponsored by DAPRPA Grant 1-482427-25978.

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to Professor Brodersen for his support and encouragement throughout the course of my graduate studies. I am very lucky that he chose me to work as a research assistant under him, and I have learned a great deal about integrated ciruit design and signal processing from him.

I would also like to thank the EECS faculty and fellow graduate students in the integrated circuits group and speech group. They are all very knowledgeable, friendly, and helpful. Special thanks go to Ron Fellman for all his help, Ron Kaneshiro and Tat Choi for their help in the fabrication lab, and Hy Murveit and Asa Romberger for providing invaluable programming assistance.

My most important source of recreation during my graduate studies has been playing softball with the RSB softball team (record 46-20). The RSB team members are all quality softball players and good guys, and I thank them all.

Finally, I would like to thank my mother and sister for sticking with me through this long research project, my father for his support, and Jane Berryhill for being so helpful and understanding this past year.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Recently, synthetic speech has appeared in many consumer products such as automobiles, educational games for children, chess games, microwave ovens, and alarm clocks. The successful integration of speech synthesis algorithms in silicon technology has provided manufacturers with low cost speech synthesis systems. But the inverse process, speech analysis, has remained a task handled by large computer systems or expensive general purpose signal processing integrated circuits, often not operating in real time. Vocoders, store-and-forward message systems, and speech recognition systems will remain prohibitively expensive unless low cost analysis systems are made available.

A real-time linear predictive speech analysis system is described in this thesis. The analysis system employs a mix of switched-capacitor analog circuitry and MOS digital circuitry. The heart of the system is a switched-capacitor autocorrelation integrated circuit (IC). An automatic gain control circuit which reduces the dynamic range of the incoming speech precedes the autocorrelator. A small microprocessor system converts the autocorrelator's output into linear prediction model coefficients.

The autocorrelator itself was integrated as the final stage of this research. The chapters follow the design of the system from start to finish, roughly in chronological order. Chapter 2 covers the speech analysis theory which forms the basis of the IC. Chapter 3 presents circuits which are used to implement the system. The operation of the circuits is explained. In chapter

4, the circuits chosen in chapter 3 are discussed in detail. Non-idealities due to integration of the circuits in MOS technology are considered. Various system-level design decisions are made. The design, layout, and processing of the autocorrelator IC are discussed. Chapter 5 presents the results obtained from the breadboarded system and from the autocorrelator IC. Conclusions and suggestions are made in chapter 6.

CHAPTER 2

LPC THEORY AND APPLICATIONS

Speech is a verbal form of communication. The rate at which information is transmitted during a conversation is approximately 100 bits per second [1]. Hence, speech could theoretically be encoded in such a way as to convey all the information while transmitting only 100 bits per second. One simple digital transmission scheme used today, 8 bit quantization of speech sampled at 8kHz, requires 64k bits per second. This is 640 times larger than the information rate, the discrepancy reflects how inefficient direct transmission of the speech time waveform is.

The speech waveform is a sequence of sounds. In the case of vowels, each sound is a periodic repetition of a distinct waveform. Speech analysis attempts to eliminate the transmission of the redundant information in the time waveform. Many different techniques have been applied to speech analysis, some meeting with more success than others. Mathematical tractability and automatic implementation of analysis algorithms are very desirable properties. One analysis method, linear predictive coding (LPC), has these properties and has been widely applied to the speech analysis problem. Autocorrelation LPC will be described in detail in the following sections.

2.1. A MODEL FOR SPEECH

To understand speech analysis, we must first understand how we produce speech sounds. A sound is produced as follows (see figure 2.1):

Air is forced out of the lungs by muscle contraction. As the air passes through the vocal cords, it is either allowed to flow through unrestricted (unvoiced sounds), or it is transformed into short bursts of air by the periodic opening and closing of the vocal cords (voiced sounds). The air then passes through the vocal tract. The shape of the vocal tract and position of the lips and tongue determine the resonant frequencies (formants) of the sound. The vocal tract acts as an acoustic filter, spectrally shaping the air pressure wave as it passes from the vocal cords to the lips.

A simple model for the speech production process described is shown in figure 2.1. The unrestricted flow of air at the vocal cords is modeled as white noise. The periodic bursts of air at the vocal cords are modeled as periodic

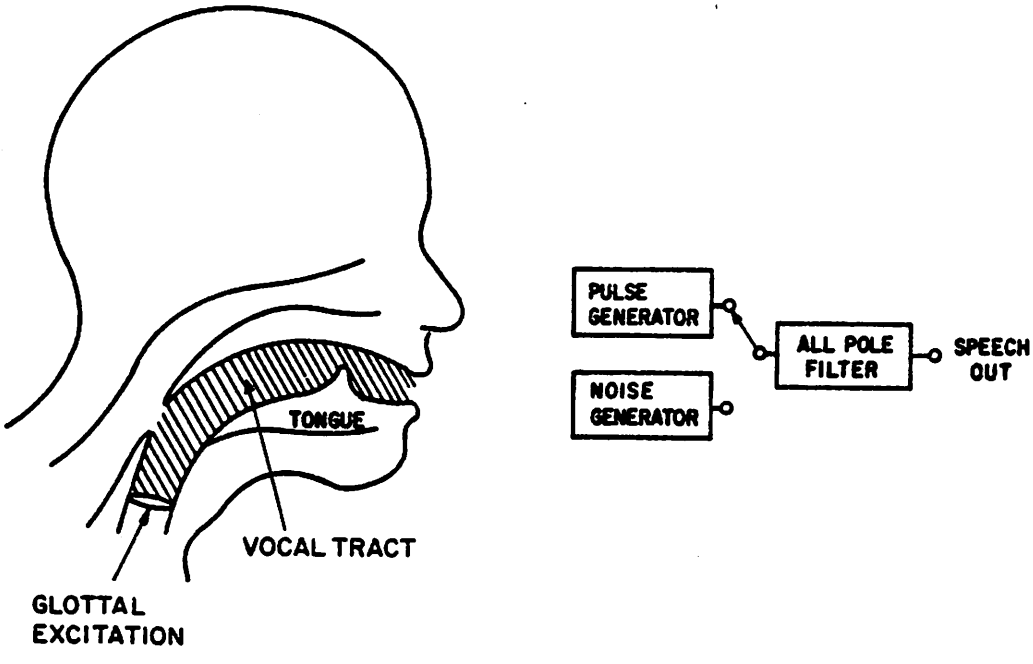


Figure 2.1. Cross-section of the vocal tract (left) and a simple model for the speech production process (right)

impulses. The period of the impulses is called the pitch period. The vocal tract is modeled as a time-varying filter.

Most speech synthesizers are realizations of this simple model. The excitation waveform and filter parameters are changed at a rate of 20Hz to 100Hz. Each segment of synthetic speech produced by a given set of parameters is called a frame of speech. The update rate or frame rate corresponds to the rate at which different sounds occur in conversational speech.

In speech analysis, the inverse process is performed. The continuously changing speech waveform is broken into pitch and vocal tract information each frame. There are many different approaches to speech analysis [1]; one particular method will be described in detail.

2.2. LPC THEORY

With LPC, the present speech sample is modeled as a linear combination of the past p speech samples plus an unknown input

$$s(n) = \sum_{i=1}^{p} a_i s(n-i) + bu(n);$$

$s(n)$ is the sample of the speech waveform $s(t)$ at time $t=nT_s$ (T_s is the sampling period), $u(n)$ is the input (glottal excitation), and a_i is the i^{th} LPC coefficient. In the frequency domain, this prediction equation gives an all-pole transfer function

$$G(z) = \frac{S(z)}{U(z)} = \frac{b}{1 - \sum_{i=1}^{p} a_i z^{-i}}; \qquad (2.1)$$

$S(z)$ and $U(z)$ are the z-transforms of $s(n)$ and $u(n)$, respectively. For a given sound, LPC finds an all-pole filter $G(z)$ which models the vocal tract transfer function. The LPC model can be viewed physically as a cascade of p acoustic tubes of constant length and differing cross-sectional area [2]. The

cross-sectional areas are related to the LPC coefficients (see section 2.4). An LPC transfer function is plotted in figure 2.2, superimposed on the FFT of the same sound. The LPC all-pole filter attempts to model the resonances of the vocal tract (see section 2.5). Each resonance modeled requires a complex conjugate pair of poles. Typically, vocal tract resonances are spaced 1kHz apart. So if the speech waveform is sampled at a frequency $f_s = \frac{1}{T_s}$ so that we are modeling the vocal tract from 0Hz to $\frac{f_s}{2}$ we need a minimum of $p = \frac{f_s}{1kHz}$ poles to model the vocal tract resonances. One or two more poles are often included to model the non-flat spectrum of the glottal waveform.

In LPC speech analysis, we are given the speech waveform $s(n)$ and want to determine the LPC coefficients, a_i, $1 \le i \le p$. Since we have no prior information about the excitation $u(n)$, we predict the present sample $s(n)$ from the

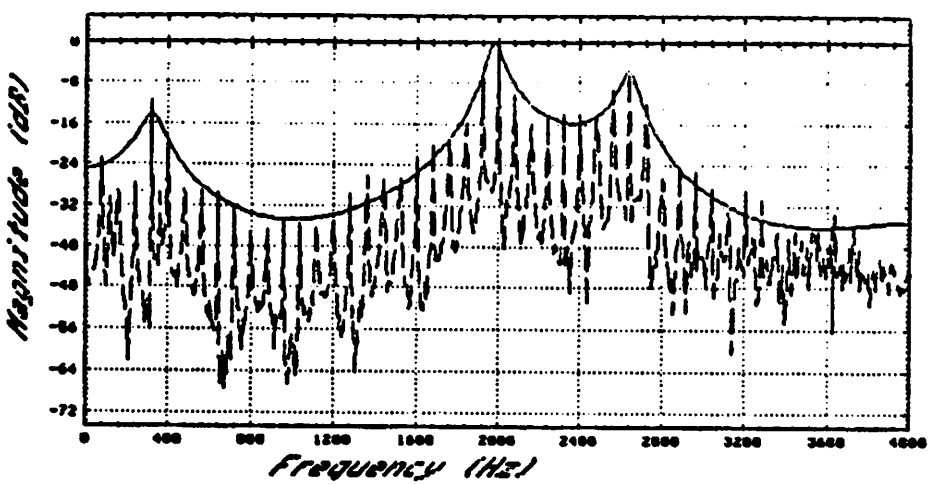


Figure 2.2 LPC fit (solid line) to a speech spectrum (broken line)

past p samples

$$\bar{s}(n)=\sum_{i=1}^{p} a_i s(n-i).$$

The error between the actual speech sample $s(n)$ and the predicted value $\bar{s}(n)$ is the prediction error $e(n)$

$$e(n)=s(n)-\bar{s}(n). \tag{2.2}$$

There are various error minimization criterion which might be used to find the LPC coefficients. One is to minimize the total squared error E_p

$$E_p = \sum_{n=-\infty}^{\infty} e^2(n) = \sum_{n=-\infty}^{\infty} [s(n) - \bar{s}(n)]^2. \tag{2.3}$$

To minimize E_p, we differentiate E_p with respect to a_k and set $\frac{\partial E_p}{\partial a_k} = 0$ for each k, $1 \le k \le p$. This yields p linear equations

$$\sum_{i=1}^{p} a_i \sum_{n=-\infty}^{\infty} s(n-i)s(n-k) = \sum_{n=-\infty}^{\infty} s(n)s(n-k) \quad \text{for } 1 \le k \le p$$

which can be rewritten

$$\sum_{i=1}^{p} a_i R(k-i)=R(k) \quad 1 \le k \le p \tag{2.4}$$

where

$$R(k) = \sum_{n=-\infty}^{\infty} s(n)s(n-k) \tag{2.5}$$

is the autocorrelation function of the speech signal $s(n)$. The solution of (2.4) for the LPC coefficients requires solution of p linear equations in p unknowns. This will be discussed in section 2.4.

2.3. WINDOWING

Note that equation (2.3) minimizes the error over all time. If $s(n)$ is a constant sound, $R(k)$ will be that sound's autocorrelation function and solving equation (2.4) will give LPC coefficients for that sound. But for conversational speech, $s(n)$ is a continuously varying sequence of sounds. Therefore, we

would like to compute $R(k)$ separately for each frame (which should correspond to only one sound if the frame size is chosen properly). Then we can find the LPC coefficients for that particular frame. To achieve this, we window the speech waveform. That is, we multiply the speech waveform by a window function w(n) which weights the present frame more heavily than other frames. This windowing gives a new time waveform $s_w(n,j)$

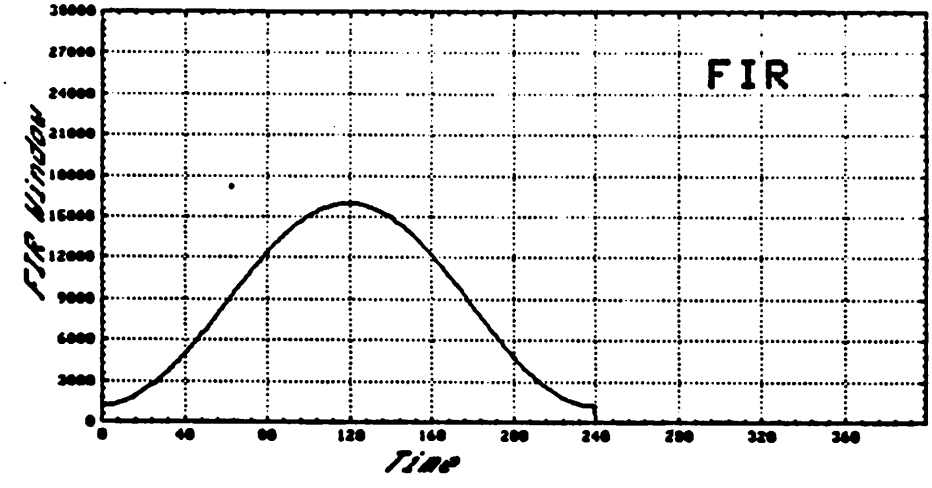
$$s_w(n,j) = s(n)w(n-j)$$

where j accounts for the positioning of the window. Then this windowed speech $s_w(n,j)$ is used in the autocorrelation computation, equation (2.5), to give

$$R(k,j)= \sum_{n=-\infty}^{\infty} s_w(n,j)s_w(n-k,j)= \sum_{n=-\infty}^{\infty} s(n)w(n-j)s(n-k)w(n-k-j). \quad (2.6)$$

Since the application of a window results in an autocorrelation function based on short segments of the speech signal, (2.6) is often referred to as the short-time autocorrelation function. Since equation (2.6) is the only autocorrelation function which will be discussed from here on, it will be simply referred to as the autocorrelation function.

The time window w(n) can be of finite duration (FIR) or infinite duration (IIR) (figure 2.3). In either case, the window should have no discontinuities, going smoothly to zero at the window's endpoints. This minimizes the spectral smearing of the formants due to the window [1]. If an FIR window is chosen, equation (2.6) may be implemented by first computing the sequence $s_w(n,j)=s(n)w(n-j)$, then using the windowed speech in equation (2.6). This approach is straightforward; requiring permanent storage (ROM) for the sequence $w(n)$, temporary memory (RAM) for storing the windowed sequence $s_w(n,j)$, a multiplier, and a summer [3].



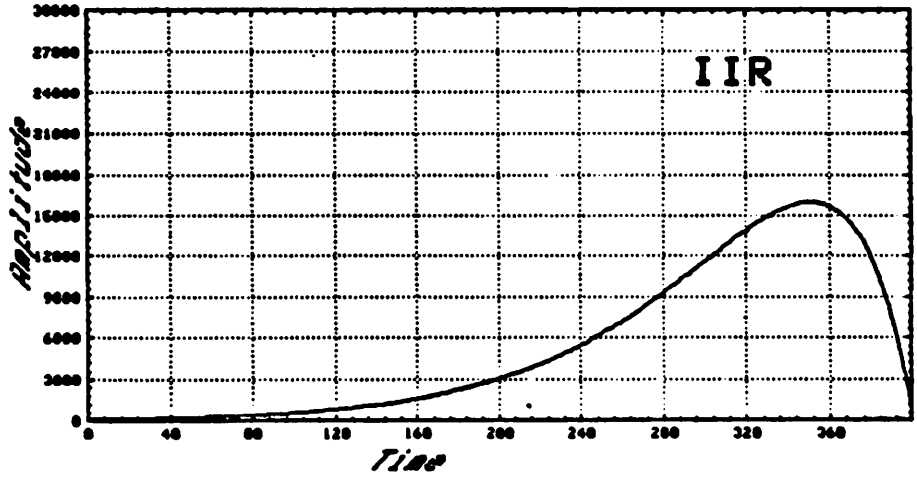


Figure 2.3 top: FIR time window (e.g. a Hamming window)
bottom: IIR time window suggested by Barnwell

The windowing and summation in equation (2.6) can be implemented as a filter following a multiplier for both the FIR and IIR windows. This can be seen by defining a time-reversed version of the window

$$h(n) = w(-n).$$

Then (2.6) becomes

$$R(k,j) = \sum_{n=-\infty}^{\infty} s(n)h(j-n)s(n-k)h(j-n+k). \tag{2.7}$$

If we define

$$s'_k(n) = s(n)s(n-k)$$

$$h'_k(n) = h(n)h(n+k)$$

and substitute these into (2.7), we have

$$R(k,j) = \sum_{n=-\infty}^{\infty} s'_k(n)h'_k(j-n).$$

We can see that $R(k,j)$ is the convolution of $s'_k(n)$ with $h'_k(n)$. So if we can produce the time sequence $s'_k(n)$ and pass it through the linear, time invariant filter with impulse response $h'_k(n)$, the output of the filter at time j will be the autocorrelation function for lag k, $R(k,j)$.

Producing the sequence $s'_k(n)$ requires only multiplication and delay. To find the z-domain transfer function for the filter with impulse response

$$h'_k(n) = h(n)h(n+k),$$

we recall that multiplication in the time domain corresponds to convolution in the frequency domain. So if we denote the z-transforms of the sequences

$$h(n) \rightarrow H(z)$$

and

$$h'_k(n) \rightarrow H'_k(z),$$

then the shifted sequence $h(n+k)$ has the transform

$$h(n+k) \rightarrow z^k H(z).$$

Now $H'_k(z)$ can be found [4]

$$H'_k(z) = H(z)*z^k H(z) = \frac{1}{2\pi j}\int H(v)H(\frac{z}{v})v^{k-1}dv. \tag{2.8}$$

So once a window $w(n)=h(-n)$ has been chosen, $H(z)$ is determined and $H'_k(z)$ can be found by integration in the complex plane (see appendix A).

For example, let us take

$$w(n) = \begin{cases} (1-n)\alpha^{-n} & n\leq 0 \\ 0 & n>0 \end{cases} \tag{2.9}$$

where $0<\alpha<1$. The corresponding $h(n)$ is

$$h(n) = \begin{cases} (1+n)\alpha^{n} & n\geq 0 \\ 0 & n<0 \end{cases}$$

with z-transform

$$H(z) = \sum_{n=0}^{\infty} h(n)z^{-n} = \frac{1}{(1-\alpha z^{-1})^2}. \tag{2.10}$$

$H(z)$ is a low pass filter with two coincident real poles. Substituting this $H(z)$ into (2.8) and evaluating the integral gives

$$H'_k(z) = \frac{(k+1)\alpha^k+(1-k)\alpha^{k+2}z^{-1}}{(1-\alpha^2 z^{-1})^3} \tag{2.11}$$

(see appendix A). Note that each value of k corresponds to a different filter.

Filter Specifications			
lag	poles	zero	$\frac{H'_k(1)}{H'_0(1)}$
k=0	0.9604	-0.9604	1.000000
k=1	0.9604	0.0000	0.999796
k=2	0.9604	0.3201	0.999184
k=3	0.9604	0.4802	0.998215
k=4	0.9604	0.5762	0.996888
k=5	0.9604	0.6403	0.995205
k=6	0.9604	0.6860	0.993216
k=7	0.9604	0.7203	0.990869
k=8	0.9604	0.7470	0.988268
k=9	0.9604	0.7683	0.985309

Table 2.1. Filter specifications for $H'_k(z)$

All the filters have three poles at $z=\alpha^2$ and one real zero. Barnwell has found that for an 8kHz sampling rate, α=0.98 is the best choice [5]. This window gave spectral errors comparable to a 240-point FIR Hamming window. For the case α=0.98, the pole and zero locations and the relative DC gains $\frac{H'_k(1)}{H'_0(1)}$ are tabulated in table 2.1. The magnitude response of the filters is shown in figure 2.4. Note the effect of the zero. For all the filters, the -3dB frequency is 25Hz.

A block diagram of an autocorrelator employing the IIR window of equation (2.9) is shown in figure 2.5. The autocorrelator requires a p stage delay line, multipliers, and filters. Note that the temporary memory storage (RAM) required is only $4(p+1)+p$. The multiplier in the IIR autocorrelator must be approximately twice as fast as the FIR autocorrelator's multiplier. The IIR autocorrelator outputs values for $R(k)$ every sample, this is useful in some applications (see section 2.7).

The delay line, multipliers, and filters of figure 2.5 can all be implemented as analog or digital circuits. A mix of analog and digital circuits will be used in the autocorrelator described in later chapters.

2.4. LPC COEFFICIENT COMPUTATION - DURBIN'S RECURSION

Once the autocorrelation values for a particular frame have been computed, the set of linear equations (2.4) must be solved to determine the LPC coefficients. The equations can be compactly written in matrix form

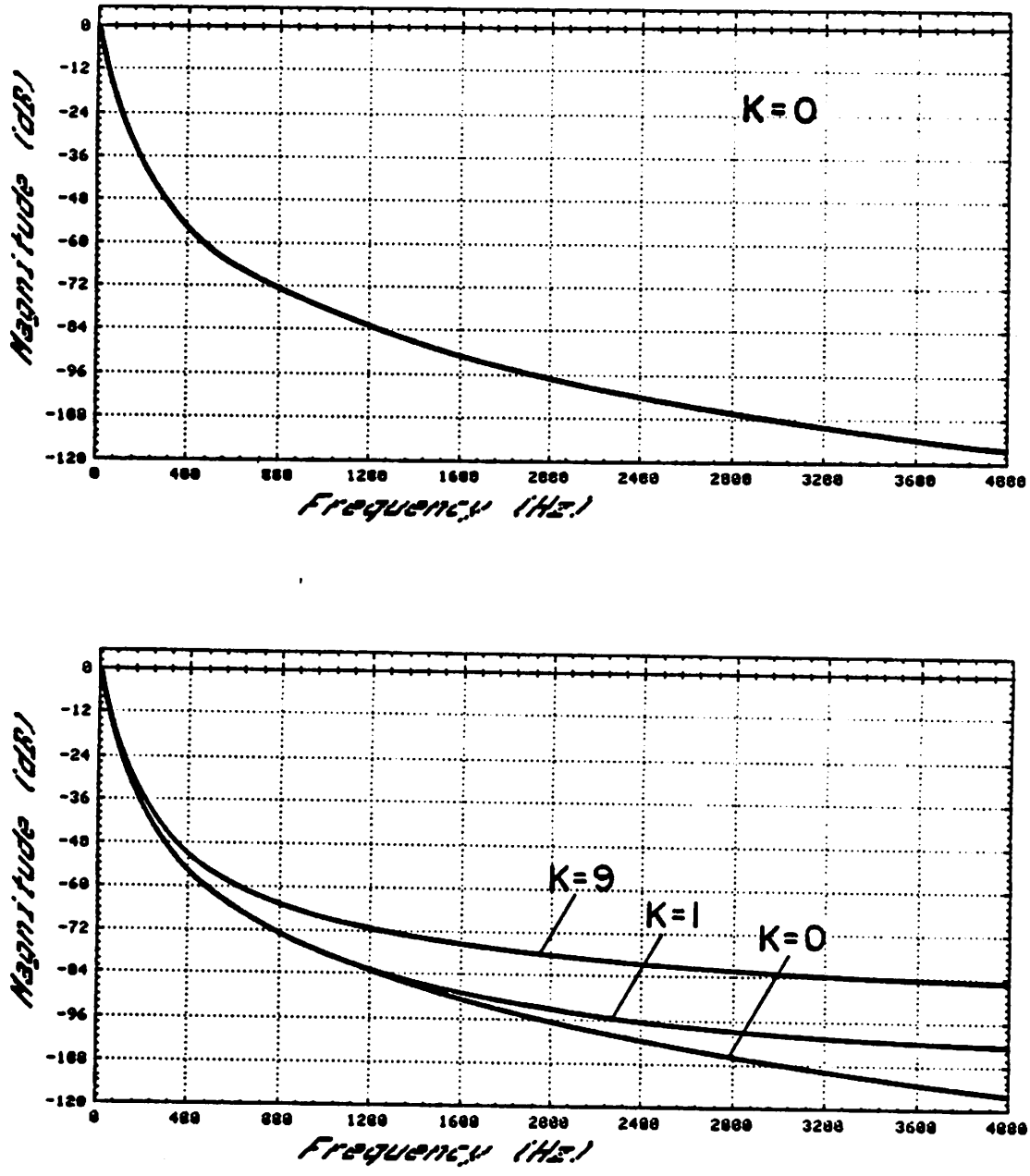


Figure 2.4 top: Magnitude response of filter $H'_0(z)$
bottom: Magnitude response of filters $H'_0(z)$, $H'_1(z)$, and $H'_9(z)$

$$\begin{bmatrix} R(0) & R(1) & R(2) & \cdots & R(p-1) \\ R(1) & R(0) & R(1) & \cdots & \cdot \\ R(2) & R(1) & R(0) & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ R(p-1) & R(p-2) & \cdot & \cdots & R(0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \cdot \\ \cdot \\ a_p \end{bmatrix} = \begin{bmatrix} R(1) \\ R(2) \\ R(3) \\ \cdot \\ \cdot \\ R(p) \end{bmatrix} \qquad (2.12)$$

or

$$\mathbf{R} \cdot \mathbf{a} = \mathbf{r}$$

where $\mathbf{R}$ is the $p \times p$ autocorrelation matrix, $\mathbf{a}$ is the $p \times 1$ vector of LPC coefficients, and $\mathbf{r}$ is the $p \times 1$ vector of autocorrelation values. In equation (2.12), the relation $R(k)=R(-k)$ has been used, and the time dependence of $R(k)$ has been dropped for simplicity. The a_i's could be found by inverting the $\mathbf{R}$ matrix and multiplying both sides by $\mathbf{R}^{-1}$. This would require $\frac{p^3}{3} + O(p^2)$ operations (multiplications or divisions) and p^2 storage locations. Fortunately, the autocorrelation matrix is symmetric, and the elements along any diagonal are identical. Such a matrix is Toeplitz, and Levinson and Durbin derived efficient recursion relations for solving such linear equations [6],[7]. The Durbin algorithm is the faster algorithm, requiring $p^2 + O(p)$ operations and $2p$ storage locations.

Durbin's recursive solution to (2.12) is

start: $E_0 = R(0)$

$m = 0$

loop: $m = m + 1$

$$k_m = \frac{R(m) + \sum_{i=1}^{m-1} a_i^{m-1} R(m-i)}{E_{m-1}}$$

$a_m^m = k_m$

$a_i^m = a_i^{m-1} - k_m a_{m-i}^{m-1} \quad 1 \le i \le m-1$

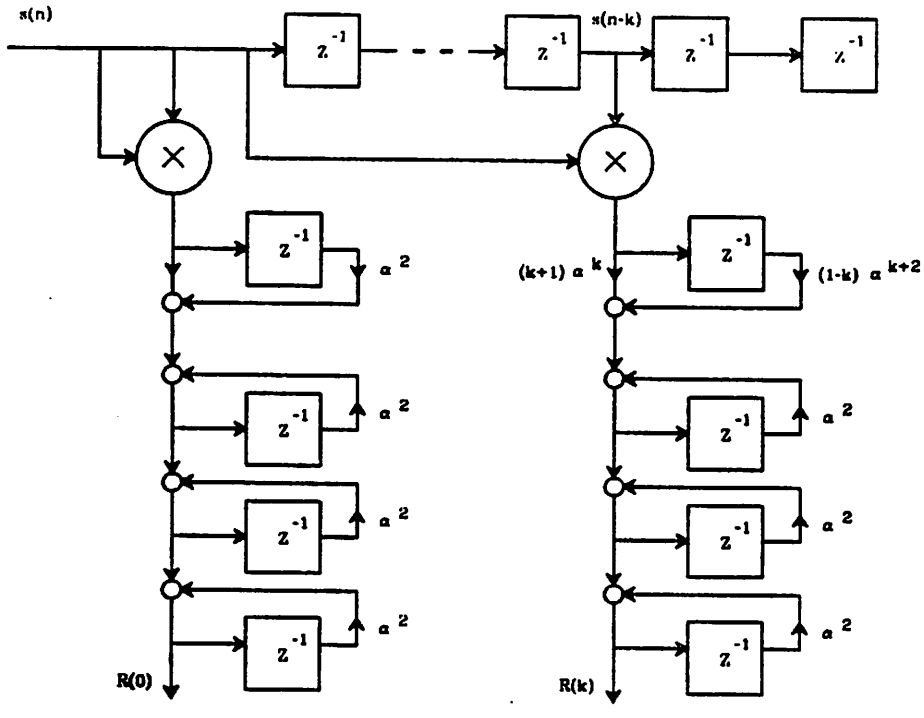


Figure 2.5. Block diagram of autocorrelator employing the IIR window of equation (2.9)

$$E_m=(1-k_m^2)E_{m-1} \tag{2.13}$$

if($m<p$) *go to loop*;

else stop;

a_i^m is the i^{th} LPC coefficient for an m-pole model. The loop evaluates the LPC coefficients for all models of order less than or equal to p. The algorithm terminates when the model of order p has been found. E_p, the total squared error (equation (2.3)), is computed in the Durbin algorithm (equation (2.13)). The algorithm computes the LPC coefficients as well as the reflection coefficients (the k_i's). The reflection coefficients have a physical interpretation. As was mentioned in section 2.2, the LPC model of the vocal tract can be viewed as a cascade of p acoustic tubes of equal length and differing cross-sectional area. If the i^{th} acoustic tube of has area A_i, then

$$k_i = \frac{A_{i-1}-A_i}{A_{i-1}+A_i}$$

In low-bit rate transmission of speech, the reflection coefficients are quantized and transmitted because the LPC model is less sensitive to quantization of the reflection coefficients than to quantization of either the autocorrelation values or the LPC coefficients.

The LPC analysis fails to give a stable all-pole filter when $|k_i|>1$ for any i [2]. If $|k_i|>1$, then $E_i<0$ which contradicts the definition of E_i as the total squared error which must always be non-negative (see equation (2.13)). In terms of acoustic tubes, $|k_i|>1$ corresponds to either $A_i<0$ or $A_{i-1}<0$; neither makes sense physically. It has been shown that the autocorrelation method always produces a stable filter if all computations are performed with infinite precision [8]. In practice, the use of finite precision arithmetic can result in an unstable filter.

At this point, it is worth noting that the solution of (2.12) for the LPC coefficients is not affected if all the autocorrelation values $R(k)$ are scaled by a non-zero constant. For example, if we replace each $R(k)$ in (2.12) with $cR(k)$, then the c factors out to give $c\mathbf{Ra}=c\mathbf{r}$ which reduces to $\mathbf{Ra}=\mathbf{r}$. This observation is of interest in later chapters.

The gain in the LPC model (b in equation (2.1)) has not been determined because it is associated with the input $u(n)$ which is unknown. But we may assume the input $u(n)$ is either an impulse $\delta(n)$ (for voiced speech) or samples of a white noise process (for unvoiced speech) with the property that the energy in $u(n)$ is unity. With that constraint, and the reasonable requirement that the energy in the LPC model equal the energy in the speech, it can be shown [8] that

$$b^2 = E_p = R(0) + \sum_{k=1}^{p} a_k R(k).$$

2.5. FREQUENCY DOMAIN INTERPRETATION OF LPC

The popularity and success of LPC is due to LPC's accurate modeling of the resonances of the vocal tract. Early experiments determined that the spectral peaks were perceptually far more important than spectral valleys [9]. The LPC error criterion will now be considered in the frequency domain where the reason for the accurate resonance modeling can be seen.

Recall the definition of the prediction error in equation (2.2)

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

which transforms to

$$E(z) = \left[1 - \sum_{i=1}^{p} a_i z^{-i}\right] S(z) = A(z) S(z)$$

where A(z) is the denominator of the vocal tract model (equation (2.1))

$$G(z) = \frac{b}{A(z)} = \frac{b}{1 - \sum_{i=1}^{p} a_i z^{-i}}.$$

Now the total squared error of (2.3) can be evaluated in the frequency domain by invoking Parseval's Theorem

$$E_p = \sum_{n=-\infty}^{\infty} e^2(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} |E(e^{j\omega})|^2 d\omega = \frac{1}{2\pi} \int_{-\pi}^{\pi} |S(e^{j\omega})|^2 |A(e^{j\omega})|^2 d\omega$$

which can be rewritten

$$E_p = \frac{b^2}{2\pi} \int_{-\pi}^{\pi} \frac{|S(e^{j\omega})|^2}{|G(e^{j\omega})|^2} d\omega. \tag{2.14}$$

From this equation, we can see that E_p depends on the integrated ratio of two spectra. Points where $|G|$ is smaller than $|S|$ make a larger contribution to the total error than points where $|G|$ is larger than $|S|$. Therefore, the error minimization (2.14) can be viewed in the frequency domain as one which

weights the modeling of spectral peaks more than valleys. So LPC models the resonances of the vocal tract very well.

2.6. PRACTICAL CONSIDERATIONS IN SPEECH ANALYSIS

2.6.1. PRE-EMPHASIS

The speech production process was explained earlier. A long-time spectrum for continuous speech shows a roll-off of approximately -6dB/octave above 500Hz [10]. This is typical of voiced sounds. The speech spectrum is the product of the excitation's spectrum, the vocal tract transfer function, and the lip-to-air radiation transfer function. The glottal excitation for voiced sounds has a -12dB/octave slope at high frequencies. The vocal tract has unity transfer gain at each resonance and the lip-to-air radiation has a +6dB/octave characteristic at high frequencies. The net effect is a -6dB/octave roll-off at high frequencies as shown in figure 2.6.

To achieve the most accurate model of the spectral peaks, it is advantageous for all spectral peaks to have the same magnitude. Otherwise, the LPC error criterion will cause the lower amplitude peaks to be modeled much more poorly that the higher amplitude peaks. Therefore, the speech waveform is pre-emphasized before LPC analysis is begun. The pre-emphasis is in the form of a simple, one zero filter (zero at 500Hz) which removes the -6dB/octave roll-off present in typical voiced speech spectra. Pre-emphasis reduces the likelihood of instability in the LPC model $G(z)$ [2].

Ideally, pre-emphasis is applied only to voiced speech since unvoiced speech does not have the -6dB/octave roll-off characteristic (because the excitation for unvoiced speech doesn't have the -12dB/octave roll-off). But usually a constant pre-emphasis filter is used for all speech, voiced and

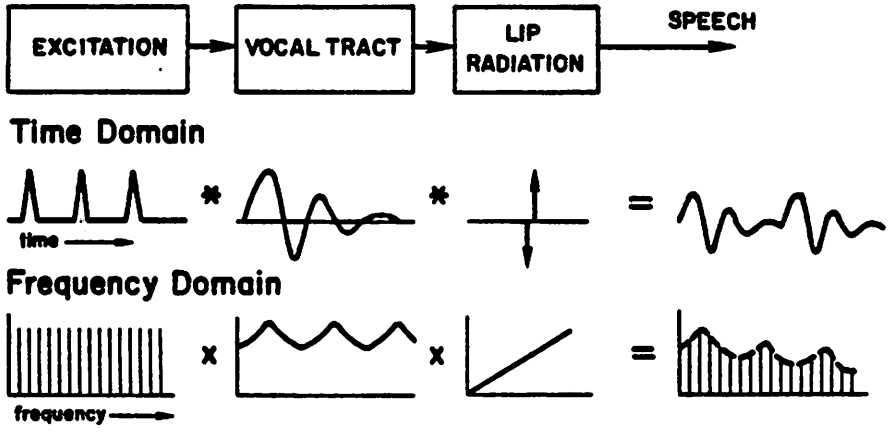


Figure 2.6 Construction of the speech waveform in time and frequency domains

unvoiced. The main reason is that unvoiced speech has a very simple spectral shape associated with it, hence modeling is fairly simple and is not hampered by pre-emphasis. Furthermore, the voiced/unvoiced decision is not an easy one; and the modeling error resulting from not pre-emphasizing a voiced segment of speech is much larger than the error resulting from pre-emphasizing an unvoiced segment. Also, adaptive pre-emphasis requires additional analysis of the speech to determine the optimal pre-emphasis filter.

2.6.2. DYNAMIC RANGE

Another concern is the dynamic range of the incoming speech. With a particular speaker, the speech has a dynamic range on the order of 30dB. When different speakers and slight variations in the speaker-to-microphone distance are allowed, the dynamic range increases to 40dB. If the incoming

speech is quantized by a linear analog-to-digital converter (ADC), 7 bits of resolution would be necessary. More bits would be used in practice to minimize clipping and quantization noise; a 10 or 12 bit ADC is commonly utilized.

To eliminate the need for a 10 or 12 bit ADC at the input, an automatic gain control (AGC) circuit should be used. A properly designed AGC would decrease the dynamic range of the speech considerably without affecting the speech's spectral characteristics and thereby relax the ADC requirements. An 8 bit ADC is sufficient if an AGC is employed (see chapter 4). The AGC would scale all the speech samples in a given frame and hence the frame's autocorrelation values $R(k)$. As was noted in section 2.4, scaling all the autocorrelation values does not change the resulting LPC coefficients.

2.6.3. ANTI-ALIAS FILTER

Before sampled-data processing can begin, the speech must be bandlimited to $\frac{f_s}{2}$. If the bandlimiting filter causes a large attenuation of the speech spectrum below $\frac{f_s}{2}$, some of the LPC parameters will be wasted modeling this attenuation near $\frac{f_s}{2}$. This is undesirable. So a high order filter or elliptic filter with a sharp roll-off characteristics and cutoff frequency close to $\frac{f_s}{2}$ is preferred. Unfortunately, such a filter introduces significant phase nonlinearity. But as can be seen from equation (2.14), the LPC error criterion depends only on the magnitude of the spectrum of the incoming speech - not on the phase.

2.7. OTHER APPLICATIONS OF THE AUTOCORRELATOR

Autocorrelators have many uses in speech processing. In addition to LPC analysis for speech transmission, some speech recognition systems use an autocorrelation-based LPC distance measurement between sounds. The Itakura-Saito measure [11] requires storage of a reference template for each sound, the template is the sound's LPC coefficient vector $\mathbf{a}_{ref}$ (see equation (2.12)). When an unknown sound is uttered, its autocorrelation matrix **R** is found (see equation (2.12)), and the distance between the unknown sound and each reference template is computed as

$$d = 10\log\left[\frac{\mathbf{a}_{ref}^T \mathbf{R}\mathbf{a}_{ref}}{\mathbf{a}^T \mathbf{R}\mathbf{a}}\right]. \qquad (2.15)$$

$\mathbf{a}^T$ is the transpose of **a**. The denominator is the minimum LPC error E_p achievable for a p-pole model of the unknown frame (see equation (2.3)) [8]. The numerator represents the total error E'_p obtained if the unknown frame of speech is modeled by the stored template $\mathbf{a}_{ref}$. Therefore the numerator will always be greater than or equal to the denominator, so the ratio is greater than or equal to one and therefore $d \geq 0$. This distance measure will equal zero only if $\mathbf{a}_{ref} = \mathbf{a}$, i.e. when the unknown speech and the reference template have identical LPC spectra.

Pitch detection is an extremely difficult problem, and pitch detection errors are more often responsible for poor quality synthetic speech than are vocal tract modeling errors. The best pitch detectors rely on a variety of pitch detection techniques, one technique is the autocorrelation technique [12]. The autocorrelation function, if computed over a time much larger than the pitch period, will be periodic with period equal to the pitch period. This periodicity is easily measured; a simple peak-to-peak distance measurement on the autocorrelation function reveals the pitch period. The autocorrelation

function should be computed every sample to find the pitch period, and an IIR autocorrelator is an excellent choice for such an application. Accuracy of the autocorrelation values is not crucial, so simpler filters (e.g. one pole filters) for $H'_k(z)$ could be used.

The voiced/unvoiced/silence decision used in pitch detectors, speech recognition systems, adaptive pre-emphasis subsystems, and telephone multiplexing systems can be made very accurately using the first few autocorrelation values $R(0)$, $R(1)$, $R(2)$ and a zero-crossing detector [13].

CHAPTER 3

CIRCUITS FOR THE AUTOCORRELATION LPC SYSTEM

Circuitry for implementing a nine pole autocorrelation LPC analysis system will be described in this chapter. The autocorrelator employs the IIR window discussed in chapter 2. The system is a mix of analog and digital circuitry. A completely digital implementation of the LPC system - excluding the input antialias filter - would be possible, either by designing a custom integrated circuit (IC) or by programming a general purpose digital signal processor. But combining analog MOS switched-capacitor techniques and digital MOS circuitry potentially offers a lower power, smaller area integrated circuit approach.

In this chapter, circuits are presented for each major block of the system, and their operation is described. So as not to cloud the functional descriptions, non-idealities associated with the IC implementation are not considered until next chapter.

3.1. SYSTEM OVERVIEW

A block diagram of the complete system is shown in figure 3.1. The speech first passes through a bandlimiting, low pass filter. The filter is a fifth order elliptic which has a -3dB cutoff frequency of 3.4kHz, 1dB of passband ripple, and includes a zero at 500Hz for pre-emphasis. The bandlimited, pre-emphasized speech passes through an automatic gain control circuit (AGC) and then to the autocorrelator where the autocorrelation values for the present frame are computed. The ten autocorrelation values are transformed to LPC coefficients by a small microprocessor system. The

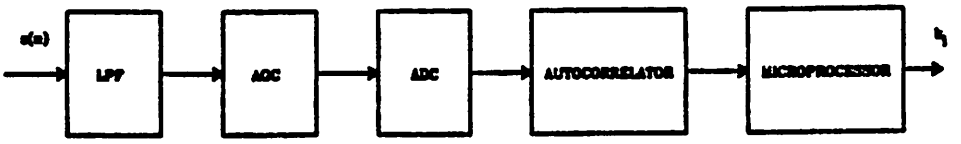


Figure 3.1. LPC system block diagram

speech is sampled at a rate of f_s=8kHz; the autocorrelation values are sampled at a frame rate of 80Hz.

3.2. THE AUTOCORRELATOR

3.2.1. CIRCUIT OPTIONS

An autocorrelator using the IIR window of equation (2.9) requires delay, multiplication, and filtering. An analog delay line of length p can be constructed by cascading p switched-capacitor (SC) sample-and-hold circuits. Such a delay line requires p operational amplifiers (op amps). Time multiplexing of one op amp and p capacitors to implement a p stage delay is possible [14]. If the signal to be delayed is available in digital form, a RAM of p words will function as a p stage delay line.

Fast, accurate multiplication of signals is another requirement of the autocorrelator. A digital multiplier provides exact multiplication of two digital numbers. Multiplication speed can be directly traded for IC area - a parallel multiplier is faster and larger than a serial multiplier. A multiplying digital-to-analog converter (MDAC) provides accurate multiplication of a digital signal with an analog signal; the product is in the form of an analog output voltage. The time per multiply for an MDAC is dependent on the speed of the op amp used and the type of MDAC (e.g. serial or parallel). Analog continuous time multipliers which multiply two analog signals and produce an analog product as output are also available. The digital multiplier is compatible with a digital delay line, the MDAC with either a digital or analog delay line, and the analog multiplier with an analog delay line.

The filters required for the autocorrelator are third-order low pass filters with a cutoff frequency of 25Hz. Digital filtering is required if a digital multiplier is chosen to multiply the speech signals. If either the MDAC or analog multiplier is chosen, the signal to be filtered will be analog. Either continuous time or discrete time analog filters can be used, but a 25Hz cutoff frequency would require a much larger chip area in a continuous time (resistor-capacitor) implementation. SC filters are discrete time filters and hence are described by z-domain transfer functions, so they can accurately realize the transfer functions $H'_k(z)$, $0<k<9$, required for this system (see equation (2.11)). With a sampling rate of 8kHz, the 25Hz low pass filters can be integrated as SC filters in a reasonable amount of silicon area. (For further discussion of analog circuits for signal processing, see appendix B.)

3.2.2. THE SELECTED CIRCUITS

The autocorrelator must provide adequate accuracy for speech analysis. Also, the goal of integrating the autocorrelator onto an IC must be considered. The above discussion breaks down to three basic choices - fully analog, fully digital, or some combination thereof. The fully analog autocorrelator - analog delay line, continuous time analog multiplier, and SC filters - could be constructed with only four op amps using multiplexed op amps for the filters and delay line. An ADC would be required to digitize the analog autocorrelation values for the microprocessor which implements the Durbin algorithm. A fully digital approach would require an ADC to digitize the incoming speech for processing. Previous work on autocorrelation LPC with finite word length arithmetic recommends 16 bit by 16 bit multipliers and 32 bit intermediate results to minimize the effects of quantization and truncation [15],[16]. A fast parallel multiplier could be multiplexed to handle all computations. Such a digital autocorrelator would essentially be a custom designed digital signal processor with a hardware multiplier and very limited instruction set. A hybrid approach - consisting of an MDAC, digital delay line, and SC filters - could be constructed with only three op amps because the MDAC can be incorporated into one of the SC filter sections as a variable input capacitor (see chapter 6 for more on this). Again, an ADC is required to digitize both the incoming speech and the autocorrelation values.

The hybrid approach potentially offers the lowest power, smallest area IC of the three options, as well as high accuracy and simple control logic. This approach was chosen for integration of the autocorrelator. Circuits for the MDAC and filters are considered below. Accuracy and circuit limitations are discussed in the following chapter.

The speech is sampled at an 8kHz rate, so the time between samples is 125μseconds. This is a very long time when compared to typical op amp settling times which are on the order of microseconds. Therefore it is possible and advantageous to employ time multiplexing of the MDAC and filters for the computation of the autocorrelation values. Time multiplexing results in the most efficient use of integrated circuit area.

The autocorrelator block diagram is shown in figure 3.2. The preprocessed speech is sampled at f_s=8kHz and held for the analog-to-digital converter (ADC) and multiplying digital-to-analog converter (MDAC). The digitized sample $s_d(n)$ (i.e. $s(n)$ digitized) is stored in RAM along with past samples $s_d(n-k)$, $1 \le k \le 9$. The 125μsecond sampling period is broken into ten 12.5μsecond time slots. During the k^{th} time slot, the sample $s_d(n-k)$ is read out of RAM and presented as the digital input to the MDAC where it is

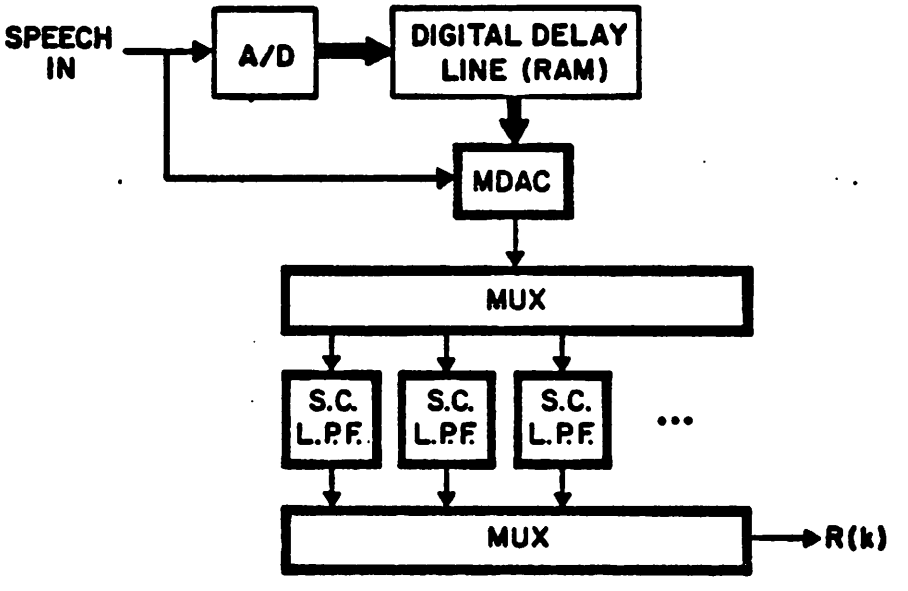


Figure 3.2 Autocorrelator block diagram

multiplied by the MDAC's analog input $s(n)$. The analog output of the MDAC, $s(n)s_d(n-k)$, is multiplexed to the k^{th} SC filter (with transfer function $H'_k(z)$, equation (2.11)) which implements the windowing and summation for the computation of $R(k)$. This procedure is carried out for each of the ten time slots, $0 \le k \le 9$; then the process is repeated for the new sample $s(n+1)$.

3.2.2.1. MDAC

A four-quadrant SC MDAC for the autocorrelator is shown schematically in figure 3.3. The MDAC is "parallel" rather than "serial" - multiplication occurs in one clock cycle for B bits rather than B cycles for B bits. The analog input V_{in} is connected to the analog sample-and-held voltage $s(n)$. The digital input word $s_d(n-k)$ in offset-binary format controls the switches S_1 through S_B. The MDAC is controlled by two non-overlapping clocks Φ_1 and Φ_2

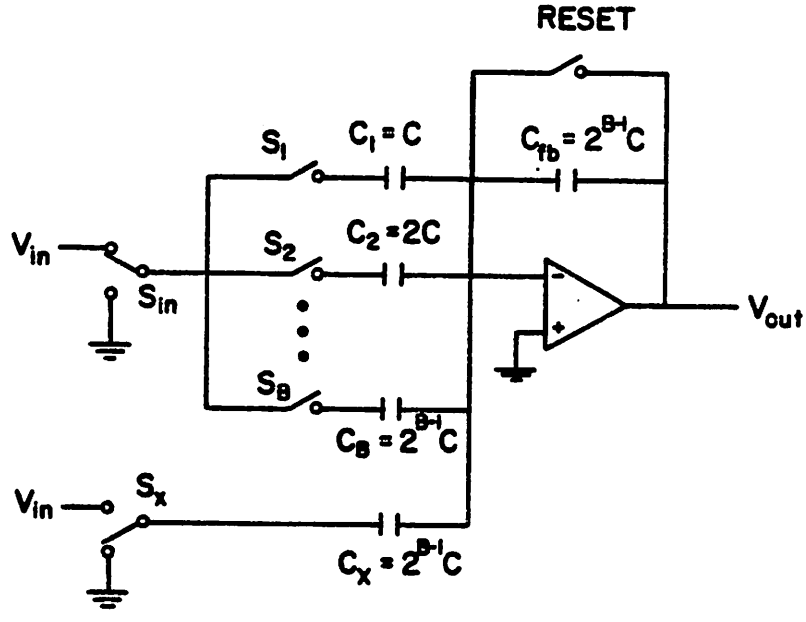


Figure 3.3. Four-quadrant switched-capacitor MDAC

(i.e. $\Phi_1 \cdot \Phi_2 = 0$, see figure 5.6.) During Φ_1, the reset switch is closed and switch S_x is connected to ground, so the feedback capacitor C_{fb} and the offset capacitor C_x discharge. The digital input word is presented, with switch S_j turning on and connecting capacitor C_j into the circuit if the j^{th} bit B_j is a logical '1'. The input switch S_{in} connects V_{in} to one side of the selected capacitors; the other side of the capacitors is held at zero volts because the op amp inverting input is a virtual ground. So the selected capacitors charge to V_{in}. On Φ_2, the reset switch across C_{fb} opens and switch S_{in} connects the selected capacitors to ground, forcing the stored charge onto C_{fb}. Simultaneously, C_x is connected to V_{in}, forcing a charge $Q = C_x V_{in}$ onto C_{fb}. The output voltage after the op amp settles is

$$V_{out} = \frac{\sum_{j=1}^{B} B_j C_j - C_x}{C_{fb}} V_{in}$$

which reduces to

$$V_{out} = \frac{\sum_{j=1}^{B} B_j 2^j - 2^B}{2^B} V_{in} = \left[\sum_{j=1}^{B} B_j 2^{j-B} - 1 \right] V_{in}$$

for a binary weighted capacitor array. As can be seen from the above equation, capacitors C_1 through C_B with C_{fb} act as a variable gain amplifier, the gain being controlled by the digital word $s_d(n-k)$. Capacitor C_x is switched out-of-phase with the other capacitors, thereby providing the half-scale offset required for a four-quadrant MDAC.

3.2.2.2. FILTERS

Switched-capacitor filters are analog, sampled-data filters. For the autocorrelator, sampled-data filters are required as discussed in chapter 2. The z-domain transfer function for the filters was given in equation (2.11). Therefore, the filters can be implemented in SC technology once an appropriate

configuration has been found.

The filters required each have three real poles at $z=\alpha^2$ and a real zero at $z=\frac{k-1}{k+1}\alpha^2$, $0\le k\le p$. A SC filter which implements a real pole and real zero is shown in figure 3.4. The circuit is controlled by two non-overlapping clocks Φ_1 and Φ_2 (i.e. $\Phi_1\cdot\Phi_2=0$.) On Φ_1, the switches are connected as shown in the figure. On Φ_2, the switches are flipped to the other positions. Writing charge transfer equations at two consecutive sample times, n and $n+1$, gives the time-domain relation

$$V_{out}(n+1) = \frac{C_{fb}}{C_{fb}+C_r}V_{out}(n)+\frac{C_i+C_z}{C_{fb}+C_r}V_{in}(n+1)-\frac{C_z}{C_{fb}+C_r}V_{in}(n) \qquad (3.1)$$

assuming that V_{out} is sampled on Φ_1 and that V_{in} changes only on the rising edge of Φ_1. Transformation of (3.1) into the z-domain gives

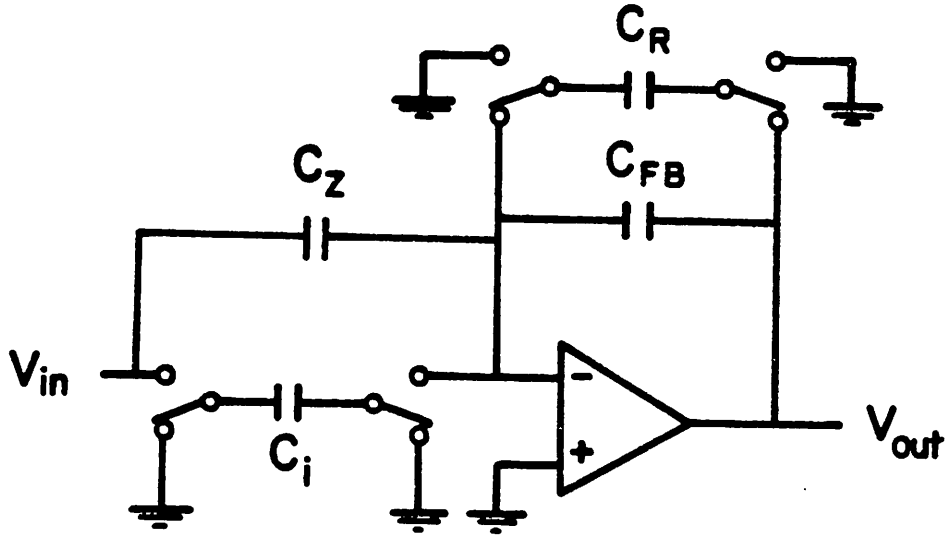


Figure 3.4 Switched-capacitor filter, real pole and real zero

$$\frac{V_{out}(z)}{V_{in}(z)} = -\frac{C_z + C_i}{C_{fb} + C_r} \frac{1 - \frac{C_z}{C_z + C_i} z^{-1}}{1 - \frac{C_{fb}}{C_{fb} + C_r} z^{-1}}. \tag{3.2}$$

Note that the pole location is $\frac{C_{fb}}{C_{fb} + C_r}$; the zero location is $\frac{C_z}{C_z + C_i}$; and the DC gain is $\frac{V_{out}(1)}{V_{in}(1)} = -\frac{C_i}{C_r}$. Since passive capacitor values are non-negative, the pole and zero locations for this configuration are in the interval [0,1]. Therefore, this filter can realize $H'_k(z)$ for any $k>0$ (see table 2.1). The case $k=0$ requires a negative real zero which can be achieved by reversing the switch phasing of C_i. If the phasing of C_i is reversed, the sign of the C_i term in equation (3.2) is inverted:

$$V_{out}(n+1) = \frac{C_{fb}}{C_{fb} + C_r} V_{out}(n) + \frac{-C_i + C_z}{C_{fb} + C_r} V_{in}(n+1) - \frac{C_z}{C_{fb} + C_r} V_{in}(n),$$

or in the z-domain

$$\frac{V_{out}(z)}{V_{in}(z)} = -\frac{C_z - C_i}{C_{fb} + C_r} \frac{1 - \frac{C_z}{C_z - C_i} z^{-1}}{1 - \frac{C_{fb}}{C_{fb} + C_r} z^{-1}}. \tag{3.3}$$

The pole location is the same as the previous circuit. The difference is the zero location which is now $\frac{C_z}{C_z - C_i}$. The zero can be any real value excluding the interval [0,1]. So the two circuits, identical except for the switch phasing of C_i, offer zero locations over the entire real axis.

A circuit with a transfer function identical to equation (3.2) but with different switch timing than figure 3.4 is shown in figure 3.5. This circuit switches the feedback capacitor C_{fb} out of the feedback loop rather than C_r. Likewise, C_z is switched out of the circuit rather than C_i. On Φ_1, the reset switch resets the op amp, C_i is connected to ground, and C_{fb} and C_z are floating. On Φ_2, the reset switch is open, C_i is connected to the input, and C_{fb} and

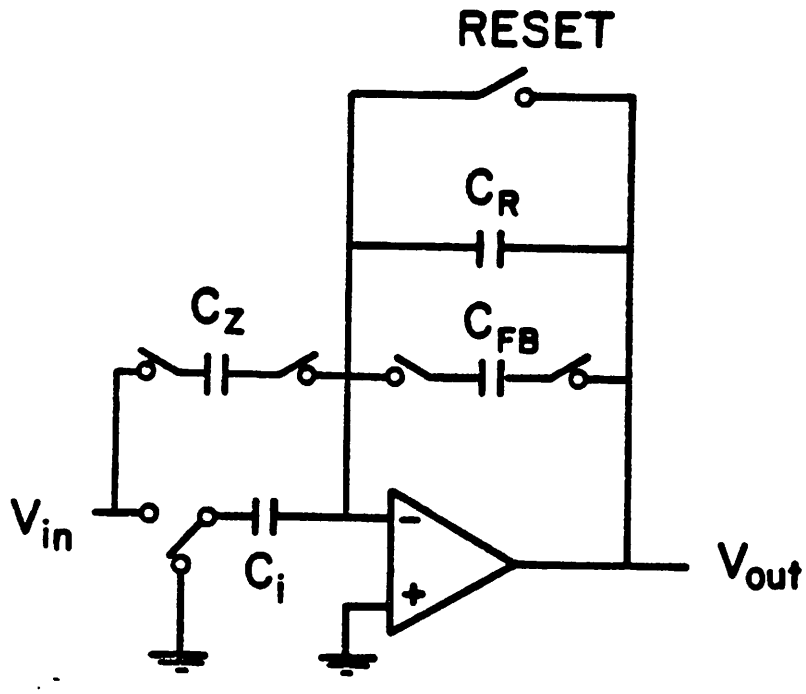


Figure 3.5. Switched-capacitor filter, one pole and one zero, different switching scheme

C_z are connected into the circuit. Again, notice that by simply inverting the phasing on the input switch for C_i, the input voltage $V_{in}(n+1)$ is introduced with a sign inversion. This changes the transfer function of the filter from (3.2) to (3.3).

Time multiplexing of the op amp of figure 3.5 for multiple filters is now possible since C_{fb} and C_z hold the state of the filter (figure 3.6). When C_{fb_k} and C_{z_k} are floating, a different pair of capacitors, $C_{fb_{k+1}}$ and $C_{z_{k+1}}$, may be switched into the circuit so that another filtering function can be performed. Each pair of capacitors, C_{fb_k} and C_{z_k} form one channel of the multiplexed filter section. There are ten channels per op amp to implement the ten filters required for the autocorrelator. Three multiplexed filter sections must be cascaded to implement the 3 pole, one zero transfer functions required

($H_k(z)$ of equation 2.11).

For the purpose of illustration, let $\Phi_A=\Phi_B=\Phi_2$ and RESET=Φ_1 in figure 3.6. The timing is as follows: during Φ_1, the reset switch is closed and discharges C_r, all C_{fb} and C_z capacitors are floating. During Φ_2, the reset switch is open, the input switch changes position, and one set of capacitors C_{fb_k} and C_{z_k} are connected into the circuit. C_i and C_z transfer charge onto $C_{fb}+C_r$. The filter transfer function is

$$\frac{V_{out}(z)}{V_{in}(z)} = -\frac{C_{z_k}+C_i}{C_{fb_k}+C_r}\,\frac{1-\dfrac{C_{z_k}}{C_{z_k}+C_i}z^{-1}}{1-\dfrac{C_{fb_k}}{C_{fb_k}+C_r}z^{-1}}.$$

This clocking repeats, but on Φ_2, $C_{z_{k+1}}$ and $C_{fb_{k+1}}$ are connected into the circuit. C_{z_k} and C_{fb_k} are now floating. If ten sets of capacitors are stacked on one

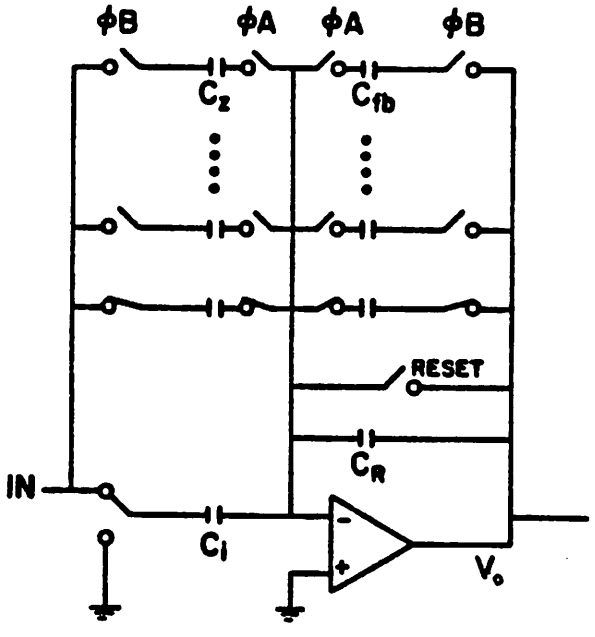


Figure 3.6. Multiplexed version of figure 3.5

Capacitor Ratios for SC Filters					
lag	transfer function	pole	zero	$\frac{C_{fb}}{C_r}$	$\frac{C_z}{C_i}$
k=0	equation (3.3)	0.9604	-0.9604	24.25	0.4899
k=1	equation (3.2)	0.9604	0.0000	24.25	0.0000
k=2	equation (3.2)	0.9604	0.3201	24.25	0.4708
k=3	equation (3.2)	0.9604	0.4802	24.25	0.9238
k=4	equation (3.2)	0.9604	0.5762	24.25	1.3596
k=5	equation (3.2)	0.9604	0.6403	24.25	1.7801
k=6	equation (3.2)	0.9604	0.6860	24.25	2.1847
k=7	equation (3.2)	0.9604	0.7203	24.25	2.5753
k=8	equation (3.2)	0.9604	0.7470	24.25	2.9526
k=9	equation (3.2)	0.9604	0.7683	24.25	3.3159

Table 3.1. Capacitor ratios for implementing $H'_k(z)$

op amp, one pole-zero pair for the ten filters $H'_k(z)$, $0 \le k \le 9$, have been realized. If the C_z capacitors are eliminated, one pole for ten filters is realized. The filters $H'_k(z)$ can be efficiently constructed using one multiplexed pole-zero section followed by two multiplexed pole sections. As noted earlier, changing the switch phasing of C_i during the $k=0$ time slot results in a sign reversal of the C_i term in the transfer function. This is necessary to implement the negative real zero for filter $H'_0(z)$.

The capacitor ratios for the filters $H'_k(z)$ are listed in table 3.1. Since all channels of a multiplexed filter section share the same C_r and C_i, all have the same DC gain which is $\frac{V_{out}}{V_{in}}(1) = -\frac{C_i}{C_r}$. As was noted in chapter 2, the DC gain for the filters $H'_k(z)$ is different for each k. Rather than have a different C_i

for each filter, the different DC gains are implemented by scaling in the microprocessor (see section 4.1.7).

3.3. DURBIN RECURSION MICROPROCESSOR

The analog autocorrelation values output by the autocorrelator must be transformed into LPC coefficients or reflection coefficients by the Durbin recursion algorithm (see section 2.4). For a 9 pole LPC model, the recursion involves 100 operations (multiplications or divisions) each frame. Accuracy considerations necessitate digital processing. Previous investigations have reported a need for 16 bit multiplies and 32 bit intermediate results in the recursion algorithm to minimize error due to finite word arithmetic [16],[15]. Given this, only 16 bit microprocessors with a multiply instruction were considered, and the Intel 8088 was chosen.

The microprocessor system is shown in figure 3.7. To digitize the analog autocorrelation values, an ADC is included. The memory requirements of the Durbin algorithm are minimal and are easily handled by a 256 by 8 bit RAM. The program size was estimated to be less than 1k bytes, so a 1k byte EPROM was chosen. Speed, word length considerations, and the program will be discussed in the following chapter.

3.4. AN AUTOMATIC GAIN CONTROL CIRCUIT

Ideally, the input speech would generate an R(0) (analog voltage) equal to the ADC's full scale input at every frame; this would minimize the error due to the quantization of the autocorrelation values. The magnitude of $R(0)$ has twice the dynmaic range of the speech because R(0) is the short-time energy of the speech waveform which is proportional to the speech amplitude squared. The dynamic range of the input speech is typically 30dB, resulting

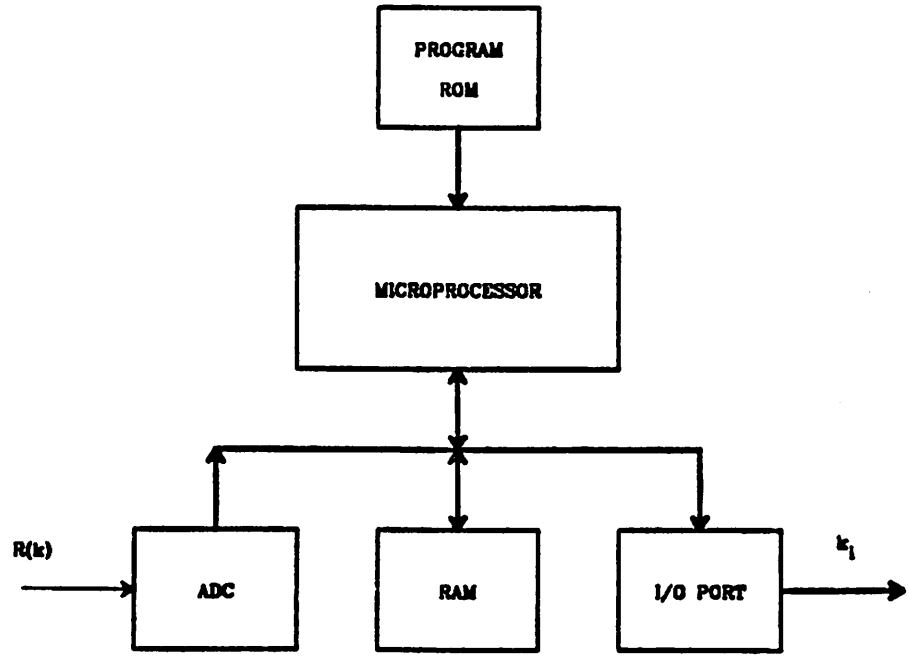


Figure 3.7. Block diagram of 8088 system

in a 60dB range for R(0). To minimize the dynamic range of $R(0)$, and thereby reduce errors due to quantization of the autocorrelation values, scaling of the input speech at each frame is necessary.

An automatic gain control circuit (AGC) for this purpose is shown in block diagram form, figure 3.8. The speech passes through two parallel paths. The upper path delays a frame of speech while the lower path computes an estimate of $\sqrt{R(0)}$ for the frame. Then, as the speech passes out of the delay line, it is scaled by $g \propto \frac{1}{\sqrt{R(0)}}$.

The lower path estimates $\sqrt{R(0)}$ rather than compute it exactly. An exact computation might seem in order, but the IIR window used in the autocorrelator overlaps multiple frames and therefore exact computation is neither possible nor worthwhile. Furthermore, the square root operation is not easily realized. To simplify matters, $\sqrt{R(0)}$ is estimated using the absolute

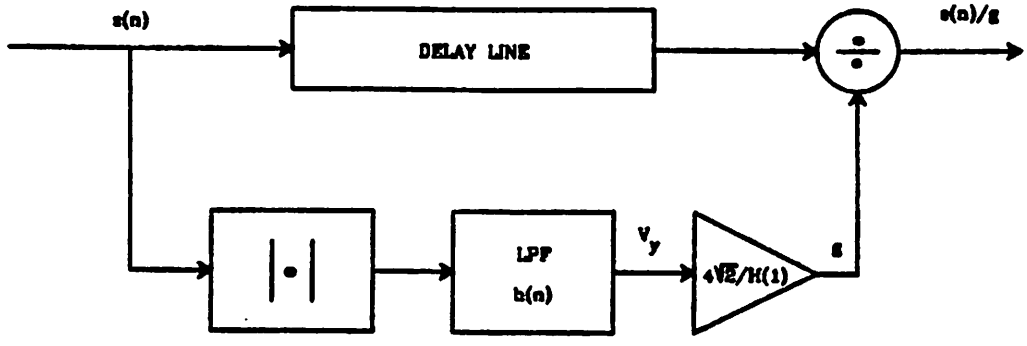


Figure 3.8 Block diagram of automatic gain control circuit (AGC)

value function and filtering,

$$V_y = \sum_{n=-\infty}^{\infty} |s(n)|w(n-j) = \sum_{n=-\infty}^{\infty} |s(n)|h(j-n),$$

where $w(n)$ is the same two pole window function as is used in the autocorrelation computation, equation (2.6), and $w(-n) = h(n)$ as in chapter 2. The use of the same window function should generate an estimate of $\sqrt{R(0)}$ which is fairly accurate since $R(0)$ and V_y are based on the same weighted speech segment.

V_y must be scaled to give an estimate of $\sqrt{R(0)}$. To find the scaling factor, we make use of statistical data on the amplitude distribution of speech. Speech has an amplitude distribution which is very closely approximated by the Laplacian distribution [17]

$$p(s) = \frac{1}{\sqrt{2}\sigma_s}\exp\left[-\frac{\sqrt{2}|s|}{\sigma_s}\right]$$

where σ_s is the root mean square value of speech. If we take the expected value of V_y

$$E(V_y) = \sum_{n=-\infty}^{\infty} E(|s(n)|)h(j-n) = E(|s(n)|)\sum_{n=-\infty}^{\infty} h(j-n) = E(|s(n)|)H(1)$$

because $\sum_{n=-\infty}^{\infty} h(j-n)$ is the DC gain of the filter $H(z)$, $H(1)$ (DC or 0Hz corresponds to $z=1$). $E(|s(n)|)$ is found by integrating

$$E(|s(n)|) = \int_{-\infty}^{\infty} |s|p(s)ds = \frac{\sigma_s}{\sqrt{2}}.$$

If the speech signal is assumed ergodic over a frame (12.5msec), then $\sqrt{R(0)}\approx\sigma_s$, so $E(V_y)\approx\frac{\sqrt{R(0)}H(1)}{\sqrt{2}}$.

Not only should the AGC scale the speech in an attempt to give $R(0)$ equal to the ADC's full scale input, it should set the gain so as to minimize clipping at the ADC. Clipping of the speech generates errors in large amplitude samples which heavily contribute to the autocorrelation values. Under the Laplacian distribution assumption, speech has the property that the amplitude is greater than $4\sigma_s$ only 0.35% of the time because

$$Prob[|s(n)|>4\sigma_s]=1-Prob[-4\sigma_s<s(n)<4\sigma_s]=1-\int_{-4\sigma_s}^{4\sigma_s} p(s)ds=0.0035.$$

So if the gain is set to $\frac{1}{g}=\frac{H(1)}{4\sqrt{2}V_y}\approx\frac{1}{4\sigma_s}$, the speech will be clipped about 0.35% of the time, so the error due to clipping will be negligible.

If the analyzed speech is to be reconstructed, the AGC gain should be stored and transmitted for use in the synthesizer. For such situations, an ADC is included to quantize the present scale factor g. Then, to handle the scaling of the speech, an MDAC is employed in the feedback loop of an op amp to

implement division. Only a small number of bits of resolution is required to give adequate performance in the AGC; five or six bits is sufficient [18]. A companding ADC with logarithmic transfer characteristic is preferred rather than a linear ADC. In any case, the AGC gain should be changed at the frame rate of 80Hz.

A feedback AGC system, without the delay line, was considered for this application. This would greatly simplify the AGC and make an IC realization much easier and smaller in area. A feedback AGC uses the last frame's energy, $R(0)$, to estimate the scaling factor for the present frame. If the signal does not vary rapidly from frame to frame, this would work very well. Unfortunately, speech changes very rapidly from frame to frame. To illustrate this point, let us consider a segment of background noise proceeding the word 'peat'. The background noise fills frame 1, and the /p/ sound begins at the start of frame 2. The energy in the frame of background noise will be very low, so the feedback AGC will set the gain to be very large for frame 2. The result is excessive clipping of the /p/ sound as it is scaled by the AGC. This causes a large error in the computation of the autocorrelation values for frame 2, resulting in an incorrect LPC model or an unstable model. When synthesized, the /p/ sound cannot be distinguished from a /b/ or /ee/, and the word 'peat' can not be distinguished from 'eat' or 'beat'. This is a common occurrence - plosives are transient in nature and are heavily clipped by a feedback AGC. Setting the gain for frame 2 based on the average energy in frame 2 eliminates this problem.

An AGC of the type described would be very useful as a speech preprocessor, whether the subsequent processing is analog or digital. The AGC reduces the bit requirement of the following ADC and minimizes quantization

error at the ADC which might generate unacceptable errors in computation.

CHAPTER 4

DESIGN OF THE SYSTEM

In the last chapter, circuits were chosen for implementing the autocorrelator, the AGC, and the Durbin recursion algorithm; but many design details such as the number of bits in the MDAC and the op amp design were not considered. The design details fall into two groups. Decisions such as the number of bits in the MDAC and the quantization of the autocorrelation values are speech related. On the other hand, design of the op amp and the SC circuits is independent of the speech processing application; the SC circuits should be designed to give accurate voltage transfer and filter characteristics. Computer simulation was used extensively during the design phase. The op amp and other analog circuits were simulated on the circuit simulation program SPICE [19]. Speech related simulations were done by running sentences of digitized speech through specially written programs.

4.1. SPEECH RELATED DESIGN

4.1.1. SIMULATIONS

A high quality microphone (Shure model SM-10) and 12 bit ADC were used to digitize sentences spoken by a variety of speakers. The sentences and speakers are listed in appendix C along with the source code for the speech simulation programs. The main program, Generateauto.c, simulates figure 3.1. This program takes speech as input and generates the autocorrelation values for each frame. It also features optional control of various aspects of the autocorrelator design such as AGC pre-processing, quantization of the

MDAC input, and quantization of the autocorrelation values.

After the speech is analyzed, it can be synthesized and played back by the program Speakpitch.c which is a direct-form LPC synthesizer (all computations in Speakpitch.c use floating-point arithmetic). For synthesis, the pitch periods are estimated by a Gold-Rabiner pitch detector [20]. Intelligibility is a subjective but important criterion which is applied to the synthesized speech to determine the quality of the analysis. Alternatively, the autocorrelation values can be compared to a reference set of autocorrelation values using the Itakura-Saito spectral distance measure, equation (2.15). The program spec_dev.c computes the spectral distance between two sets of autocorrelation values. This spectral distance measure, or spectral deviation from the reference, is extensively employed below to quantify the effect of quantization and other modifications to the LPC analysis discussed in chapter 2. Spectral deviation and the number of unstable frames are plotted versus design variables. This is an objective method of measuring errors. A spectral deviation of 3dB is considered to be the acceptable limit. Unfortunately, spectral deviation is not directly related to perceived errors. Large spectral deviations at high frequencies are not as objectionable as the same amount of spectral deviation at lower frequencies (especially near the first resonance). Nonetheless, 3dB of spectral deviation is generally considered tolerable (slight perceptual error); 1dB of spectral deviation is usually undetectable [21],[22].

Occasionally the LPC analysis results in an unstable filter modeling the vocal tract due to errors in computation (e.g. quantization and truncation). The effect of an unstable frame on the quality of the synthesized speech is not easily determined. If an unstable frame is isolated (i.e. surrounded by long

spans of stable frames), it can be replaced by a stable LPC model either by repeating the last stable frame or by interpolating the reflection coefficients of the neighboring frames. If a string of consecutive unstable frames occurs, it is difficult to find a stable model to replace them, and the result is an audible discontinuity in the synthesized speech.

4.1.2. DOWNSAMPLING

Each filter $H'_k(z)$ (equation (2.11)) is realized by cascading three first order low pass filter sections; each section has a cutoff frequency of 50Hz. As such, the third low pass filter section is presented with an input signal with very little signal power above 50Hz. This makes downsampling at the third filter section an attractive possibility. Downsampling reduces the computational workload in a digital approach and reduces capacitor ratios - and hence chip area - in a SC filter approach. This is a modification to the system which seems worthwhile.

A downsampling factor of ten was considered because it is convenient to implement. If the speech signal is spectrally flat, downsampling by a factor of ten would introduce aliases in the filter's passband (0Hz - 50Hz) that are 47dB below the baseband signal.

The simultation program includes an option for downsampling in the third filter section. This was used to test the effect of downsampling by a factor of ten. The results are tabulated in table 4.1, row 1. The units of spectral deviation are millibels (mB), 1mB = 0.01dB. Note that the errors contributed by downsampling are extremely small and are perceptually undetectable.

The downsampling by a factor of ten means the third filter section operates at a sampling rate of 800Hz. To achieve the desired cutoff frequency of 50Hz, the capacitor ratio $\frac{C_{f0}}{C_r}$ is only 2.04. This is much smaller than the

Spectral Deviation Results				
test condition	mean (mB)	standard deviation (mB)	number of unstable frames	total number of frames
downsampling by factor of 10 at third filter section	0	6	0	4239
AGC pre-processing	6	19	0	4239
AGC pre-processing and truncation of R(k) to nearest integer	11	32	55	4239
Truncation of R(k) to nearest integer without AGC pre-processing	143	184	2527	4239

Table 4.1. Spectral deviation results for various test conditions

ratio required for the first two filter sections which operate at an 8kHz sampling rate - they have $\frac{C_{fb}}{C_r}$=24.25.

4.1.3. EFFECT OF AGC ON STABILITY

An AGC is very important in this system - it relaxes the bit requirement of the MDAC and the ADCs. But an AGC can cause instability in the LPC model if the AGC changes its gain too often and/or too much, resulting in a time waveform which is difficult to model. The effect of the AGC was simulated, and the results are presented in table 4.1, row 2. Note that the AGC works very well because the spectral deviations are very small. The scaled speech (output of the AGC) was played back through loudspeakers and found to be extremely intelligible, only lacking the dynamic range of the original speech.

If the autocorrelation values are truncated to the nearest integer prior to the Durbin recursion algorithm, as is the case in this LPC system, the results are not as good (table 4.1, row 3). The larger spectral deviations and

the unstable frames are caused by quantization of the autocorrelation values. Such quantization is unavoidable because the autocorrelation values must be digitized for the microprocessor which handles the Durbin recursion algorithm. Still, the number of unstable frames is fairly small, only amounting to 1.2% of the frames analyzed. This should be contrasted with the results obtained without the AGC but including the integer truncation (table 4.1, row 4). Without the AGC, the large dynamic range of the speech results in a large dynamic range for $R(0)$. Small values of $R(0)$ result in heavy quantization of all the autocorrelation values and can cause large errors or instability in the LPC model. Figure 4.1 shows the distribution of $R(0)$ with and without AGC pre-processing. A comparison of the graphs clearly demonstrates that the AGC is beneficial.

4.1.4. MDAC BITS

The number of bits required in the MDAC should be minimized since for a SC MDAC the area of the capacitor array roughly doubles for each bit required. So, from an area standpoint, a minimum number of bits is desirable. Also, the RAM delay line size increases as the number of bits increases. Furthermore, the ADC is simplified and conversion time is reduced as the number of bits in the MDAC decreases. A good AGC reduces the number of bits required by keeping the signal amplitude large.

The simulation program Generateauto.c provides optional control of the number of bits in the MDAC. Due to quantization of the speech onto the computer, the MDAC is simulated by a 12 bit by B bit multiply where $1 \leq B \leq 12$. This is not the same as the MDAC which multiplies an analog voltage (ideally represented as a floating point quantity) and a finite length word. But quantization of both the multiplier and the multiplicand in the computer simulation

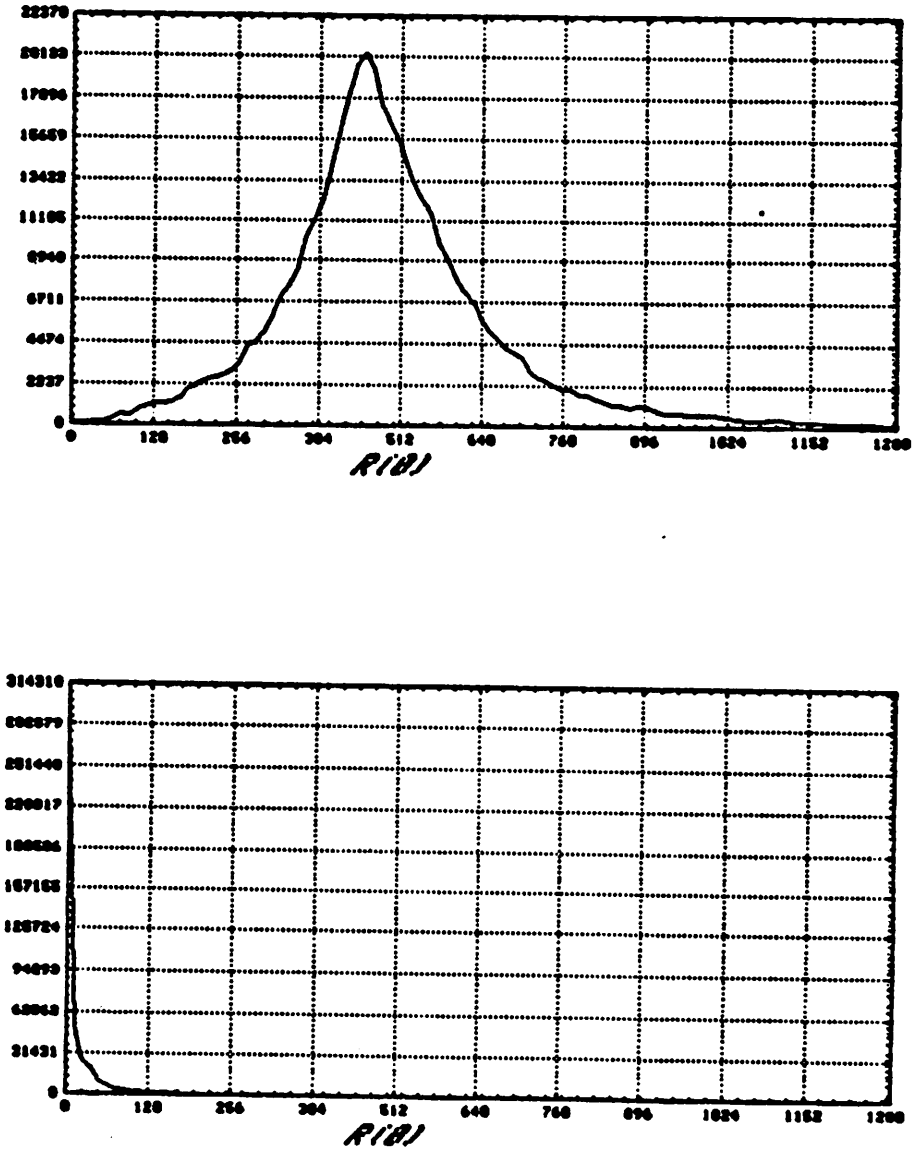


Figure 4.1. top: Distribution of R(0) values with AGC pre-processing
bottom: Distribution of R(0) values without AGC pre-processing

does provide an upper limit on the error due to quantization in the MDAC.

Let's consider the problem of quantization in the computation of $R(k)=\sum_{n=-\infty}^{\infty} s(n)s(n-k)w(n)w(n-k)$. Since the delayed speech is quantized prior to multiplication in the MDAC, $s(n-k)$ should be replaced by $s_d(n-k)=s(n-k)+q(n-k)$ where $q(n-k)$ is the quantization error due to the ADC. For an ADC which rounds the analog input to the nearest quantization level, $q(n)$ has a uniform probability distribution over the interval $-\frac{1}{2^B}\leq q\leq\frac{1}{2^B}$. If this quantized value is used in the autocorrelation computation, we get an approximation to $R(k)$:

$$\begin{aligned} R_a(k) &= \sum_{n=-\infty}^{\infty} s(n)s_d(n-k)w(n)w(n-k) \\ &= \sum_{n=-\infty}^{\infty} s(n)[s(n-k)+q(n-k)]w(n)w(n-k) \\ &= \sum_{n=-\infty}^{\infty} s(n)s(n-k)w(n)w(n-k)+\sum_{n=-\infty}^{\infty} s(n)q(n-k)w(n)w(n-k) \\ &= R(k)+\sum_{n=-\infty}^{\infty} s(n)q(n-k)w(n)w(n-k) \end{aligned}$$

where $R_a(k)$ is the approximation to $R(k)$. The quantization error q is a random variable with $E(q)=0$ and $Var(q)=\frac{2^{-2B}}{3}$. Furthermore, assuming the speech waveform traverses many quantization levels, $q(n)$ is independent of $q(m)$ for $m\neq n$ and q is uncorrelated with s [4]. If we assume a given speech waveform $\{s(n)\}$, then only q is a random quantity. The random variable $R_a(k)$ has expected value

$$\begin{aligned} E[R_a(k)] &= E\left[R(k)+\sum_{n=-\infty}^{\infty} s(n)q(n-k)w(n)w(n-k)\right] \\ &= R(k)+E\left[\sum_{n=-\infty}^{\infty} s(n)q(n-k)w(n)w(n-k)\right] \end{aligned}$$

$$= R(k) + \sum_{n=-\infty}^{\infty} s(n)E[q(n-k)]w(n)w(n-k)$$

$$= R(k),$$

so $R_a(k)$ is unbiased, and variance

$$Var[R_a(k)] = Var\left[R(k) + \sum_{n=-\infty}^{\infty} s(n)q(n-k)w(n)w(n-k)\right]$$

$$= Var\left[\sum_{n=-\infty}^{\infty} s(n)q(n-k)w(n)w(n-k)\right]$$

$$= \sum_{n=-\infty}^{\infty} Var[s(n)q(n-k)w(n)w(n-k)]$$

$$= \sum_{n=-\infty}^{\infty} s^2(n)Var[q(n-k)]w^2(n)w^2(n-k)$$

$$= Var[q(n-k)] \sum_{n=-\infty}^{\infty} s^2(n)w^2(n)w^2(n-k) \propto Var(q)R(0).$$

The proportionality stems from the observation that the final summation looks just like the expression for $R(0)$ if $w(n)$ is replaced by $w(n)w(n-k)$ (see equation (2.6)). So the final summation expresses the computation of $R(0)$ when a different window, $w(n)w(n-k)$, is used. The standard deviation of $R_a(k)$ is

$$\sigma_{R_a} = \sqrt{Var(R_a)} \propto \sqrt{Var(q)R(0)},$$

and the ratio of the desired output, $R(0)$, to the standard deviation of the actual output, σ_{R_a}, is

$$\frac{R(0)}{\sigma_{R_a}} \propto \frac{R(0)}{\sqrt{Var(q)R(0)}} = \frac{\sqrt{R(0)}}{\sqrt{Var(q)}}.$$

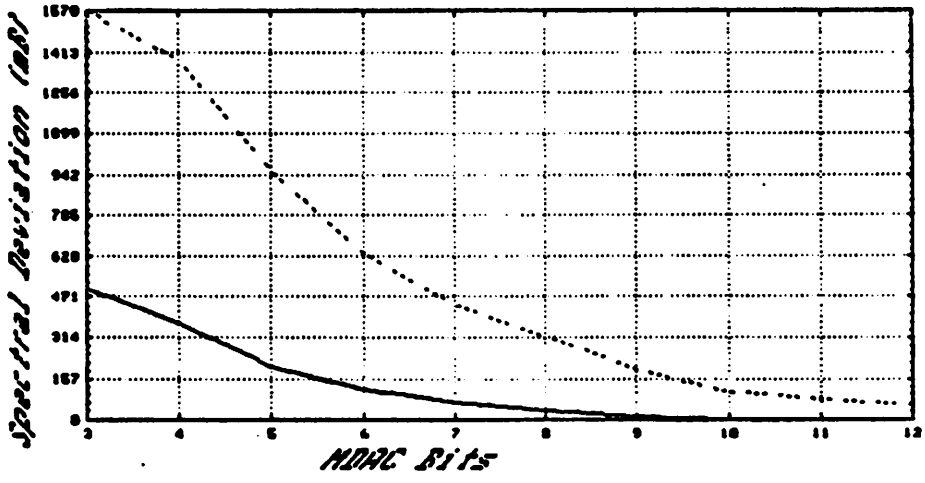
This ratio, which is similar to a signal-to-noise ratio, increases as $R(0)$ increases, so a large $R(0)$ is desirable. Scaling the input speech by using an AGC is one possible way to achieve a consistently large $R(0)$. The ratio also increases if $Var(q) = \frac{2^{-2B}}{3}$ decreases, this occurs if the number of bits B in the ADC is increased.

Spectral Deviation Results				
MDAC bits	mean (mB)	standard deviation (mB)	number of unstable frames	total number of frames
4	368	338	1302	4239
5	205	251	1105	4239
6	116	173	820	4239
7	65	126	383	4239
8	34	94	170	4239
9	14	59	41	4239
10	5	34	5	4239
11	2	24	0	4239
12	1	17	0	4239

Table 4.2 Spectral deviation results versus MDAC bits

Simulations with the number of MDAC bits as the independent variable were performed on all the sentences in the data base to determine the minimum number of bits acceptable. The results are tabulated in table 4.2 and graphed in figure 4.2. In the upper graph, the mean spectral deviation is plotted versus the number of MDAC bits with a solid black line; the mean plus three times the standard deviation of the spectral deviation data ($mean+3\sigma$) is plotted on the same graph with a dashed line. By Tchebychev's inequality [23] at least 89% of all the data lies below the dashed line. As can be seen from the graphs of figure 4.2, 8 bits (7 bits + sign) is acceptable.

In the autocorrelator, the MDAC is followed by a low pass filter $H'_k(z)$ with very narrow bandwidth. Therefore, ADC/MDAC non-linearities can be tolerated in moderation because they generate harmonic distortion at higher frequencies which will be attenuated by the filters.



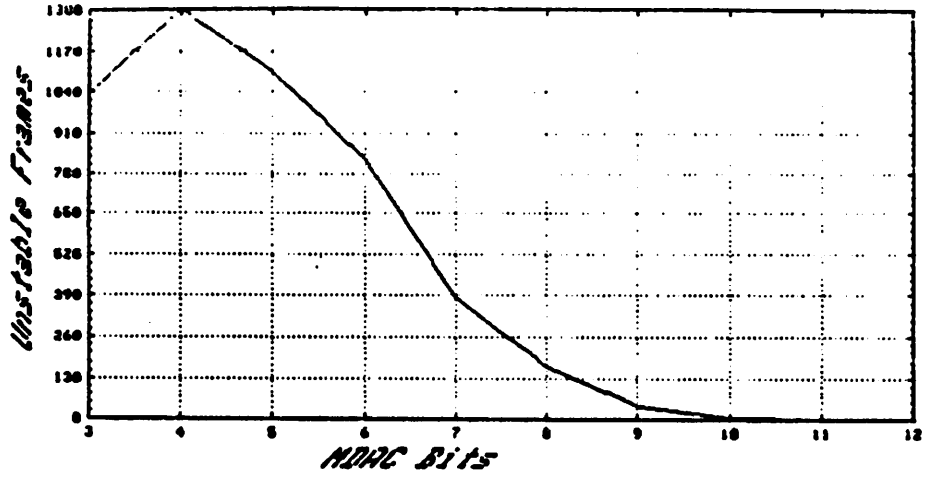


Figure 4.2 top: Spectral deviation versus MDAC bits
bottom: Unstable frames versus MDAC bits

Spectral Deviation Results				
R(k) quantization (bits)	mean (mB)	standard deviation (mB)	number of unstable frames	total number of frames
6	57	95	479	4239
7	35	79	268	4239
8	18	57	147	4239
9	10	38	71	4239
10	5	27	16	4239
11	2	11	9	4239
12	1	6	6	4239
13	0	5	1	4239
14	0	1	0	4239
15	0	0	0	4239
16	0	0	0	4239

Table 4.3 Spectral deviation results versus R(k) quantization

4.1.5. QUANTIZATION OF THE AUTOCORRELATION VALUES

Since the analog autocorrelation values must be quantized prior to the Durbin recursion algorithm, the effect of such quantization must be investigated. Quantization can lead to large spectral errors or even instability.

The simulation program Generateauto.c provides optional control of the quantization of the autocorrelation values. At each frame, the autocorrelation values are computed exactly using floating point arithmetic. Then, for quantization to B bits (B-1 bits + sign), all $R(k)$ are scaled by the factor $\frac{2^{B-1}}{R(0)}$ and truncated to the nearest integer. This results in $R(0)=2^{B-1}$. Table 4.3 and figure 4.3 show the results of such quantization. Quantization to the 10 bit level is acceptable, the number of unstable frames rapidly increases below 10

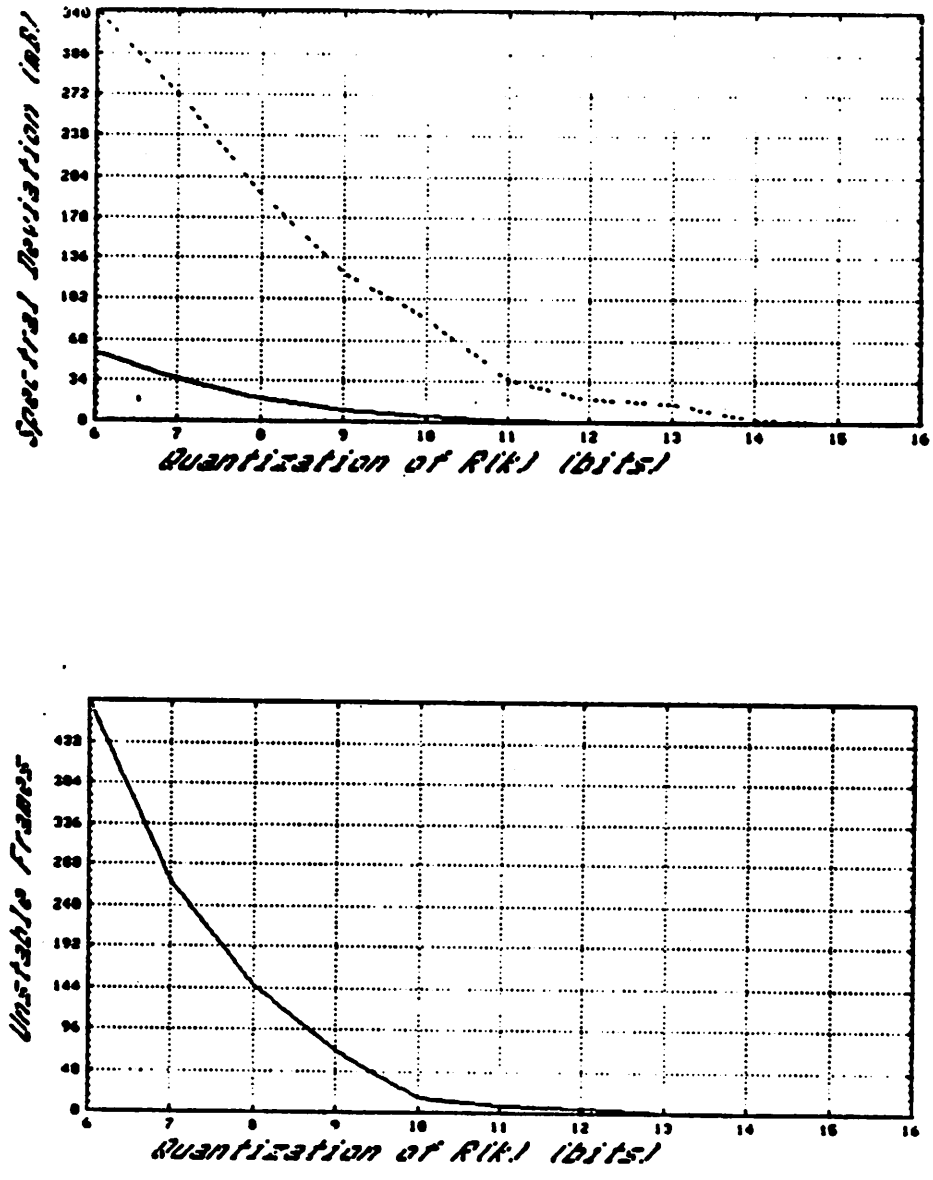


Figure 4.3 top: Spectral deviation versus R(k) quantization
bottom: Unstable frames versus R(k) quantization

bits.

4.1.6. DC GAIN OF FILTERS

Due to the filtering, the maximum signal amplitude at the output of each filter section is smaller than its maximum input signal amplitude. To maintain maximum output voltage and therefore the maximum $R(0)$ possible, each filter section has a DC gain larger than one. The gain was determined by running speech through the system. The main concern was to avoid clipping.

The filters are a cascade of first order sections. The one pole, one zero filter section was positioned first, followed by a one pole section which is in turn followed by the final one pole section which operates at the lower sampling rate. The selected gain values are 2.5, 1.38, and 1.3 respectively.

4.1.7. THE 8088 MICROPROCESSOR AND DURBIN'S RECURSION

The Durbin recursion algorithm transforms the quantized autocorrelation values into LPC coefficients a_i and reflection coefficients k_i. In a microprocessor implementation, the algorithm must be implemented with fixed point arithmetic. The two goals of the microprocessor system are real time computation while generating negligible modeling errors due to the integer arithmetic. The two requirements are conflicting in nature; increasing computational accuracy requires longer word lengths which means longer multiplication and division times. The design strategy was to first find word lengths at every step in the algorithm which produce negligible errors. Then this program was run on the microprocessor system to see if it was fast enough for real time analysis. Due to the availability of 8 bit and 16 bit manipulations with the 8088 instruction set, word lengths of 8, 16, 24, and 32 bits were considered.

The program spec_dev.c was modified to check the effect of finite length arithmetic in the Durbin algorithm. The flexibility of the C programming language, which allows manipulation of bits by shifting and masking, was very helpful.

Durbin's recursive solution of equation (2.12) is listed below, and the number of bits assigned to each variable is listed in Table 4.4. The real number 1.0 is scaled to $ONE=2^{14}$ in the 8088 program. After reading a set of autocorrelation values $R(k)$, $0 \le k \le 9$, each $R(k)$ is shifted left until there is a '1' in bit 14 of $R(0)$, so that $2^{14} \le R(0) < 2^{15}$. Each $R(k)$ is treated as a 16 bit integer. Since $|k_i| < 1$ for a stable model, $|k_i| < ONE$ in the 8088 program, so k_i is stored as a 16 bit number. The a_i's do not have a theoretical bound, as do the k_i, but they do have a practical bound for speech modeling; a practical bound of $|a_i| < 8$ was determined experimentally. In the 8088 program, $|a_i| < 8 \cdot ONE$ and therefore the largest a_i will require 18 bits; so each a_i is stored as a 24 bit word for convenience of manipulation. Also, it was found that the magnitude of the product $a_i R(m-i)$ never exceeds 2^{31} (32 bits) despite the fact that a_i is an 18 bit integer and $R(m-i)$ is a 16 bit integer. Therefore the product can be stored as a 32 bit integer. Likewise, the product $k_m a_{m-i}^{m-1}$ never exceeds 32 bits. The 8088 program includes a subroutine for

Word sizes for the Durbin algorithm	
variable name	bit length
$R(k)$	16
m	8
k_m	16
a_i^m	24
$a_i^{m-1} R(m-i)$	32
$k_m a_{m-i}^{m-1}$	32
E_m	16

Table 4.4. Word size at different points in the Durbin algorithm.

multiplying an 18 bit word (a_i) by a 16 bit word. This multiplication time is reduced, on the average, by checking the top two bits of the 18 bit integer because if the top two bits are just a sign extension, a 16 bit by 16 bit multiplication will suffice.

The microprocessor also must modify the output of the SC filters before beginning the Durbin recursion. Each SC filter has an undesirable DC offset voltage which adds to the autocorrelation value. Also, the SC filters all have the same DC gain (see section 3.2.2.2). But the gains should be slightly different as can be seen in table 2.1. The 8088 program goes through an initialization phase during which the autocorrelator's input is zero. This allows the 8088 to sample the DC offsets of the filters and store them. Then the microprocessor begins execution of the main program. Each frame, a new set of autocorrelation values are read. For each autocorrelation value, the corresponding offset voltage is subtracted and the difference is scaled by the relative DC gain of table 2.1. The result is the autocorrelation value which is used in the Durbin recursion algorithm:

start: $E_0=R(0)$

$m=0$

loop: $m=m+1$

$$k_m=\frac{R(m)+\sum_{i=1}^{m-1}a_i^{m-1}R(m-i)}{E_{m-1}}$$

$a_m^m=k_m$

$a_i^m=a_i^{m-1}-k_m a_{m-i}^{m-1} \quad 1\le i\le m-1$

$E_m=(1-k_m^2)E_{m-1}$

if($m<p$) *go to loop*;

else stop;

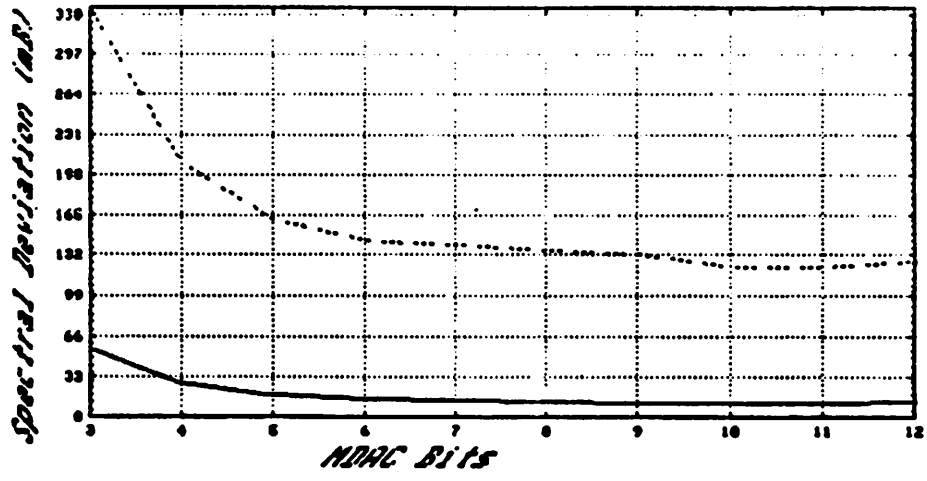
Spectral Deviation Results				
MDAC bits	mean (mB)	standard deviation (mB)	number of unstable frames	total number of frames
4	28	60	246	4239
5	18	48	117	4239
6	15	43	79	4239
7	14	42	49	4239
8	13	41	52	4239
9	12	40	51	4239
10	12	37	62	4239
11	12	37	52	4239
12	13	38	51	4239

Table 4.5. Spectral deviation results versus MDAC bits with all modifications

The final version of the 8088 program required 120 bytes (960 bits) of RAM to store all the variables. A 256 by 8 RAM was chosen; this allows 136 bytes for the program stack which was more than necessary. The stack is only used to save a few registers temporarily during a subroutine call.

4.1.8. TOTAL ERROR FROM ALL MODIFICATIONS

As might be expected, inclusion of all the modifications discussed in sections 4.1.2 through 4.1.7 results in an error which is larger than the error due to any one modification alone but less than the sum of the individual errors. The results of downsampling, MDAC quantization, $R(k)$ truncation to the nearest integer, and AGC pre-processing all simultaneously in effect are tabulated in table 4.5 and graphed versus MDAC bits in figure 4.4. Using an 8 bit MDAC results in acceptable performance. Auditioning synthesized speech



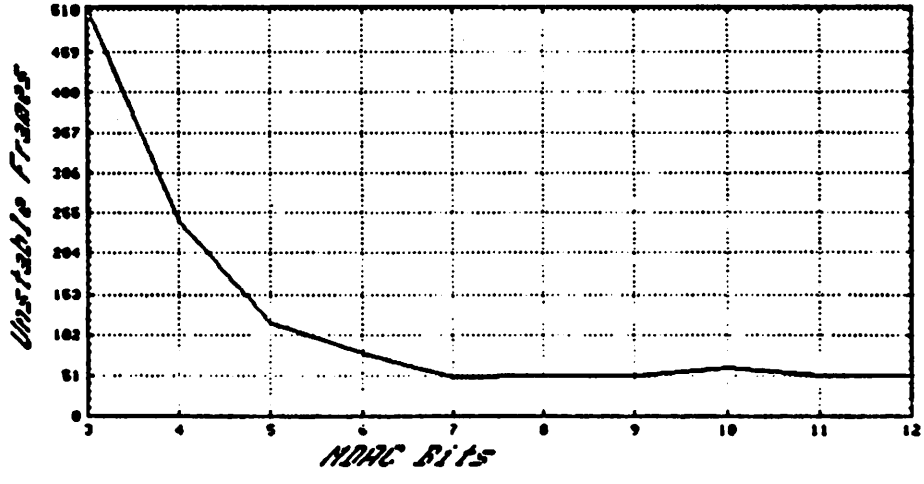


Figure 4.4 top: Spectral deviation versus MDAC bits, final system
bottom: Unstable frames versus MDAC bits, final system

confirmed this decision. If an 8 bit MDAC was used in the analysis, the synthesized speech was easily understood although occasional minor flaws could be detected.

4.2. ANALOG IC DESIGN

To demonstrate the feasibility of the analog/digital hybrid approach, the SC circuits for the autocorrelator were integrated on a single CMOS chip. The digital circuits for the autocorrelator are simple ciruits (i.e. small RAM, control logic) and were not included on the chip. MOS was chosen because it accommodates SC circuitry and is compatible with digital circuitry; CMOS, rather than NMOS or PMOS, was chosen because it offers more flexibility as well as more gain per amplifier stage. The IC design was a two part job, one being MDAC and SC filter design to minimize parasitic related errors, the other the design of the op amp. One parasitic, MOSFET channel charge redistribution during turn-off, is not well understood and hence it is not modeled properly in circuit simulators such as SPICE [24]. So reliable computer simulation of these charge effects was not possible. Therefore, a breadboard version of the autocorrelator was built for experimentation. Op amp design was done exclusively on SPICE.

4.2.1. THE PROCESS

For integration of the IC, a reliable metal-gate CMOS process was chosen. A P-type substrate, 1000 angstrom gate oxide thickness, and 30V oxide breakdown voltage are characteristics worth noting. The minimum drawn line width is 10μm which results in 8μm channel lengths after out-diffusion of the source and drain. The gate-to-drain and gate-to-source overlaps are 1 to 2 microns for the NMOSFET, but only 0.1 to 0.2 microns for the PMOSFET due to a self-

aligning P+ implant. Capacitors are MOS capacitors (metal-oxide-N+). The threshold voltages are typically 1V for the NMOSFET and -2V for the PMOSFET. Since the substrate is P-type, the NMOS transistors are used as switches; PMOS transistors must be placed in an N-well so they are inefficient from an area standpoint unless a number of switches can be grouped into one well. A more detailed description of the process can be found in appendix D as well as in [14].

4.2.2. A SIMPLE MODEL FOR THE MOSFET

For analog circuit design, simple equations which model the MOSFET are useful. Equations for an N channel MOS transistor are given below, and all parameters are defined. The P channel MOS transistor follows the same equations if the direction of positive drain current is changed. Typical values for the parameters are listed in table 4.6.

If the transistor is in the linear region of operation (i.e. if a continuous channel extends from drain to source), the MOSFET acts as a resistance of value

Typical Values for MOSFET Model Parameters			
Parameter	SPICE Symbol	NMOS	PMOS
V_{th}	VTO	1.0V	-2.0V
k	KP	$24\frac{\mu A}{V^2}$	$7\frac{\mu A}{V^2}$
λ	LAMBDA	$0.01V^{-1}$	$0.005V^{-1}$
N_{SUB}	NSUB	1.6×10^{15}	1.0×10^{16}
t_{ox}	TOX	0.1μ	0.1μ

Table 4.6. Model parameters for SPICE simulations

$$R_{on} = \frac{1}{k\frac{W}{L}(V_{gs}-V_{th}-\frac{V_{ds}}{2})} \approx \frac{1}{k\frac{W}{L}(V_{gs}-V_{th})} \quad (4.1)$$

The approximation is valid if $V_{gs}-V_{th} \gg \frac{V_{ds}}{2}$. The parameters are

$k = \mu C'_{ox}$,

μ = the mobility of the channel charge,

C'_{ox} = the gate to channel capacitance per unit area,

W = the width of the active area,

L = the length of the active area,

V_{gs} = the gate to source voltage,

V_{ds} = the drain to source voltage,

V_{th} = the threshold voltage of the transistor.

The transistor is in the linear region if both $V_{gs}>V_{th}$ and $V_{gd}>V_{th}$. If $V_{gs}>V_{th}$ but $V_{gd}<V_{th}$, then the transistor is in the saturation region (i.e. the channel is pinched off at the drain), and the transistor follows the relation

$$I_{ds} = \frac{k}{2}\frac{W}{L}(V_{gs}-V_{th})^2(1+\lambda V_{ds})$$

where

I_{ds} = the drain to source current,

λ^{-1} = the Early voltage of the transistor.

The threshold voltage V_{th} is dependent on the source to substrate voltage

$$V_{th} = V_{th0}+\gamma\left[\sqrt{|V_{bs}|+2|\varphi_f|}-\sqrt{2|\varphi_f|}\right]$$

where

V_{th0} = the threshold voltage measured with $V_{bs}=0$,

γ = the body effect parameter,

V_{bs} = the substrate to source voltage,

φ_f = the Fermi level of the substrate.

The capacitances associated with the MOSFET are shown in figure 4.5. The N+ source and drain diffusions in the P substrate form diode junctions which are always reverse-biased and have an associated non-linear junction capacitance. The gate metal overlaps the source and drain diffusions slightly, giving rise to overlap capacitances C_{ov}. These overlap capacitances provide an undesirable coupling between the gate and source and between the gate and drain when the transistor is off. The total gate to source, gate to drain, and gate to substrate capacitances are complicated functions of the terminal voltages [25].

4.2.3. OPERATIONAL AMPLIFIER

Since all circuits employing the op amp are SC circuits, the op amp loading is capacitive. A simple transconductance amplifier which uses the capaci-

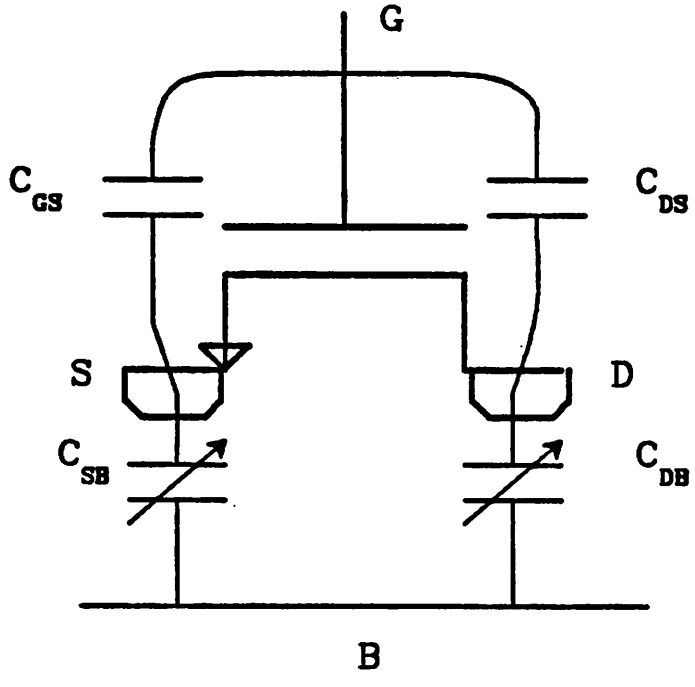


Figure 4.5. Capacitances associated with a MOSFET

tive load for compensation was designed. Important design goals are settling to 0.1% in 3μseconds, open loop gain greater than 1000, large output swing and low noise. Low power dissipation and small area are desirable but not crucial due to the multiplexing of each op amp.

The op amp schematic is shown in figure 4.6. Current source M14 biases the input differential pair M1-M2. Wilson current mirrors M3-M4-M5 and M6-M7-M8 provide current gain as well as high output impedance. The drain current of M4 is mirrored to the output by cascode current mirror M9-M10-M11-M12. The op amp has a differential mode transconductance of G_m

$$G_m = \frac{1}{2} g_{m_{in}} \left[\frac{\frac{W_8}{L_8}}{\frac{W_6}{L_6}} \right] + \frac{1}{2} g_{m_{in}} \left[\frac{\frac{W_5}{L_5}}{\frac{W_3}{L_3}} \right] \left[\frac{\frac{W_{11}}{L_{11}}}{\frac{W_{10}}{L_{10}}} \right];$$

$g_{m_{in}}=g_{m_1}=g_{m_2}$ is the transconductance of the input transistors M1 and M2. If the $\frac{W}{L}$ ratios are as shown in figure 4.6, G_m reduces to

$$G_m = g_{m_{in}} \left[\frac{\frac{W_8}{L_8}}{\frac{W_6}{L_6}} \right] = g_{m_{in}} \left[\frac{\frac{W_5}{L_5}}{\frac{W_3}{L_3}} \right] = g_{m_{in}} A_I \propto \sqrt{I_{bias}}.$$

A_I is the current gain of the Wilson mirrors M3-M4-M5 and M6-M7-M8, A_I=2.5 in figure 4.6. The differential mode voltage gain is just $A_v = G_m R_{out}$ where R_{out} is the output impedance of the Wilson mirror M6-M7-M8 in parallel with the output impedance of the cascode mirror M9-M10-M11-M12.

$$R_{out}(Wilson) \approx r_{o7}(g_{m6} r_{o6}) = \frac{\sqrt{16 \mu C'_{ox} \frac{W_6}{L_6}}}{\lambda_7 \lambda_6 A_I I_{bias}^{\frac{3}{2}}}$$

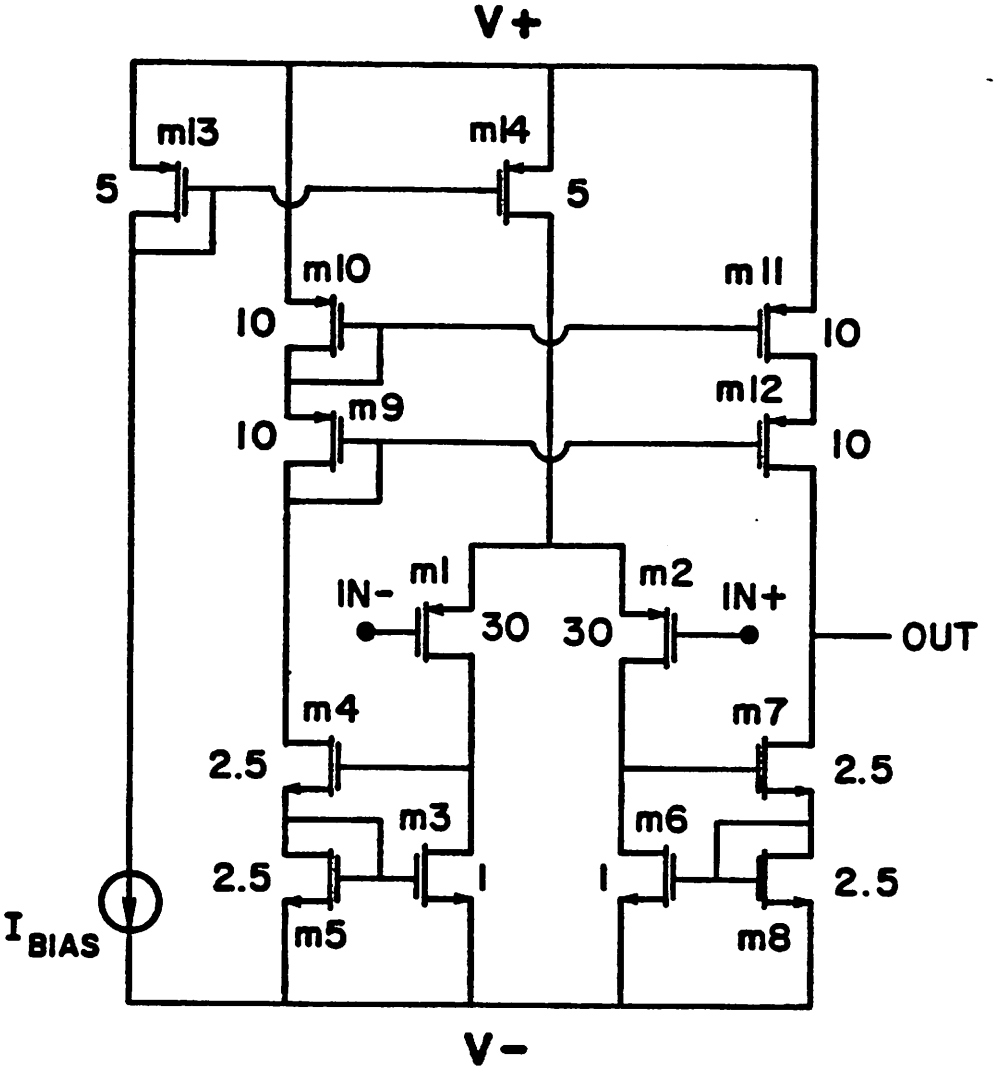


Figure 4.6. CMOS op amp schematic ($\frac{W}{L}$-ratios next to transistors)

$$R_{out}(Cascode) \approx r_{o12}(g_{m12}r_{o11}) = \frac{\sqrt{16\mu C'_{ox}\frac{W_{12}}{L_{12}}}}{\lambda_{12}\lambda_{11}(A_I I_{bias})^{\frac{3}{2}}}$$

so

$$R_{out} = R_{out}(Wilson)\,||\,R_{out}(Cascode) \propto I_{bias}^{-\frac{3}{2}}$$

giving

$$A_v = G_m R_{out} \propto \sqrt{I_{bias}}\, I_{bias}^{-\frac{3}{2}} = \frac{1}{I_{bias}}.$$

So the low frequency voltage gain is inversely proportional to bias current. But reducing the current to achieve higher gain reduces bandwidth and increases the noise [26].

Since the CMOS process uses a P-type substrate, the PMOS transistors sit in an N-well and therefore have a higher output resistance than the NMOS transistors. So the PMOS transistors were chosen for the simple current source for biasing the input stage. The input transistors M1 and M2 are large area devices to minimize the op amp's noise [26].

The op amp is employed in a shunt-shunt feedback configuration with its non-inverting input grounded. In such a configuration, the common mode input signal is very small so a large common mode rejection ratio is not required. Nonetheless, the symmetry of the op amp should provide a very large common mode rejection.

4.2.4. SPEED AND COMPENSATION

The op amp is basically a one stage design. As such, it should provide a wide bandwidth. All nodes internal to the op amp have an low frequency impedance of $\frac{2}{g_m}$; the highest impedance node is the output node. G_m has two non-dominant poles due to the total capacitance at the drains of the

input transistors M1-M2 interacting with the equivalent resistance looking into the Wilson mirrors

$$p_{nd} = -\frac{g_m}{2C_i} = -\frac{g_m}{2(C_{ov}+C_j)}.$$

(C_{ov} is the gate overlap capacitance, C_j is the junction capacitance associated with the source and drain diffusions). The dominant pole is at the output node due to the large output resistance R_{out} interacting with the feedback and load capacitors.

Compensation is only meaningful in the context of feedback circuits. In the SC autocorrelator, the op amp is subjected to numerous feedback configurations. In all cases, the feedback is shunt-shunt (i.e. voltage output, current input).

Stability and settling time can be determined from the phase margin of the loop gain [27]. The loop gain of the shunt-shunt feedback circuit figure 4.7 is

$$T(s) = \frac{(G_m - y_f)y_f}{(y_i+y_f)(y_o+y_f)} \tag{4.2}$$

where the reverse transmission through the op amp has been neglected - it is assumed to be much smaller than $|y_f|$ at all frequencies of interest (appendix E). All terms of (4.2) are functions of the complex frequency s. The loop gain $T(s)$ can be simulated directly on SPICE using the configuration of figure 4.8. The op amp with feedback element y_f across it has a voltage gain

$$\frac{V_o}{V_x} = -\frac{(G_m - y_f)}{(y_o+y_f)}.$$

The voltage divider has the transfer function

$$\frac{V_r}{V_o} = \frac{y_f}{(y_i+y_f)}.$$

The cascaded gain is the loop gain

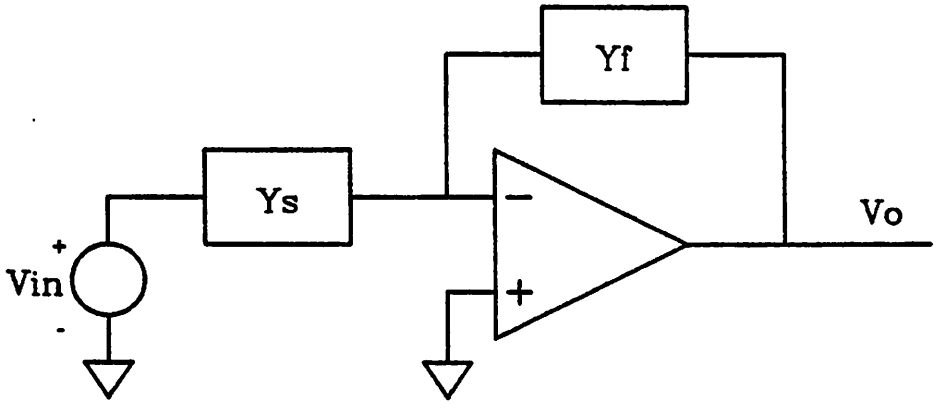


Figure 4.7. Shunt-shunt feedback circuit

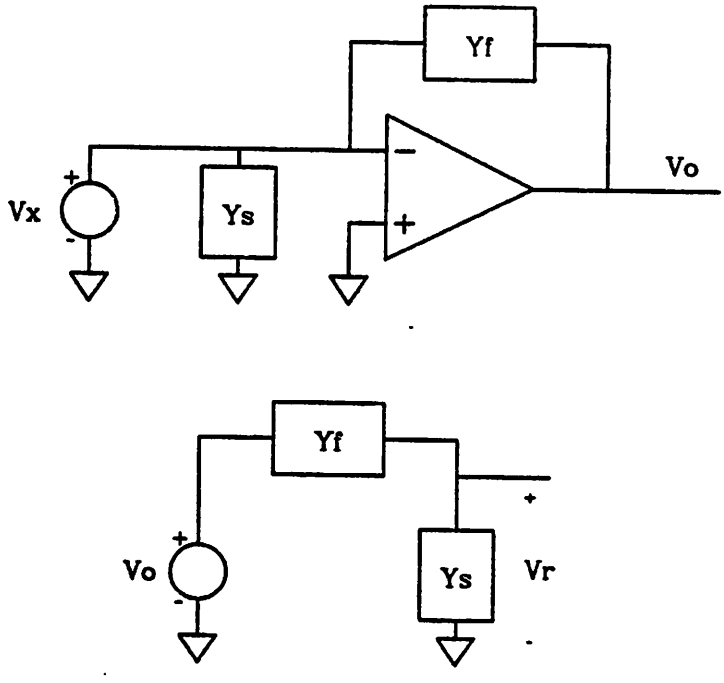


Figure 4.8. Circuit for simulating loop gain

$$T(s) = -\frac{V_r}{V_x} = \frac{V_o}{V_x}\frac{V_r}{V_o}\frac{(G_m - y_f)y_f}{(y_i + y_f)(y_o + y_f)}.$$

This provides a very simple means of simulating the loop gain directly. This is important in circuits which have frequency dependent feedback such as those in the autocorrelator. Exact simulation of loop gain is discussed further in appendix E.

Once an acceptable phase margin was found using figure 4.8, the step response of the feedback circuit was simulated. The worst case for stability occurs in the reset configuration. Settling times for MDAC and multiplexed filters are tabulated in table 4.7. The SPICE model parameters which were used are listed in table 4.6. These are conservative model parameters based on data from previous experiments [28].

Settling Time Simulation Results				
Circuit Simulated (figure)	Switch Phasing	C_f or C_1	Load Capacitance	Settling Time to 0.1%
MDAC (fig 3.3)	reset switch on	C_1=0.1pF	0pF	2.2μsec
MDAC (fig 3.3)	reset switch off	C_1=0.1pF	0pF	1.3μsec
Multiplexed filter (fig 3.6)	reset on	C_f=0.22pF	10pF	1.2μsec
Multiplexed filter (fig 3.6)	reset off	C_f=0.22pF	10pF	1.9μsec

Table 4.7. Settling time results from SPICE for different configurations

4.2.5. EFFECT OF PARASITICS ON MDAC AND SC FILTERS

To implement the SC circuits of chapter 3, an N channel MOSFET is used wherever a switch is needed. But the MOSFET is not an ideal switch. Aside from finite on-resistance and gate capacitance, the gate overlap capacitance and channel charge introduce error charge in the signal path of the SC circuits. In some configurations, this error charge is signal dependent which causes distortion. These non-idealities must be considered and their effects minimized during the design phase.

In SC circuits, the finite on-resistance of the MOSFETs slows the charging of the capacitors. For a capacitance C and MOSFET on-resistance R_{on}, the associated time constant is $\tau=R_{on}C$. To charge the capacitance to within 0.1% of its final value requires seven time constants (7τ). Charging time does not present a problem in this system because the largest time constants are a few hundred microseconds, so $7\tau<1\mu$sec which is much less than the smallest time allowed for charging (which is 4μsec). If the charging time were a problem, the time constant could be reduced by decreasing C or by decreasing R_{on}. The on-resistance can be decreased by increasing gate drive voltage $(V_{gs}-V_{th})$ or increasing the transistor's $\frac{W}{L}$(see equation (4.1)).

For the SC MDAC of figure 4.9, the input switches S_1 through S_B and S_{in} can be considered to be ideal switches. This can be understood by noting that the MDAC operates by transferring charge from the input capacitor array C_1 through C_B onto C_{fb} on Φ_2. The Gaussian surface, shown as the dotted line in figure 4.9, represents the charge transfer region. As long as the reset switch is open, the charge inside the surface must remain constant. Since the switches S_1 through S_B are outside this surface, they do not contribute to the charge transferred and therefore have no effect on the output voltage. The

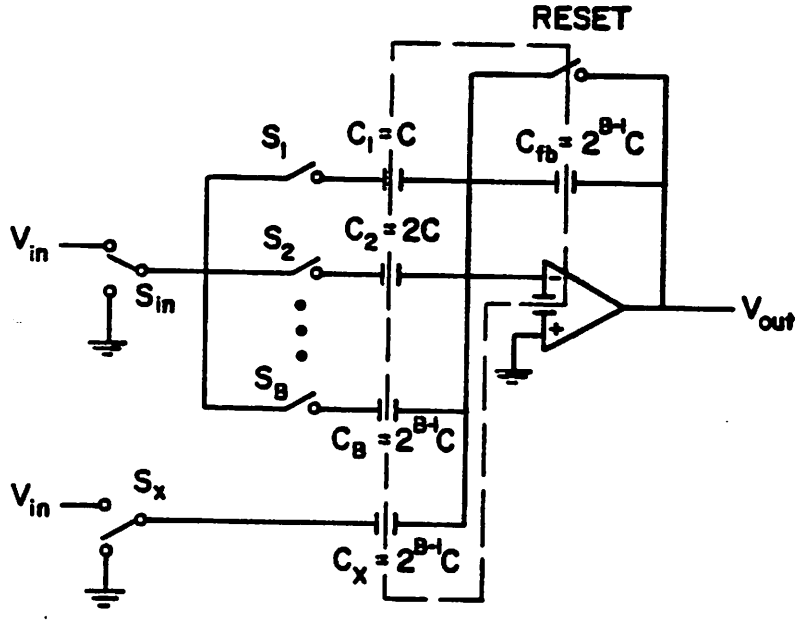


Figure 4.9. SC MDAC

only error charge is contributed by the reset switch which provides DC stability and therefore breaks the Gaussian surface. As the reset switch turns off, some fraction of its channel charge will be trapped on the inverting node. Also, the reset clock is coupled to the inverting node through the gate overlap capacitance C_{ov}. The output voltage due to these sources can be approximated as

$$V_{out} \approx \frac{C_{ov}}{C_{fb}}\Delta V_{ck} + \frac{rQ_{channel}}{C_{fb}}$$

where ΔV_{ck} is the amplitude swing of the clock signal, $Q_{channel} = C'_{ox} WL(V_{gs} - V_{th})$ is the charge accumulated in the channel when the transistor is in the linear region, and $0 \le r \le 1$ is the fraction of channel charge dumped onto the inverting node when the transistor turns off. Fortunately, prior to each multiplication, the MDAC is in the reset mode. So the feedthrough causes a constant DC offset voltage at the MDAC output. The

clocking scheme results in the op amp's offset voltage (V_{offset}) being sampled onto the input capacitor array each reset period. So the op amp's offset voltage is not multiplied by the variable gain of the MDAC, but it does appear directly at output. The total DC output voltage is

$$V_{out_{DC}} \approx V_{offset} + \frac{C_{ov}}{C_{fb}}\Delta V_{cl} + \frac{\tau Q_{channel}}{C_{fb}}. \qquad (4.3)$$

This offset passes through the SC filters and is subtracted digitally in the microprocessor as discussed in section 4.1.7.

Finite op amp gain results in a gain error in the MDAC. This gain error occurs for all products and therefore scales all autocorrelation values by the same factor. Since the LPC model is not changed by a uniform scaling of all autocorrelation values (see section 2.4), gain error is of no consequence in this application.

The diode junction composed of a capacitor's N+ diffusion and the P- substrate is always reverse biased, so it has an associated non-linear capacitance and reverse-bias leakage current. For the binary capacitor array, the N+ diffusions of the capacitors are connected to the input switches S_1 through S_B. This eliminates problems due to the non-linear capacitances because they are driven by the input source. For C_{fb}, the diffusion is connected to the output of the op amp. The non-linear junction capacitance associated with C_{fb} does present a non-linear capacitive load to the op amp which helps to compensate the op amp.

When all the switches S_1 through S_B are on, the leakage currents for the diode junctions of C_1 through C_B are supplied by the source driving V_{in}, and the leakage current for the diode on the output of the op amp comes from the output stage of the op amp; therefore the leakage currents have no effect. (Actually, the leakage current flowing through the MOSFET switch

drops a negligible voltage $I_{leak}R_{on}$.) The leakage current associated with a junction does present a source of error if any of the switches S_1 through S_B is open. If input switch S_j is open, a leakage current due to C_j's backplate diode flows through it and onto C_{fb}. This leakage dominates the small leakage current associated with the drain diffusion of the MOSFET switch S_j. The maximum error due to leakage currents occurs when all switches S_1 through S_B are off (i.e. all bits in the digital word are '0'). Then the error is roughly

$$\Delta V_{out} = \frac{\sum I_{leak}\Delta t}{C_{fb}} \approx \frac{2J_{leak}\Delta t}{C'_{ox}}$$

which equals $36\mu Volts$ in the worst-case ($\Delta t = 12.5\mu$seconds and $J_{leak} = 100\frac{nA}{cm^2}$). This error is negligible for an 8 bit MDAC with a 5Volt full scale output.

As was mentioned above, the op amp's offset voltage is sampled on the input capacitors during the reset phase and therefore the offset voltage is not multiplied by the MDAC gain. Since the $\frac{1}{f}$ noise of the op amp is a low frequency noise which can be viewed as a slowly varying offset voltage, the $\frac{1}{f}$ noise also appears directly at the output of the MDAC and is not multiplied by the MDAC's gain.

4.2.6. FILTERS

The filters suffer from many of the same errors as the MDAC. Unique to the SC filters are the multiplexed capacitors C_x and C_{fb}. To investigate different switching schemes, MOSFETs are included on both sides of each capacitor (see figure 3.6) [29]. The multiplexing could be accomplished with output side switches only, but this would introduce undesirable signal-dependent error as can be seen from the following: Assume only the output

side switches are used (all switches connected to the op amp's inverting input are shorted). The MOSFET output switch turns off when $V_{gs}-V_{th}=0$. Since $V_{gs}=V_{cl}-V_{out}$, the charge transferred onto C_{fb} due to C_{ov} would be $Q=C_{ov}(\Delta V_{cl}-V_{out}-V_{th})$ and the charge transferred onto C_{fb} from the channel of the MOSFET switch is

$$Q=rQ_{channel}=rC'_{ox}WL(V_{gs}-V_{th})=rC'_{ox}WL(V_{cl_{HI}}-V_{out}-V_{th})$$

where $0\leq r\leq 1$ is the fraction of channel charge dumped onto C_{fb} when the transistor turns off. r is a function of clock rise/fall times and loading on the source and drain of the transistor. If the loading at the source and drain is identical, $r\approx\frac{1}{2}$. Note that these charge quantities are dependent on V_{out}, so the errors are signal dependent and cause distortion. To eliminate such signal dependent charge transfers, a MOSFET switch is also included on the input side of each multiplexed capacitor. The input side switch is turned off before the output side switch is turned off. Since the input side switch is connected to the inverting input of the op amp which is a virtual ground, the charge transferred onto C_{fb} when an input side switch turns off is $Q=C_{ov}(\Delta V_{cl}-V_{th})+rQ_{channel}$. This error charge is dependent on V_{out} only through r since the loading on the output side of the switch is dependent on V_{out}.

Switching only on the input side is not possible due to bootstrapping. To illustrate this problem, let's consider the case of ±7.5V clocks and $V_{th}=1$V. Then, if C_{fb_k} is floating and has -5V stored on it and the output during time slot $k+1$ is -5V, the input side of C_{fb_k} would be at -10V. If MOSFET S_k has its gate at −7.5V, S_k will turning on thereby connecting C_{fb_k} into the circuit in parallel with $C_{fb_{k+1}}$. This alters the charge on both C_{fb_k} and $C_{fb_{k+1}}$ and ruins the multiplexing. This bootstrapping problem could be eliminated by reducing

the output signal swing or by increasing the clock amplitude or by adding the output side switches. Reducing the output swing reduces the dynamic range of the filters. Increasing the clock amplitude increases the amplitude of clock related errors and may not be possible if clocks are generated on chip. So the output switches eliminate bootstrapping problems and also reduce the capacitive loading on the output of the op amp.

To minimize the signal dependent errors, the following switching sequence is recommended for the multiplexed filters: 1) reset switch on, all other switches off, 2) reset off, 3) input side switch for channel k on, all other switches off, 4) output side switch for channel k on, input side switch still on, all others off, 5) input side switch turns off, output side switch still on, all others off, 6) output side switch turns off, all other switches off. Repeat cycle for channel $k+1$.

By discharging C_T at the start of each time slot, channel to channel crosstalk through C_T is eliminated. By connecting the capacitor's N+ diffusion on the output side of the op amp, problems due to non-linear junction capacitance and junction leakage current are avoided. The non-linear capacitance is driven by the output of the op amp. The junction leakage current flowing when the output side switch is on is supplied by the op amp output stage. When the multiplexed capacitors is out of the circuit, the leakage current will still flow. But it does not effect the filter at all. This can be seen by considering the Gaussian surface of figure 4.10. The analysis is simplified if the switches and op amp are assumed ideal, so we shall do so. We will look at one channel, channel k. In figure 4.10, if we start with V_{in}=0V and V_{out}=0V, the reset switch open, and the other switches on, each of the capacitors in the Gaussian surface have zero charge on them. The total charge

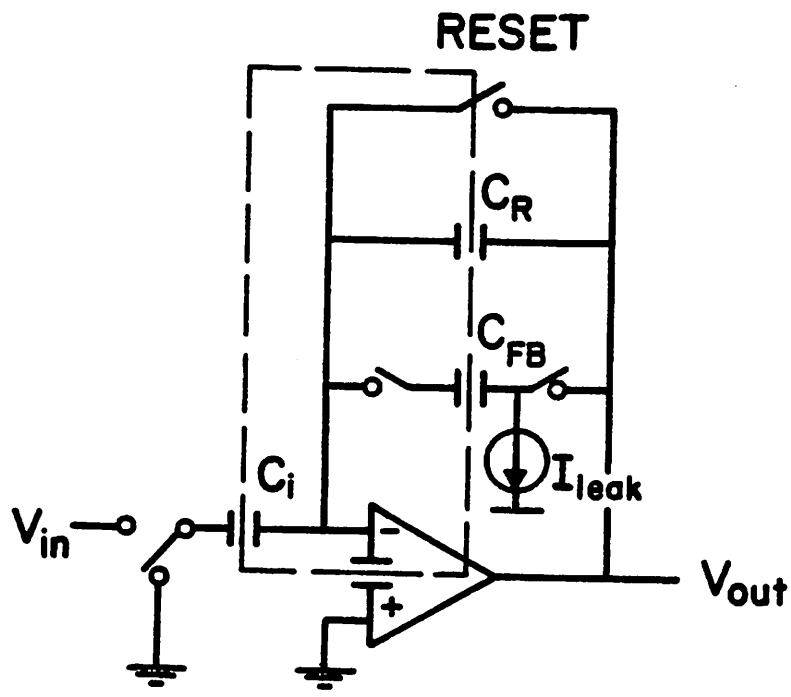


Figure 4.10. One channel of multiplexed filter showing junction leakage current

inside the Gaussian surface must remain constant since there is no current flowing into or out of this node. This total charge Q_t is initially zero. Then the switches are turned off. While channel k's switches are open, the other channels are connected into the circuit. When the switches for channel k are once again turned on, we have $V_{in}=0$ and the op amp's inverting input is a virtual ground, so the charge on C_i is zero and the charge on the op amp's input capacitance is zero. So the charge on C_{fb} and C_T, $(C_{fb}+C_T)V_{out}$, must equal zero because the total charge Q_t must remain constant. Therefore $V_{out}=0V$ so the leakage current had no effect whatsoever. (The reset switch does break the Gaussian surface and therefore does inject charge into the surface. This causes a DC offset voltage but does not invalidate the above argument.)

During the reset period, the DC offset voltage of the multiplexed filter section is equal to the offset voltage of the op amp. Otherwise, the DC output

voltage depends on the time slot. To find the the DC output voltage, we will consider the op amp and all switches except the reset switch to be ideal. Given this assumption, the DC output voltage can be found by writing charge transfer equations for a given time slot. For time slot k and sample $n+1$,

$$(C_{fb_k}+C_{T_k})V_{out}(n+1) = C_{fb_k}V_{out}(n)+\tau Q_{channel}+C_{ov}\Delta V_{cl}.$$

To find the DC output voltage, we take the limit as n goes to infinity. This gives

$$(C_{fb_k}+C_{T_k})V_{out}(\infty) = C_{fb_k}V_{out}(\infty)+\tau Q_{channel}+C_{ov}\Delta V_{cl}$$

which reduces to

$$V_{out_{DC}} = V_{out}(\infty) = \frac{C_{ov}}{C_T}\Delta V_{cl}+\frac{\tau Q_{channel}}{C_T}. \qquad (4.4)$$

The interesting result is that the DC output voltage depends only on the loss capacitor C_T and not on C_{fb}. This makes sense because, for a given filter, C_{fb} is always connected into the circuit and acts like an open circuit for DC signals.

The finite gain of the op amp causes a slight reduction of the magnitude of the filter's pole location. If the op amp gain is infinite, the pole is $z=\frac{C_{fb}}{C_{fb}+C_T}$. If the op amp has an open loop voltage gain of α, the pole moves to

$$\frac{C_{fb}}{C_{fb}+C_T}\,\frac{1+\frac{1}{\alpha}}{1+\frac{1}{\alpha}\left[\frac{C_i}{C_{fb}+C_T}\right]}.$$

For $\alpha>1000$, the shift in the pole location is negligible.

The op amp's $\frac{1}{f}$ noise and op amp's offset voltage are sampled onto C_i during the reset period. Then, when the reset switch opens, C_i retains the sampled $\frac{1}{f}$ noise and offset voltage. Since the $\frac{1}{f}$ noise changes very slowly

and the offset voltage is constant, both are effectively canceled. The autocorrelator's output noise is dominated by the third filter section due to the low sampling rate.

4.3. IC LAYOUT

The layout was made as compact as possible while maintaining isolation between analog signal paths and clock signals wherever necessary. The op amp layout is shown in figure 4.11. The input transistors are the large polygons. Ideally, the input transistors would be circular. This assures good matching independent of mask alignment errors in either the X or Y direction. The polygons are an approximation to a circle; we could not make circles with our mask making equipment. The op amp occupies $450 mil^2$ of area.

To minimize the area of the layout, the switches in the SC circuits were realized with NMOS transistors. The layout of an NMOS transistor can be seen in the op amp layout. The gate metal was extended on both sides of the active area to minimize any leakage currents between the source and drain (this is important when the NMOS transistor is used as a switch). To be safe, P+ isolation diffusions are placed between any two N+ diffusions which are not connected. This costs area but was strongly recommended by former users of this CMOS process [28].

Values for the capacitors on the chip were chosen as small as possible to minimize total chip area. But small capacitors make parasitic charge transfer errors more noticeable (see section 4.2.5). For the MDAC, C_1=0.1pF was chosen. For the multiplexed filters, C_T=0.22pF for section 1 and 2 and C_T=0.66pF for section 3. The third filter section operates at a lower clock rate than the other two sections (800Hz rather than 8kHz) and therefore requires smaller capacitor ratios to achieve the desired low frequency cutoff.

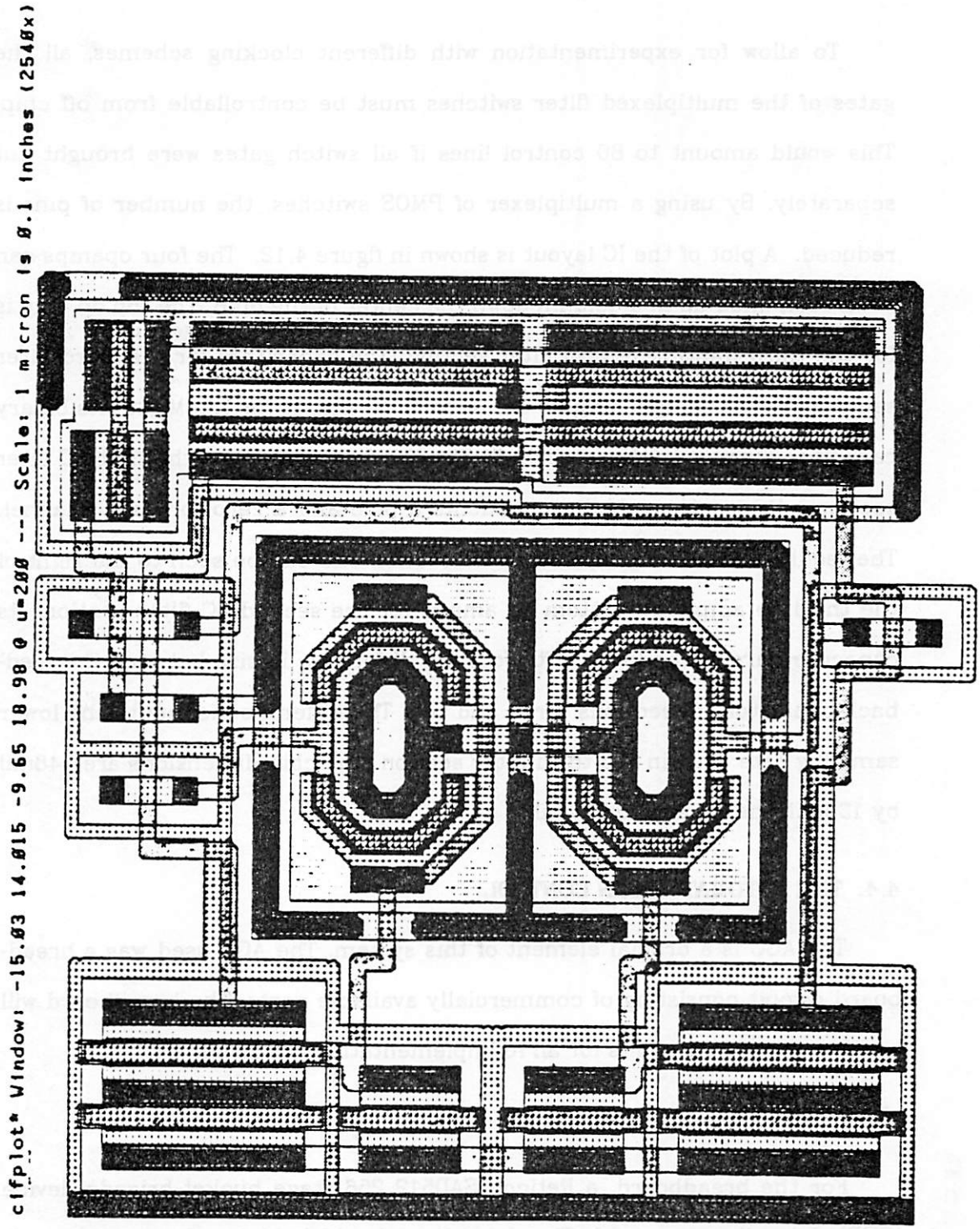


Figure 4.11. Op amp layout

To allow for experimentation with different clocking schemes, all the gates of the multiplexed filter switches must be controllable from off chip. This would amount to 80 control lines if all switch gates were brought out separately. By using a multiplexer of PMOS switches, the number of pins is reduced. A plot of the IC layout is shown in figure 4.12. The four opamps can been seen lined up in a column down the center of the chip. The top op amp is for the third multiplexed SC filter section. The capacitors for the third filter section are in the upper right. The second op amp is for the MDAC; its binary capacitor array is in the upper left. The third op amp is for the first SC filter section - the section which implements a pole and a zero for each channel. The feedback and zero capacitors for section one can be seen to the right of the third op amp. The bottom op amp is for the second SC filter section. Its capacitors are in the lower left. Note the difference in size between the feedback capacitors of sections three and two. The difference is due to the lower sampling rate used in the third filter section. The chip dimensions are 148mil by 133mil, the area is 19,700mil^2.

4.4. THE AUTOMATIC GAIN CONTROL

The AGC is a crucial element of this system. The AGC used was a breadboard circuit consisting of commercially available parts. The breadboard will be described and circuits for an IC implementation are given.

4.4.1. AGC BREADBOARD

For the breadboard, a Reticon SAD512 256 stage bucket-brigade device (BBD) delay line was used. The delay line should hold one frame of speech which is 100 samples for this system. By operating the BBD at a 16kHz sampling rate, the number of samples held is 128 which is as close to the desired

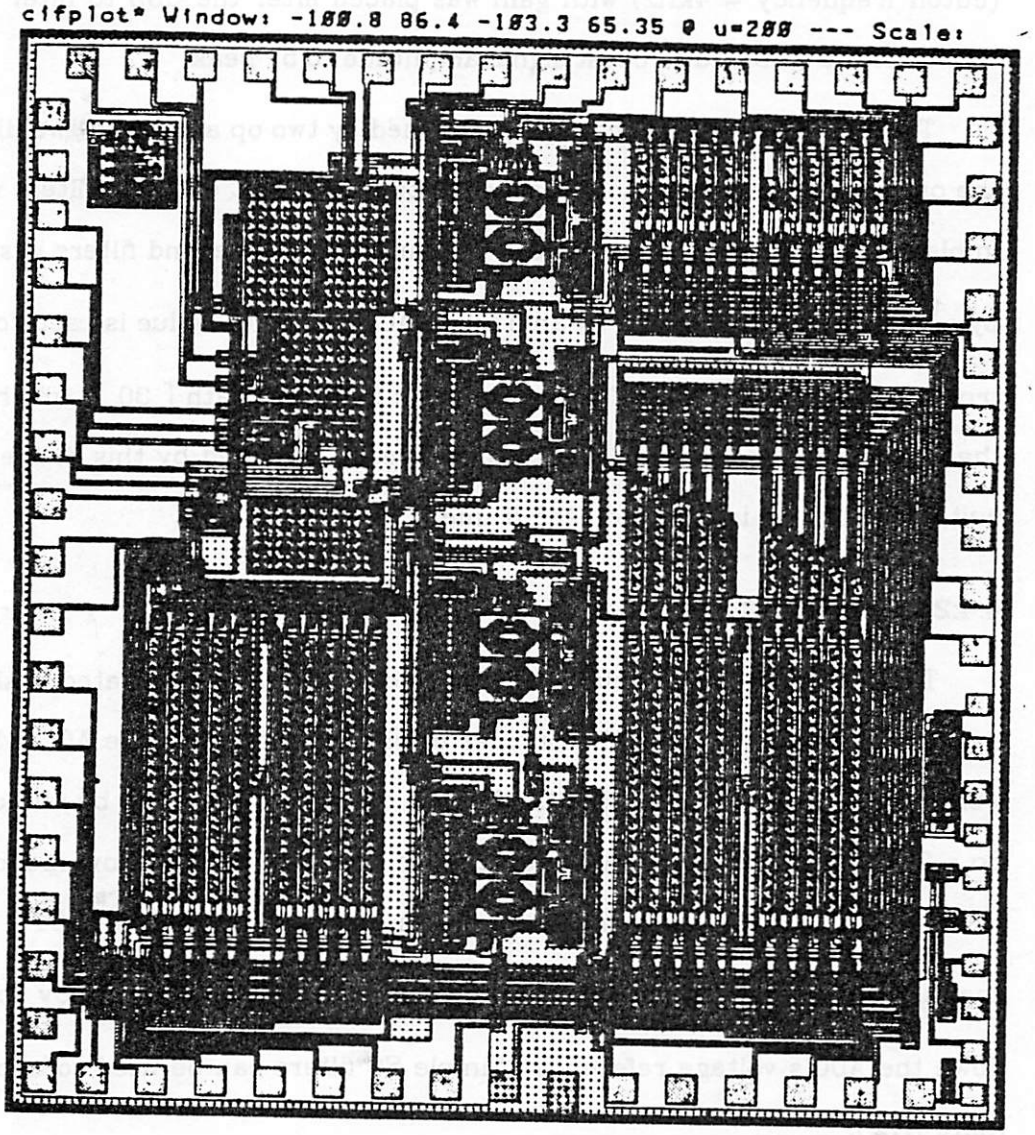


Figure 4.12 Layout of the autocorrelator IC

value as is practically possible with the available BBDs. Since the speech was bandlimited to 4kHz prior to the AGC, the signal out of the BBD has no significant signal in the 4kHz to 8kHz region and therefore the output of the BBD can be sampled at 8kHz without filtering. A one pole low pass RC filter (cutoff frequency = 4kHz) with gain was placed after the BBD to filter noise output of the BBD and to boost signal amplitude to 5V peak.

The absolute value function is performed by two op amps and two diodes, the output of which is fed to a cascade of two simple SC one pole filters which implement $H(z)$ of equation (2.10). The output of the second filters is scaled by $\frac{4\sqrt{2}}{H(1)}$ and then digitized by an 8 bit MDAC. The digital value is used to control a divider circuit (MDAC in an op amp's feedback path [30]) which sets the gain of the upper path. The delayed speech is scaled by this divider circuit, and the gain is changed each frame.

4.4.2. An MOS AGC

The ADC, MDAC, BBD delay line, and filters can all be integrated easily on an MOS IC. Simulations on the number of bits necessary for the AGC's MDAC were performed [18], and 5 bits of linear coding was found to be sufficient. So a 3 or 4 bit logarithmic ADC would serve adequately. These low speed, low resolution ADCs are easily implemented in a small silicon area [31]. The scaling of the filter output by $\frac{4\sqrt{2}}{H(1)}$ can be accomplished simply by scaling down the ADC's voltage reference. Simple SC filters can be used to approximate $H(z)$.

The absolute value function is the only element of the AGC that can not be directly copied from the breadboard onto an IC because floating diodes are not available in a standard CMOS process. Therefore, diode-connected

enhancement-mode NMOS transistors are used instead. A full wave rectifier circuit (absolute value circuit) is shown in figure 4.13. It requires matched resistors and an op amp with low offset voltage, but the requirements are not strict since the AGC scaling need not be exact. Resistor matching of a few percent and 50mV of offset are adequate.

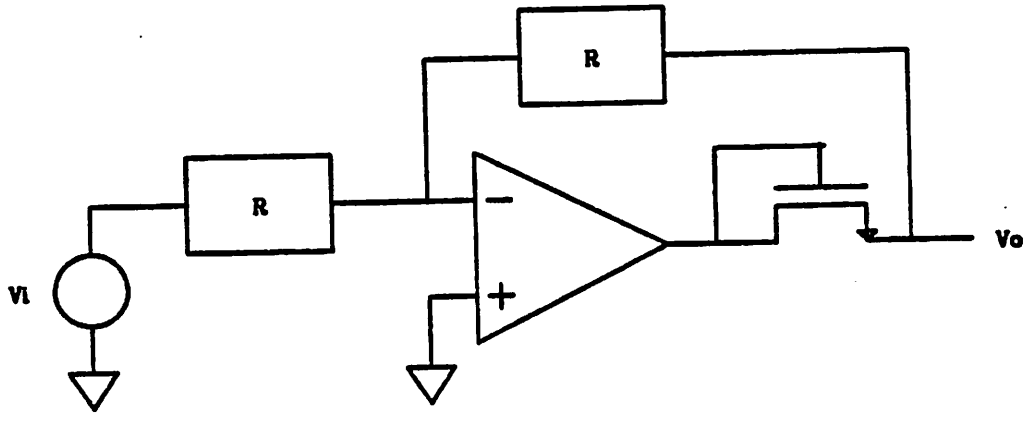


Figure 4.13. MOS absolute value circuit

CHAPTER 5

EXPERIMENTAL RESULTS

To test the system prior to integrating the autocorrelator, a breadboard was built. The breadboard was used extensively, both to test the SC multiplexed filters and to test the system's ability to function as an LPC speech analyzer. Then the SC autocorrelator IC was fabricated with a metal gate CMOS process and tested. For comparison, two sets of multiplexed filters were integrated on a polysilicon gate CMOS process. Results are given for the breadboard system, the IC autocorrelator, and the poly gate multiplexed filters.

5.1. THE BREADBOARD SYSTEM

5.1.1. AUTOCORRELATOR

An autocorrelator was constructed of commercially available parts to test various aspects of the design prior to integration of the autocorrelator. This was a necessary phase of the design since SPICE does not accurately model the MOSFET channel charge redistribution during turn off (see section 4.2.5). For the breadboard, capacitors were hand measured to assure better than 1% ratio accuracy. For all filters, switches were included only on the output side of the multiplexed capacitors (figure 3.6).

Electrical data taken from the breadboard is tabulated in table 5.1. The autocorrelator did provide adequate accuracy for speech analysis despite the large amount of adjacent channel crosstalk in the multiplexed filters. The crosstalk is due to stray wiring capacitance on the breadboard which couples

Autocorrelator Breadboard Data	
parameter	measured value
Total system S/N ratio (ADC, MDAC, and filters)	64dB
MDAC bits	12
Cutoff frequency of each filter section	50Hz
Cutoff frequency of filter sections cascaded	25Hz
Adjacent channel crosstalk of multiplexed filters (1V peak sine wave at 10Hz)	-36dB
Output swing for HD_2=1% (10Hz sine wave input)	±5V
Full scale value of R(0)	5V
Clock frequency (10 channels)	80kHz
Power supply	±8*V*, ±15*V*

Table 5.1. Data from autocorrelator breadboard

the adjacent signal paths. The dynamic range of 64dB was sufficient for speech analysis; the noise was due to analog noise sources and digital noise coupling into the analog signal lines on the breadboard. To test the breadboard with speech input, the breadboard was fed constant vowel sounds and the analog autocorrelation values output by the breadboard were digitized by a 12 bit ADC and stored on a mini-computer. The LPC spectrum for the autocorrelation values was plotted against the fft of the vowel sound to check for errors. As long as R(0) was large enough so that quantization errors due to digitization of the autocorrelation values were negligible (i.e. R(0) greater than 0.4V), the results were very good as can be seen in figures 5.1 through 5.4. In each figure, the upper plot is the LPC spectrum generated from data taken from the breadboard, and the lower plot is the LPC spectrum generated by computer simulation of the autocorrelation LPC algorithm. The LPC spectrum is plotted with the solid line, the fft of the sound with a broken line.

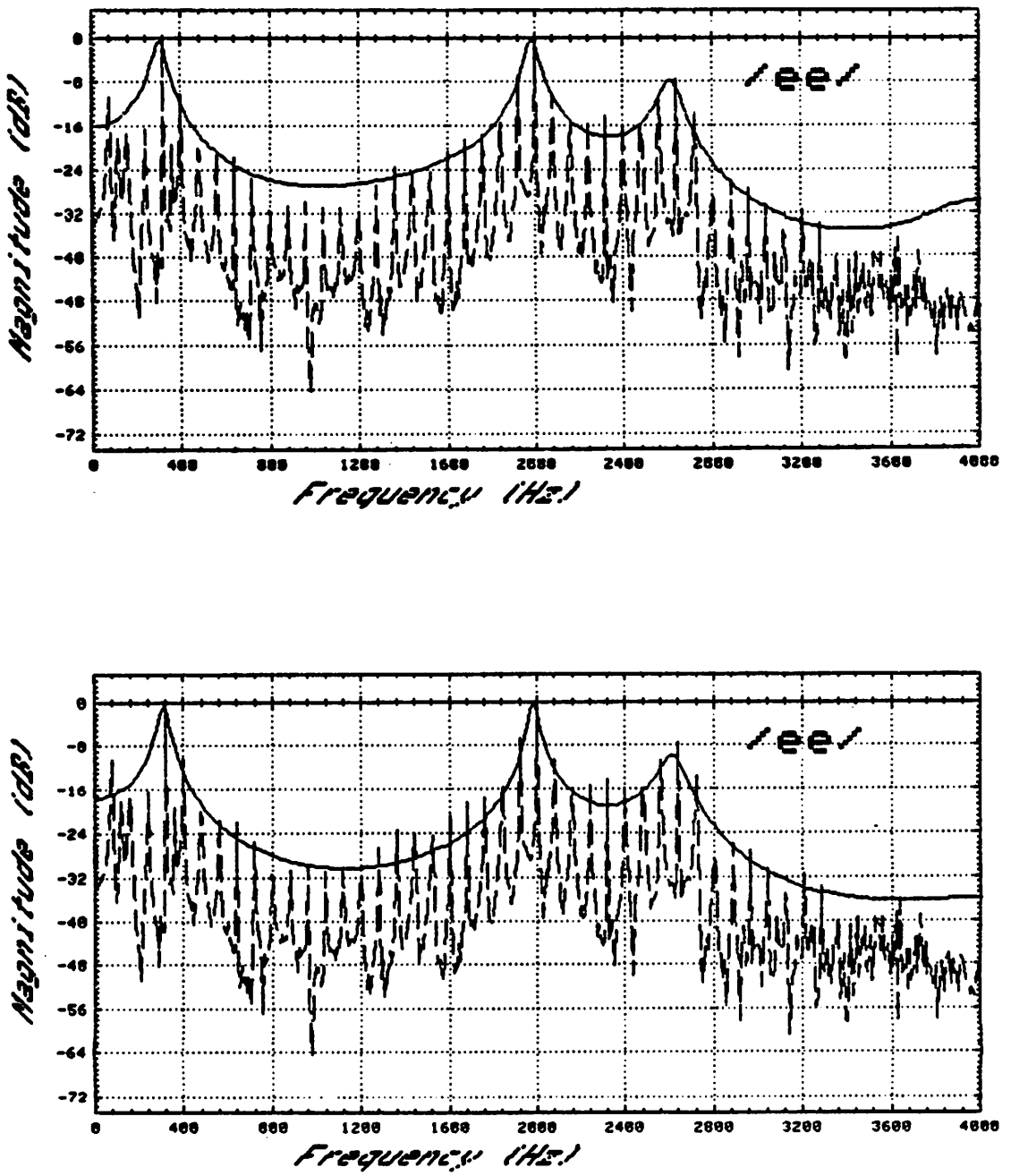


Figure 5.1. top: LPC fit to /ee/ sound, from breadboard data
bottom: LPC fit to /ee/ sound, computer simulation

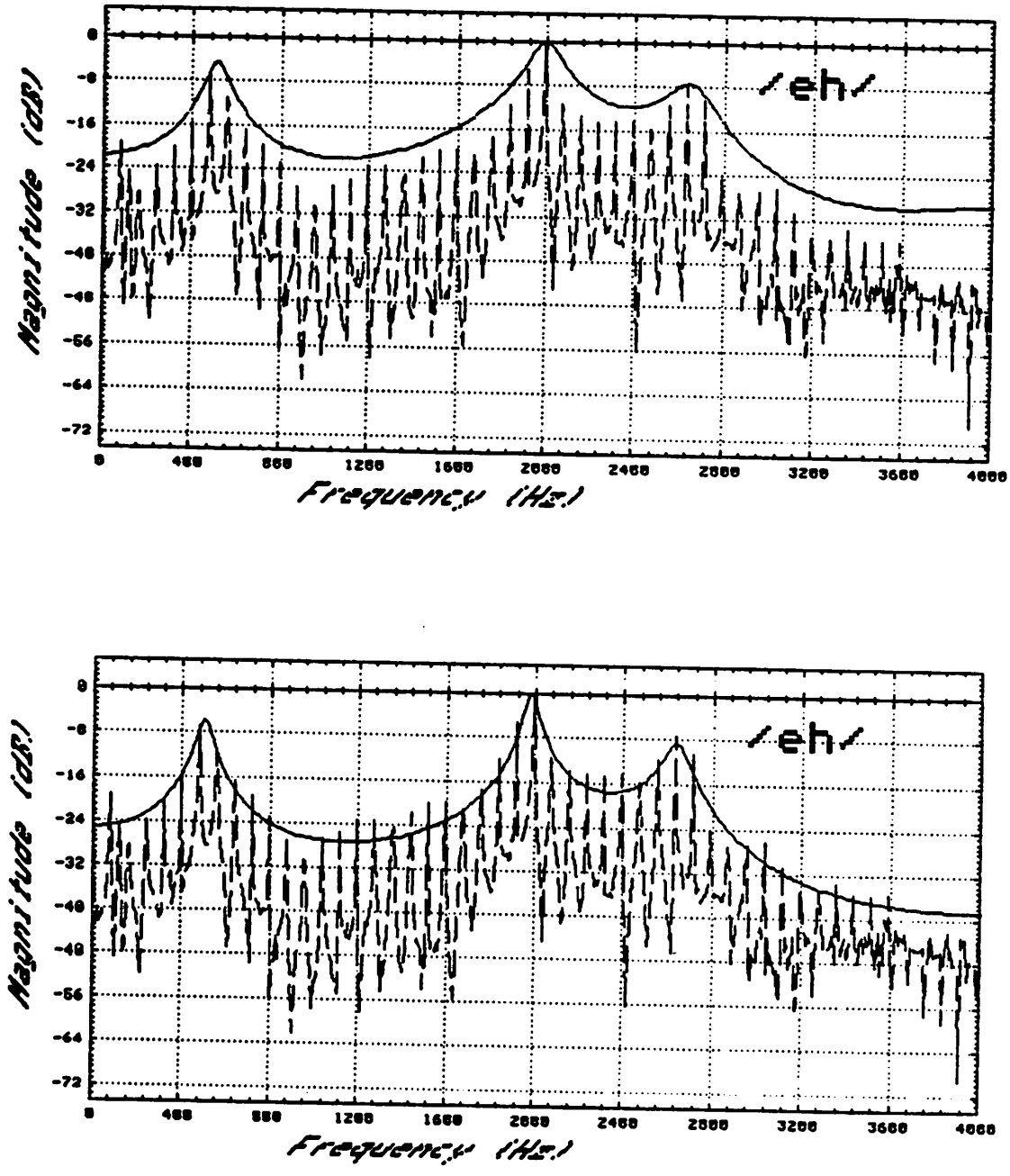


Figure 5.2 top: LPC fit to /eh/ sound, from breadboard data
bottom: LPC fit to /eh/ sound, computer simulation

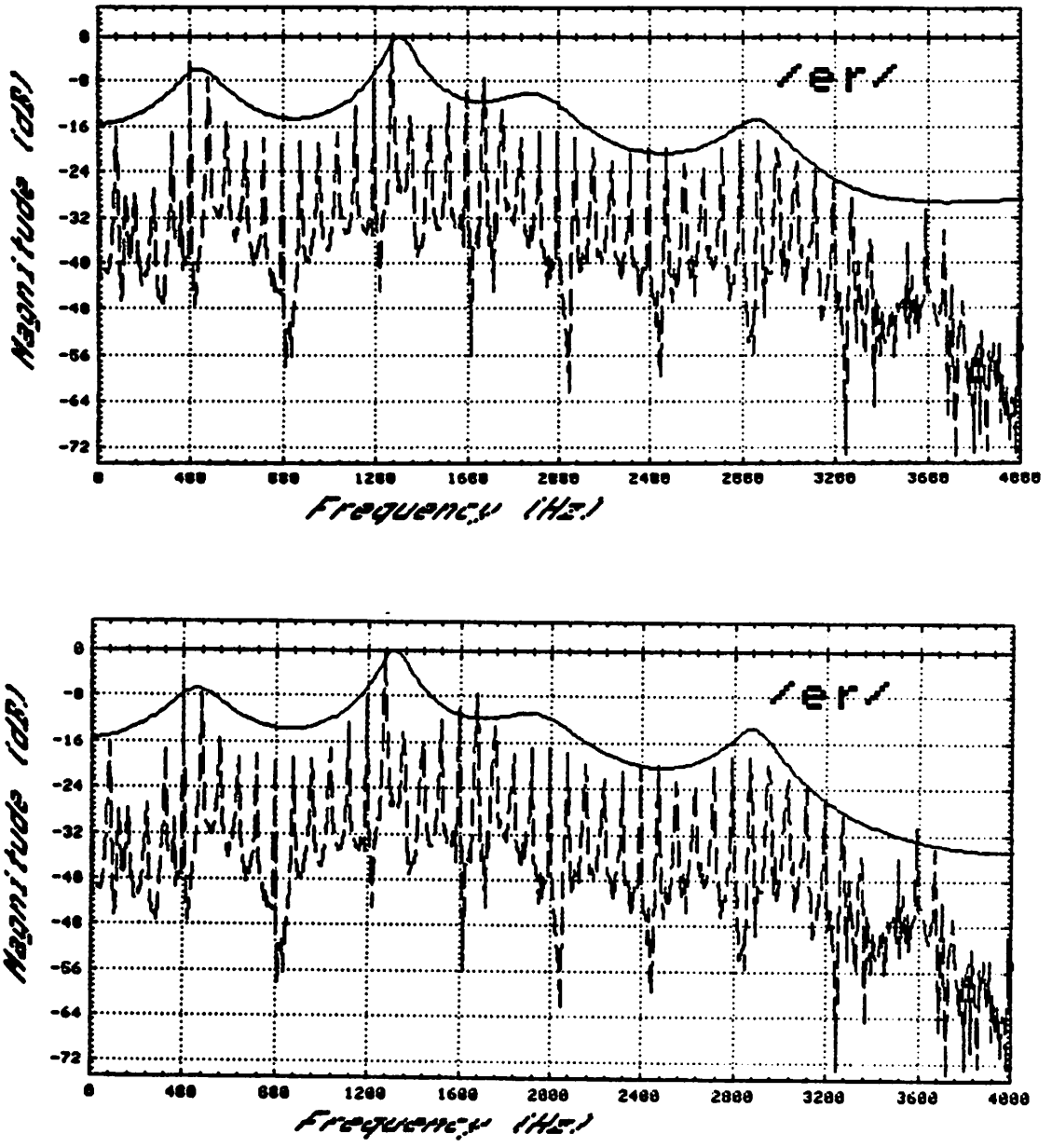


Figure 5.3 top: LPC fit to /er/ sound, from breadboard data
bottom: LPC fit to /er/ sound, computer simulation

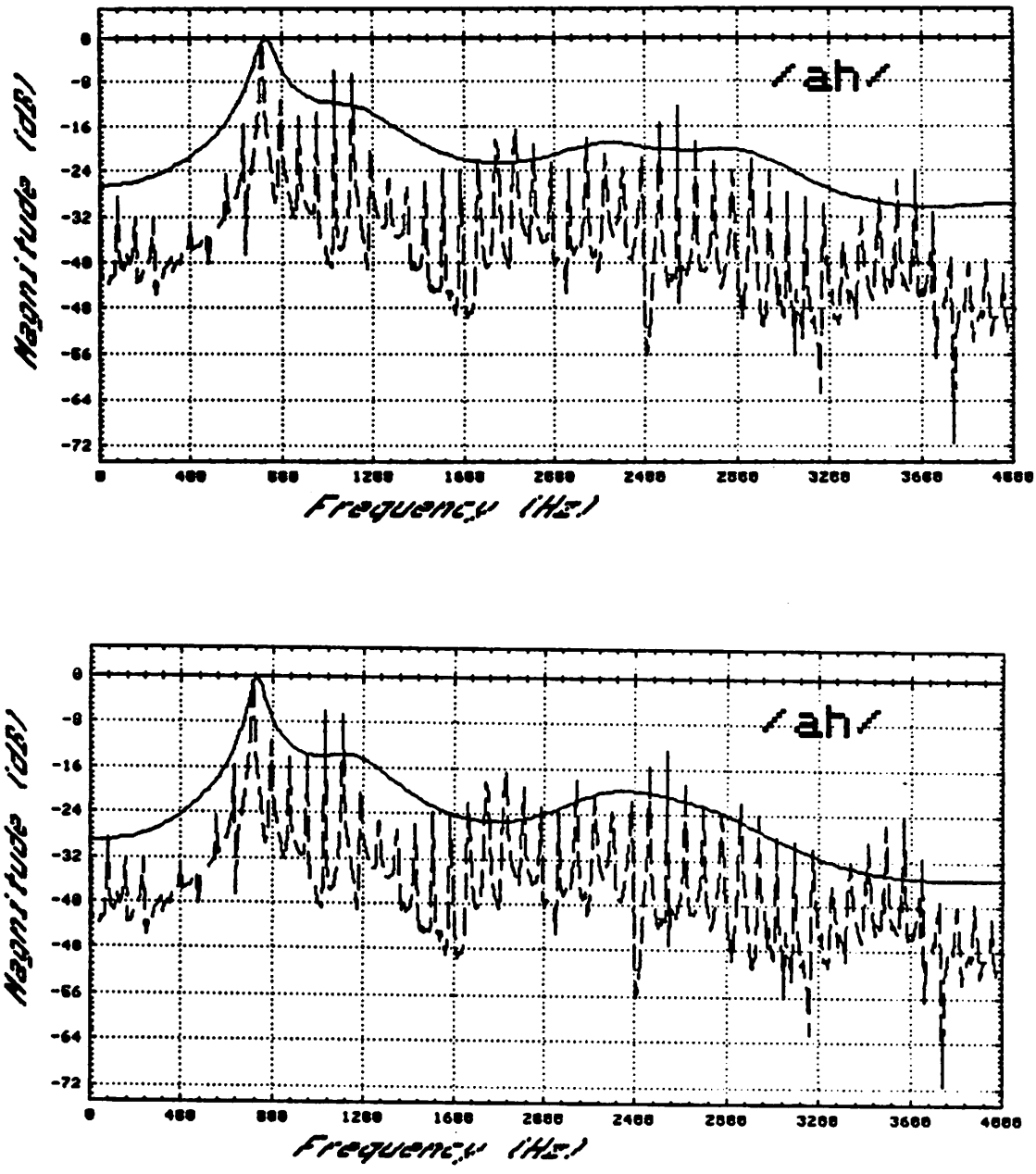


Figure 5.4 top: LPC fit to /ah/ sound, from breadboard data
bottom: LPC fit to /ah/ sound, computer simulation

As a further check of the breadboard analyzer, the autocorrelation values were used to synthesize speech. First, vowel sounds were analyzed, synthesized, and then played back. The synthesis was compared to the original and found to be indistinguishable. Then, sentences were fed through the analyzer and then synthesized. Due to occassional unstable frames (caused by excessive quantization of the autocorrelation values) and a poor pitch tracker, the synthetic speech sounded much worse than the original. Nonetheless, all synthetic speech was understandable and considered acceptable by the listeners. A better pitch tracker would have greatly improved the quality of the synthetic speech.

5.1.2. THE DURBIN RECURSION MICROPROCESSOR

The only question remaining in the microprocessor design was the speed of execution of the Durbin algorithm. The system frame rate is 80Hz which allows 12.5msec for execution of the algorithm. The execution time was found to be in the range of 10msec to 11msec when the 8088 clock frequency was 5MHz. The execution time varies from frame to frame depending upon the number of calls of the 24 bit by 16 bit multiplication routine, which is dependent on the data and the number of a_i which exceed 16 bits. Since 8088s which operate with an 8MHz clock are now available, the execution time could be cut to less than 7msec without changing the system at all. Frame periods are typically 10-50msec so the simple 8088 system (or equivalent) is adequate for all applications.

5.1.3. THE AGC

The AGC was constructed as discussed in chapter 4 - bucket-brigade delay line, full-wave rectifier, SC filter, ADC, MDAC. The AGC was found to work

very well, giving a constant autocorrelator output over an input range of 32dB. The dynamic range of the AGC was limited by the BBD delay line noise. The BBD delay line had a dynamic range of only 34dB (16kHz sampling rate). If the BBD noise had not been a factor, the autocorrelator with the AGC would have had a useful input range of roughly 60dB (32dB from the AGC plus 31dB from the autocorrelator). Without the AGC, the autocorrelator's input range was limited by the autocorrelator's dynamic range to 31dB. But the value of R(0) varies 62dB as the input varies 31dB due to the squaring of the input signal. Therefore small input signals result in very small R(0) values which are heavily quantized by the following ADC. So the autocorrelator's useful input range for LPC speech analysis without the AGC was actually about 10dB. So the AGC was a very important addition to the system because it increased the useful input range for speech signals from 10dB to 32dB.

As was mentioned in the previous chapter, the BBD delay line held 128 samples which is 28 more than one frame. This is unfortunate but acceptable. A 100 stage BBD could be integrated or the frame length could be changed to 128 samples (frame rate=62.5Hz). This was not done because the AGC was added to the system after the autocorrelator and microprocessor systems had been built, so the frame length was already set at 100 samples.

5.2. AUTOCORRELATOR IC RESULTS

The MDAC and multiplexed SC filters for the autocorrelator were integrated on a metal gate CMOS IC. To allow flexibility in testing, all outputs and inputs were brought out to pins on the IC package so that the MDAC and each filter section could be tested individually. The IC was fabricated in the integrated circuit laboratory at U.C. Berkeley (see appendix D for process schedule). The yield was very poor, only a few working chips were available

for testing after numerous runs in the fabrication lab.

5.2.1. MDAC

The MDAC was tested, and the data is presented in table 5.2. The integral non-linearity was larger than expected. This can be attributed to oxide gradients (the layout does not employ concentric placement of the capacitor array), parasitic capacitances, and rough metal edges (metal defined the capacitors in the MDAC). This non-linearity could be eliminated by more careful layout [32].

The reset feedthrough, which contributes to the DC output offset voltage of the MDAC, was also larger than expected. When the reset switch turns off (see figure 3.3), its channel charge must go somewhere. The charge can flow to the op amp output node, to the op amp inverting input node, or to the substrate. It was expected that most of the charge would go to the op amp output node, but this was not the case. In fact, most of the channel charge flowed onto the inverting node, as can be seen from the following calculations: The DC output voltage due to channel charge is (from equation 4.3)

MDAC Data	
parameter	measured value
Integral non-linearity	2 LSB
Differential non-linearity	0.9 LSB
Settling time (0.1%)	2μsec
Reset feedthrough at output	30mV
Output swing	+3V,-5V
S/N ratio	66dB
Load capacitance	$\approx 5pF$
Clock frequency	80kHz
Power supply	$\pm$7.5V
Power dissipation	15mW

Table 5.2 Data for the 8 bit SC MDAC

$$V_{out_{CC}} = r\frac{Q_{channel}}{C_{fb}} = \frac{rC'_{ox}WL(V_{gs}-V_{th})}{C_{fb}} \approx r\,37mV$$

for the MDAC with NMOS reset switch. The contribution to the DC output voltage due to the overlap capacitance is (also from equation 4.3)

$$V_{out_{OV}} = \frac{C'_{ox}WL_D\Delta V_{cl}}{C_{fb}} \approx 2.3mV.$$

(the gate overlap was estimated to be L_D=1μm under the microscope). Comparing these results, we see that the observed feedthrough at the output (30mV) is mainly due to channel charge from the reset switch being trapped onto the op amp's inverting node. From the experimental data, we can estimate $r \approx 0.75$. The value of r is dependent on the impedance at the output of the op amp as well as the clock fall time (the reset clock line was driven by a 74C08 CMOS AND gate). A value of r near zero would be desirable, but any value of r is acceptable since the feedthrough is the same every cycle and therefore only contributes to the DC output voltage.

Figure 5.5 shows oscilloscope photos of the MDAC output. The top photo shows the MDAC squaring a triangle wave (upper trace), thereby producing a parabola wave output (lower trace). The lower photo shows the MDAC output (lower trace) settling to its final value during clock phase Φ_2 (upper trace).

5.2.2. MULTIPLEXED SC FILTERS

The filters were tested individually. All three filter sections are different - section one implements ten poles and ten zeros, section two implements ten poles, section three also implements ten poles but with different capacitor ratios due to downsampling.

The clocks which were used for testing the filters are shown in figure 5.6. In figure 3.6, the switch phasing is shown as Φ_A and Φ_B. This labeling is used rather than Φ_1 or Φ_2 or Φ_3 because the best switch phasing was not known

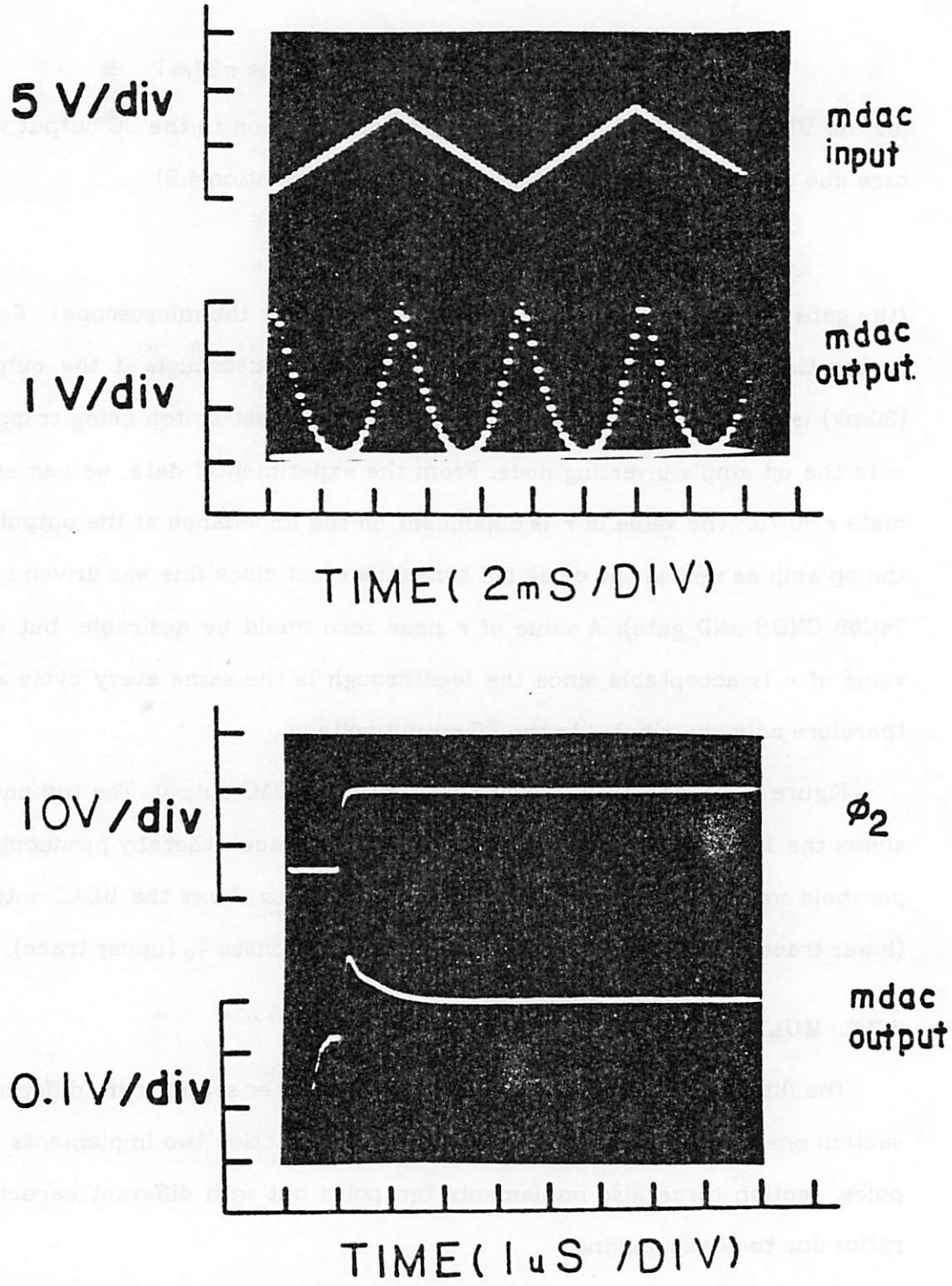


Figure 5.5 top: MDAC squaring a triangle wave
bottom: MDAC output settling

before the circuit was fabricated although the phasing $\Phi_A=\Phi_3$ and $\Phi_B=\Phi_2$ was suggested in chapter 4. Note that Φ_1 (reset) and Φ_2 are non-overlapping clocks, and Φ_3 goes low before Φ_2 goes low. The clock amplitudes are $\pm 7.5V$ ($+7.5V$ is a logic HI level and turns on an NMOS switch).

To find the best clocking scheme, the filters were tested with the following clocking schemes: 1) $\Phi_A=\Phi_3$ and $\Phi_B=\Phi_2$, 2) $\Phi_A=HI$ and $\Phi_B=\Phi_2$, and 3) $\Phi_A=\Phi_2$ and $\Phi_B=HI$. The results using clocking schemes (1), (2), and (3) were practically identical except for the DC output voltages. Also, with clocking scheme (3) the first two filter sections did not work properly, probably due to the excessive capacitive loading on the op amp output from all ten feedback capacitors' backplate diodes. But section three did work with clocking scheme (3). The measured data is listed in table 5.3. Distortion and crosstalk figures were measured with a 10Hz sine wave input signal. For the crosstalk

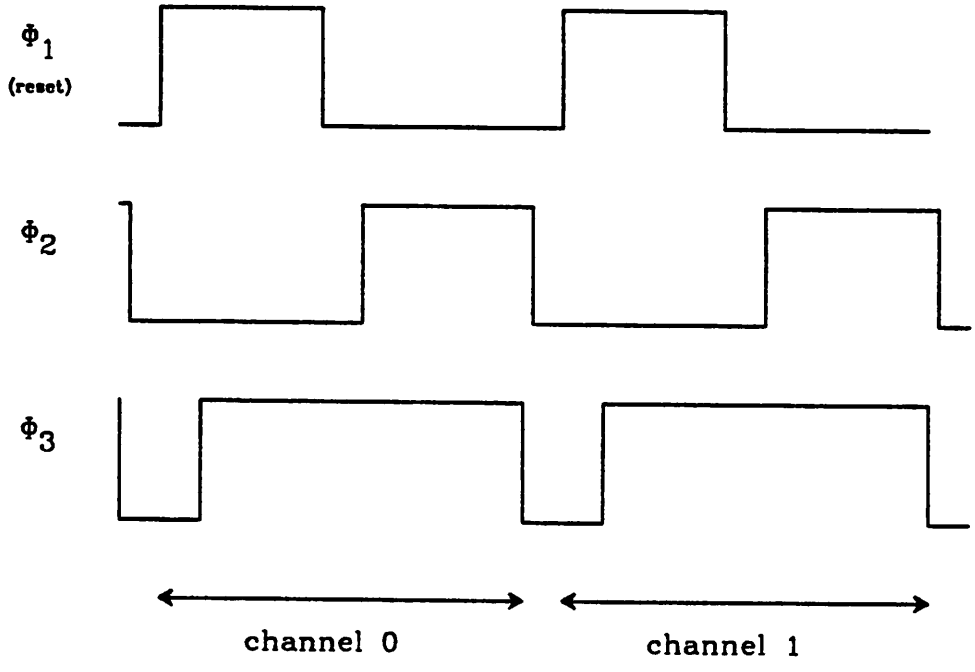


Figure 5.6 Clocks for testing the multiplexed filters

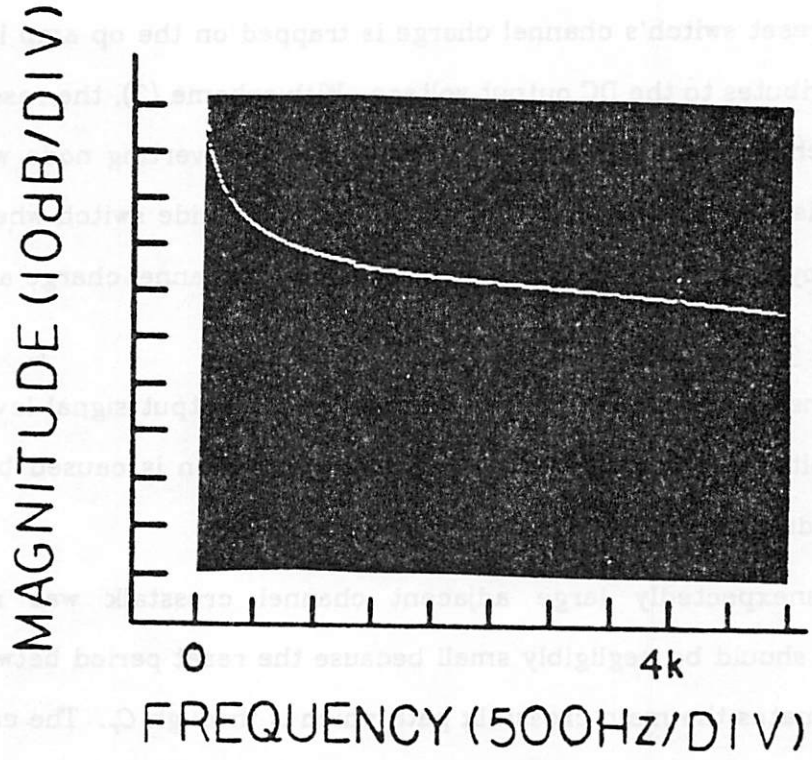
measurement, the channel being tested had its input grounded and all other channels were fed a 1V peak sine wave.

For all cases (except switching scheme (3) for filter sections 1 and 2), the filter characteristics were very accurate, see figure 5.7. The frequency response of the different channels matched well (better than 0.2dB) due to the use of repeated unit capacitors in the layout.

For a given clocking scheme, the DC output voltage for filter section 3 is smaller than for sections 1 and 2 because C_T=0.22pF in sections 1 and 2 and C_T=0.66pF for section 3 (equation 4.4). As with the MDAC, the DC offset voltage of the filters was mainly due to the transistor's channel charge. For all filter sections, the DC offset voltage for clocking scheme (3) is smaller than that of scheme (1) or (2). With schemes (1) and (2), the channel charge for the input side switches is provided by the output of the op amp. The charge flows through the reset switch during the reset period. Then, when reset goes

Measured Filter Characteristics (metal gate process)		
parameter	filter sections 1 & 2 (zeros off)	filter section 3
-3dB frequency	50Hz	50Hz
Output swing for HD_2=1%	±1V	±4V
S/N ratio	68dB	70dB
Adjacent channel crosstalk	-44dB	-48dB
Clock frequency (10 channels)	80kHz	8kHz
Sampling rate for each channel	8kHz	800Hz
Average DC output voltage of all channels (input grounded)		
$\Phi_A=\Phi_3$, $\Phi_B=\Phi_2$	2.8V	0.7V
$\Phi_A=HI$, $\Phi_B=\Phi_2$	2.3V	0.4V
$\Phi_A=\Phi_2$, $\Phi_B=HI$	0.4V	-0.18V
Range of DC output offset voltages, all channels		
$\Phi_A=\Phi_3$, $\Phi_B=\Phi_2$	440mV	40mV
$\Phi_A=HI$, $\Phi_B=\Phi_2$	200mV	18mV
$\Phi_A=\Phi_2$, $\Phi_B=HI$	50mV	20mV

Table 5.3 Multiplexed Filter Characteristics, Metal Gate Process



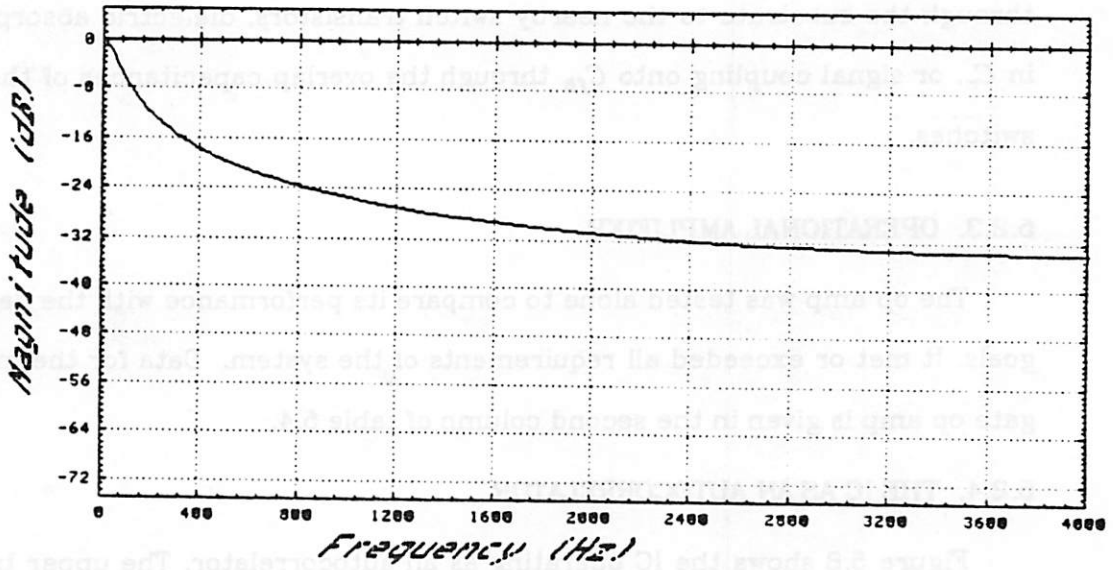


Figure 5.7. top: Frequency response of multiplexed filter section 1, channel 0
bottom: Computer simulation of the above filter

low, the reset switch's channel charge is trapped on the op amp input node and contributes to the DC output voltage. With scheme (3), the reset switch's channel charge which is trapped on the op amp inverting node when reset goes low is used to form the channel for the input side switch when it turns on, thereby giving a first order cancellation of the channel charge and a lower DC output voltage.

The harmonic distortion is a function of the output signal level and DC output voltage of the filter. The harmonic distortion is caused by channel charge redistribution which is signal dependent.

An unexpectedly large adjacent channel crosstalk was measured. Crosstalk should be negligibly small because the reset period between channels eliminates the main crosstalk path which is through C_τ. The crosstalk is either due to poor isolation between nearby N+ diffusions, charge pumping through the substrate to the nearby switch transistors, dielectric absorption in C_τ, or signal coupling onto C_{fb} through the overlap capacitances of the off switches.

5.2.3. OPERATIONAL AMPLIFIER

The op amp was tested alone to compare its performance with the design goals. It met or exceeded all requirements of the system. Data for the metal gate op amp is given in the second column of table 5.4.

5.2.4. THE IC AS AN AUTOCORRELATOR

Figure 5.8 shows the IC operating as an autocorrelator. The upper trace is the output of the MDAC. The ten products can be clearly seen. The lower trace is the output of the filters. The output is a constant autocorrelation waveform. R(0) is at the left, R(9) is at the right. In both traces, the reset

Measured Op Amp Characteristics		
parameter	Metal Gate	Poly Gate
Gain	8,000 V/V	1,000 V/V
Maximum output swing	±5.5V	+1 V, −3 V
Noise	100nV/√Hz	580nV/√Hz
Noise 1/f corner frequency	700Hz	1kHz
Power Supply	± 7.5V	± 6V
I_{BIAS}	150μA	300μA
Power Dissipation	10mW	18mW
Number of transistors	14	14
Size	450 mil²	460 mil²

Table 5.4. Op Amp Data, Metal Gate and Poly Gate Processes

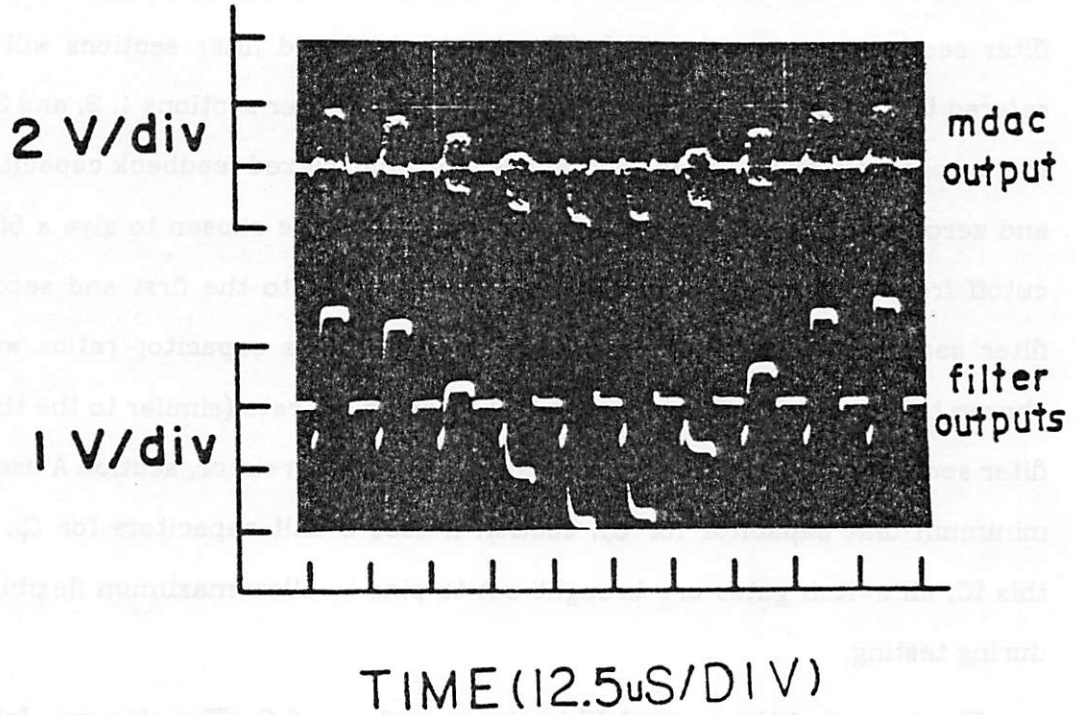


Figure 5.8. SC autocorrelator output with sine wave input

period can be seen between channels. The input signal is a 1kHz sine wave, so the output autocorrelation function is samples of a sine wave.

Unfortunately, the IC autocorrelator did not provide enough accuracy for analyzing speech due to the excessive crosstalk in the filters and the MDAC non-linearity. Also, due to low yield, no chip was found which had a filter sec-

tion 1 with all 10 zeros functioning. Speech was passed through the IC autocorrelator, but the autocorrelation values usually yielded an unstable LPC model. Nonetheless, the autocorrelation function was fairly accurate, and the IC could have been used for voiced/unvoiced decision making or other low accuracy applications.

5.3. A SECOND TEST CHIP

Since excessive crosstalk and offset voltages were measured and since the zeros were not all functional, a second IC consisting of two multiplexed filter sections was constructed. (The two multiplexed filter sections will be refered to as A and B to avoid confusing them with filter sections 1, 2, and 3 of the autocorrelator IC.) Each section has four multiplexed feedback capacitors and zero capacitors. Section A's capacitor ratios were chosen to give a 50Hz cutoff frequency at an 8kHz sampling rate (similar to the first and second filter sections of the autocorrelator IC). Section B's capacitor ratios were chosen to give the 50HZ cutoff at an 800 Hz sampling rate (similar to the third filter section of the autocorrelator). As in the autocorrelator, section A uses a minimum unit capacitor for C_T, section B uses 3 unit capacitors for C_T. In this IC, all switch gates are brought out to pins to allow maximum flexibility during testing.

The layout of the second IC is shown in figure 5.9. The chip was fabricated on a digital CMOS process with N type substrate, polysilicon gate, with poly-poly capacitors. The op amp is practically identical to the op amp described in chapter 4 with all devices replaced by their complement. This was done to assure that current sources were implemented by NMOSFETs because they are in a well and have a higher output impedance than the PMOSFETs. For the filters, PMOSFETs were used as switches since they do not

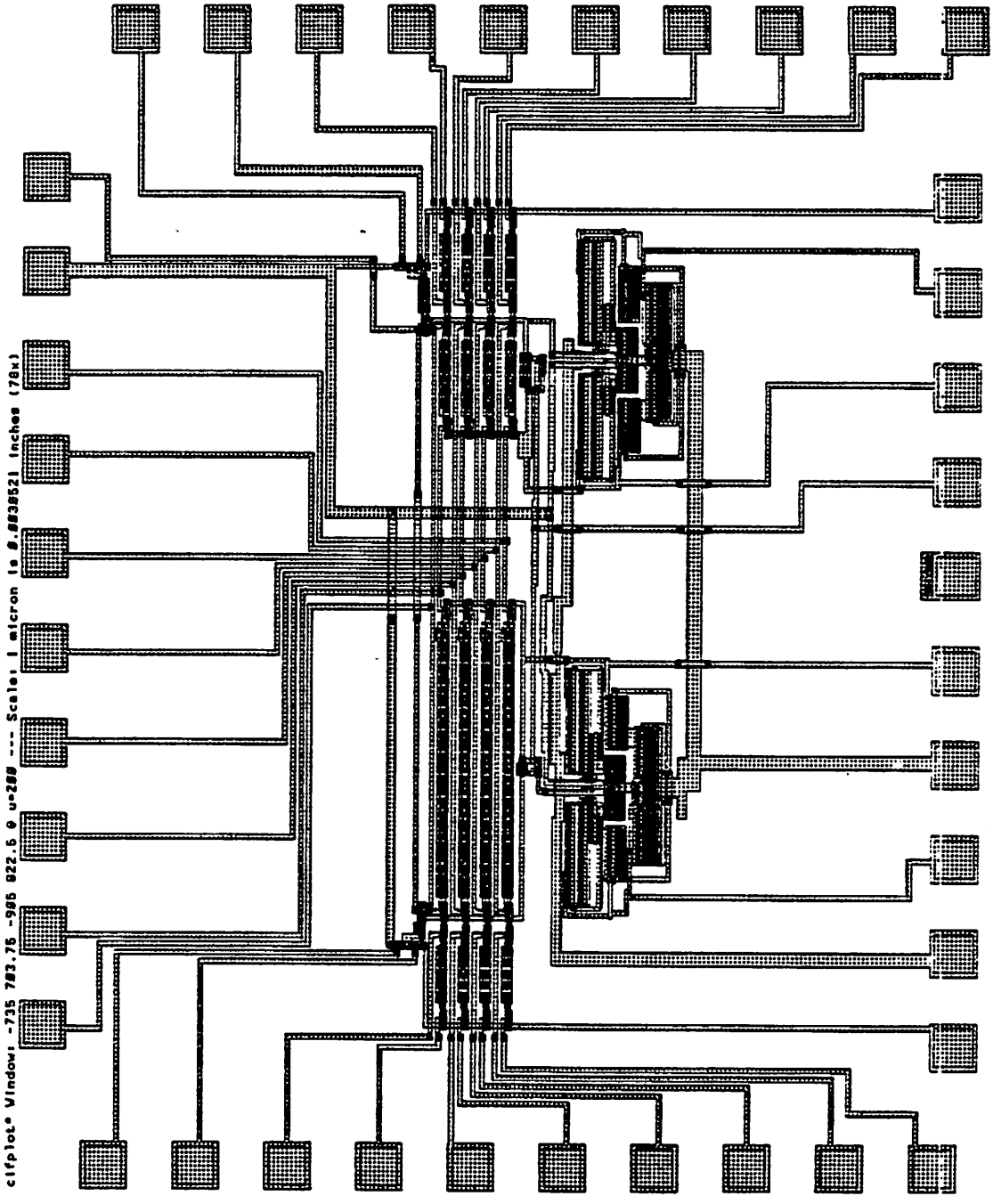


Figure 5.9. Layout of the poly gate multiplexed filters

require a well.

5.3.1. RESULTS

The poly gate ICs were extensively tested. The yield was excellent for these chips (9 out of 10 worked). This reflects the clean, reliable processing done in industry. Again, to check the effects of various switching schemes, the multiplexed filters were tested with all possible clocking schemes. The clocks used are an inverted version of those shown in figure 5.6 because the switches are PMOS. (A logic HI level is taken to be the voltage required to turn on a switch, so HI is -6V for this IC.) The measured data for the poly gate filters is tabulated in table 5.5. The most striking differences between the poly gate filters and the metal gate filters are the lower output offset voltages and lower crosstalk measured for the poly gate filters. The lower offsets are a

Measured Filter Characteristics (poly gate process)		
parameter	filter section A (zeros off)	filter section B
-3dB frequency	50Hz	50Hz
Output swing for HD_2=1%	0.7V	1V
Dynamic range	62dB	66dB
Adjacent channel crosstalk	-78dB	-80dB
Clock frequency (4 channels)	32kHz	3.2kHz
Sampling rate for each channel	8kHz	800Hz
Average DC output voltage of all channels (input grounded)		
$\Phi_A=\Phi_3$, $\Phi_B=\Phi_2$	-750mV	-130mV
$\Phi_A=HI$, $\Phi_B=\Phi_2$	-50mV	-45mV
$\Phi_A=\Phi_2$, $\Phi_B=HI$	180mV	40mV
$\Phi_A=\Phi_2$, $\Phi_B=\Phi_2$	200mV	55mV
Range of DC output offset voltages, all channels		
$\Phi_A=\Phi_3$, $\Phi_B=\Phi_2$	500mV	50mV
$\Phi_A=HI$, $\Phi_B=\Phi_2$	60mV	20mV
$\Phi_A=\Phi_2$, $\Phi_B=HI$	340mV	120mV
$\Phi_A=\Phi_2$, $\Phi_B=\Phi_2$	400mV	110mV

Table 5.5. Data for the multiplexed filters, poly gate process

direct consequence of the smaller gate area and overlap capacitances of the self-aligned poly gate transistor. The lower adjacent channel crosstalk is as expected, and the cause of the crosstalk in the metal gate filters is not clear. Possibly the crosstalk comes from coupling onto C_{fb} through the off transistors' overlap capacitances which is much larger in a metal gate process.

The harmonic distortion rapidly decreases as the output swing decreses. Typically, HD_2=1% for an output swing of 1V, HD_2=0.1% for an output swing of 0.25V, and HD_2 is buried in the noise for any output swing less than 0.1V. The harmonic distortion only varied a few dB as the different clocking schemes were tried. There was a larger variation in distortion from one chip to another for a given clocking scheme than there was for different clocking schemes on a given chip.

The frequency response for filter section A without the zeros is shown in figure 5.10. In figure 5.11, the frequency response of section A with a zero is shown. Note the difference in the frequency response when the switch phasing for C_1 is reversed. These responses agree with the transfer functions derived in chapter 3 (equations 3.2 and 3.3).

5.3.2. OP AMP

The op amp characteristics were measured and are included in the last column of table 5.4. The output swing of the op amp was limited to +1.25V, -3V due to the large body effect of the process.

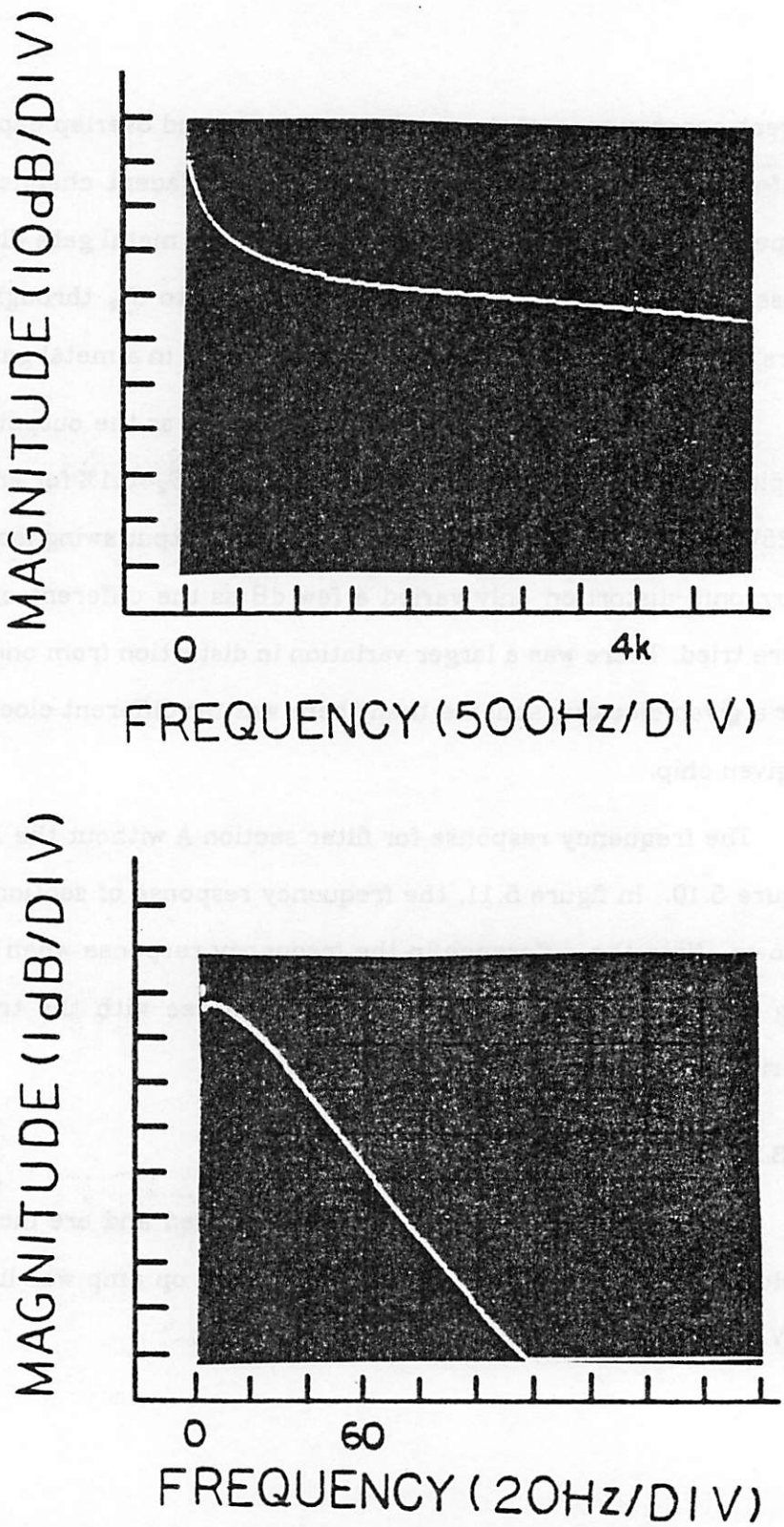


Figure 5.10 top: Frequency response of section A, channel 0
bottom: Frequency response of passband of section A, channel 0

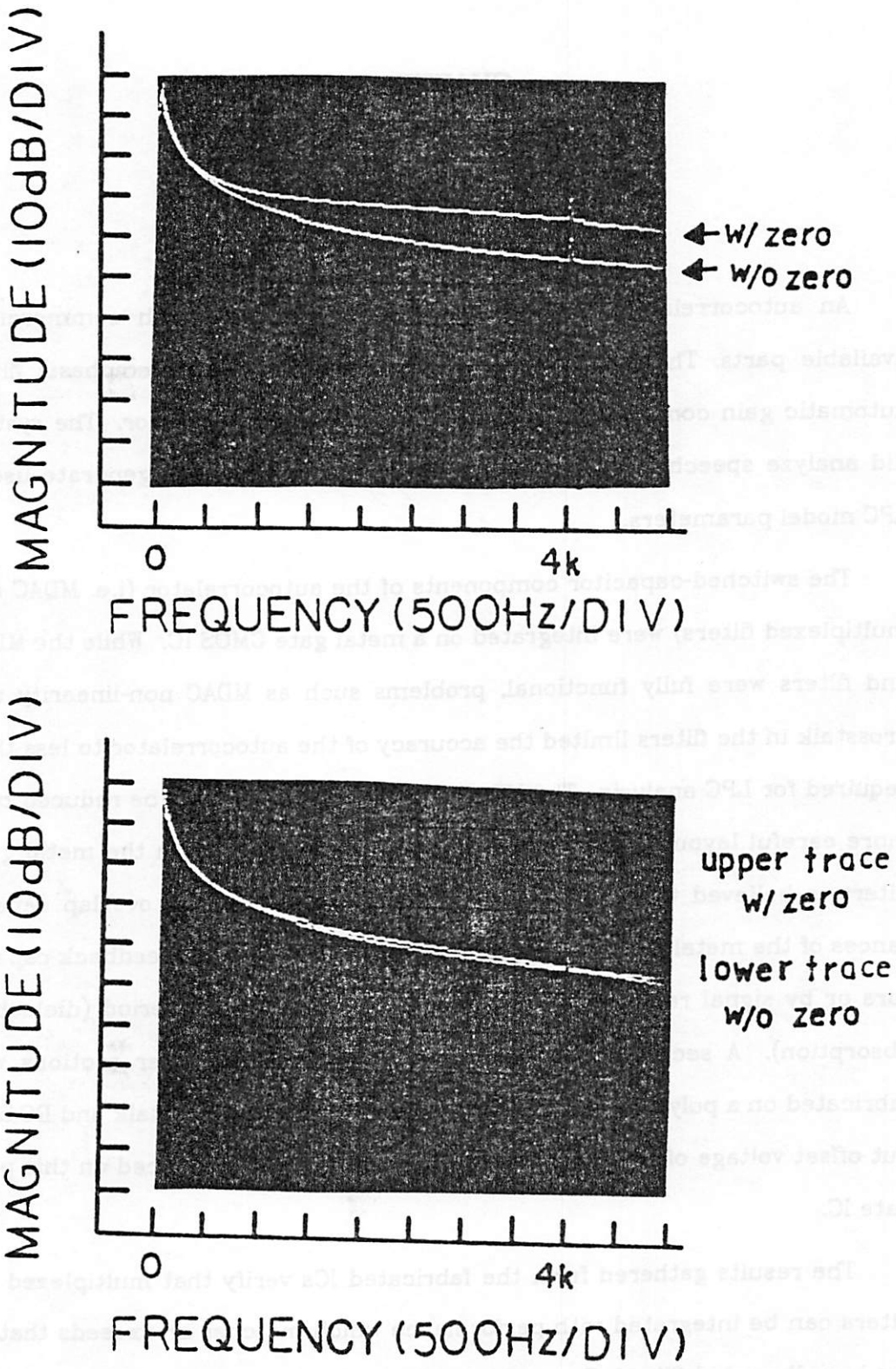


Figure 5.11. top: Frequency response of section A with zero, non-inverting phasing of C_1 (equation (3.2), $C_2=C_1$)
bottom: Frequency response of section A with zero, inverting phasing of C_1 (equation (3.3), $C_2=C_1$)

CHAPTER 6

SUMMARY AND CONCLUSIONS

An autocorrelation LPC analysis system was built with commercially available parts. The system consisted of an anti-alias/pre-emphasis filter, automatic gain control, SC autocorrelator, and microprocessor. The system did analyze speech in real time with sufficient accuracy to generate useful LPC model parameters.

The switched-capacitor components of the autocorrelator (i.e. MDAC and multiplexed filters) were integrated on a metal gate CMOS IC. While the MDAC and filters were fully functional, problems such as MDAC non-linearity and crosstalk in the filters limited the accuracy of the autocorrelator to less than required for LPC analysis. The MDAC non-linearity can easily be reduced by a more careful layout of the capacitor array. The crosstalk in the metal gate filters is believed to have been caused either by the large overlap capacitances of the metal gate transistors coupling signal onto the feedback capacitors or by signal retained on capacitor C_T after the reset period (dielectric absorption). A second IC consisting of two multiplexed filter sections was fabricated on a polysilicon gate CMOS process. Both the crosstalk and DC output offset voltage of the multiplexed filters were greatly reduced on this poly gate IC.

The results gathered from the fabricated ICs verify that multiplexed SC filters can be integrated with performance which matches or exceeds that of the breadboarded filters. By combining these filters with an accurate 8 bit SC MDAC, an autocorrelator with sufficient accuracy for speech analysis can be

integrated. (SC MDACs of 10 bit accuracy have been successfully integrated [32], so an 8 bit SC MDAC can be integrated.)

The complete autocorrelator in integrated form would include an 8 bit ADC, 8 bit MDAC, digital delay line (RAM), SC multiplexed filters, and control logic. With a modern polysilicon gate CMOS process with 3μm line widths, it is estimated that the entire autocorrelator would fit on a 10,000 mil^2 chip.

6.1.1. IMPROVEMENTS AND MODIFICATIONS

The system could be modified to give improved performance. The occassional unstable frames which are caused by course quantization of the autocorrelation values could be eliminated if the ADC which quantizes the autocorrelation values has an adjustable reference voltage which is set to R(0) or a voltage slighty greater than R(0) each frame. (This is possible because $|R(k)| \leq R(0)$ for all k.) This would make more effective use of the ADC's bits. But the DC offsets of the filters would have to be canceled prior to the ADC or the microprocessor would have to know the reference voltage each frame so it could properly subtract the offset voltages.

The op amp associated with the MDAC could be completely eliminated by making C_i of filter section 2 or 3 a binary weighted capacitor array and using the digital word $s(n-k)$ to select the capacitors and to determine the switch phasing.

If an LPC model with more poles is desired, the system is easily expanded to produce more autocorrelation values. The number of autocorrelation values which could be generated by the MDAC and multiplexed filters is limited by the settling time of the op amp.

The major drawback of the system is the need for multiple ADCs - one in the AGC, one to digitize the speech in the autocorrelator, and one to digitize the autocorrelation values for the microprocessor. Time sharing of these ADCs is possible since the autocorrelation values and AGC gain value are sampled at the frame rate which is much less than the sampling rate.

The AGC was very effective and could be more so if a quieter analog delay line were available. Other modifications to the AGC were considered and analyzed in [18].

It is possible that a simpler window could be used in the LPC autocorrelator. This would reduce the area required for the multiplexed filters. For instance, the zeros in the transfer functions have little effect and possibly could be eliminated altogether. This is worth investigating.

6.1.2. IC LAYOUT AND MODULAR DESIGN

The layout of the metal gate IC was a long, slow process. From start to finish, the layout took about six months. Learning how to use the layout program, poor graphics (a small black and white screen), and painful plotting (two tape transfers between three computers) contributed to the slow layout. Also, a great deal of time was spent experimenting with the layout to achieve the minimum total area. And, as problems came to light, circuit simulations were performed.

In contrast, the time required to lay out the poly gate IC was only three days! The layout was greatly simplified because building blocks (op amp, switches, and capacitors) had already been layed out by other students and were available. A new layout program with color graphics had been developed [33], and a plotter was available on the computer which was being used for layout. The total area of the chip was determined by the need for 40 bonding

pads, so there was no reason to worry about saving area during layout. The great reduction in layout time reflects an improvement in layout tools and the usefulness of pre-designed circuit building blocks.

6.1.3. ANALOG VS. DIGITAL PROCESSING

In most applications, the words "signal processing" are proceeded by the word "digital". Analog signal processing is dominant in high frequency applications where digital circuits are not fast enough to perform the required computations. But with switched-capacitor technology, all the basic building blocks necessary for analog signal processing are available (appendix B).

Digital processing has an advantage in that layout is not as critical for digital ICs as it is for analog ICs where parasitics must be taken into consideration. Many digital building blocks (or cells) have been developed, and they make for a quick and easy IC layout. But in digital processing, quantization, area, and power dissipation can be a problem.

Analog signal processing should be considered whenever signal processing tasks are to be performed. There are times when analog signal processing will result in a smaller, lower power integrated circuit and is worth the time and effort to design.

APPENDIX A

FINDING THE FILTER TRANSFER FUNCTION

A derivation of the filter transfer functions $H'_k(z)$ (equation 2.11) is included here for completeness. We begin by recalling the time window suggested by Barnwell [5]

$$w(n) = (1-n)\alpha^{-n} \quad n \leq 0, \; 0<\alpha<1$$

and

$$h(n) = w(-n) = (n+1)\alpha^{n} \quad n \geq 0, \; 0<\alpha<1,$$

which has the z-transform

$$H(z) = \frac{1}{(1-\alpha z^{-1})^2}, \quad |z|>\alpha. \tag{A.1}$$

The z-transform of the sequence $h'_k(n)=h(n)h(n+k)$ is desired. The z-transform of $h(n+k)$ is

$$h(n+k) \rightarrow z^k H(z) = \frac{z^k}{(1-\alpha z^{-1})^2}, \quad |z|>\alpha.$$

$H'_k(z)$ is found by integrating, equation (2.8),

$$H'_k(z) = H_k(z) * z^k H_k(z) = \frac{1}{2\pi j}\int v^k H(v) H(\frac{z}{v}) v^{-1} dv = \frac{1}{2\pi j}\int H(v) H(\frac{z}{v}) v^{k-1} dv$$

where the integration is along any closed path in the region of convergence of both $H(v)$ and $H(\frac{z}{v})$.

Substitution of (A.1) into (2.8) gives

$$H'_k(z) = \frac{1}{2\pi j}\int \frac{1}{(1-\alpha v^{-1})^2} \frac{1}{(1-\alpha \frac{v}{z})^2} v^{k-1} dv = \frac{1}{2\pi j}\int \frac{1}{(v-\alpha)^2} \frac{1}{(1-\alpha \frac{v}{z})^2} v^{k+1} dv.$$

We already know that $H(v)$ converges for all $|v|>\alpha$. So $H(\frac{z}{v})$ will converge if $\left|\frac{z}{v}\right|>\alpha$, or equivalently, $|v|<\frac{|z|}{\alpha}$. The common region of convergence is

$\alpha<|v|<\frac{|z|}{\alpha}$. Any closed path in the common region will encircle the pole of order two at $v=\alpha$; the poles at $v=\frac{z}{\alpha}$ are outside any such path. Note that since we are interested in values of $k\geq 0$, v^{k+1} is analytic everywhere and does not contribute a pole. The integral can therefore be evaluated using the residue theorem which states that an integral around a closed contour equals $2\pi j$ times the sum of the residues of the integrand evaluated at all poles encircled by the path of integration [34].

Since $H(v)$ has a pole of order two at $v=\alpha$, evaluation of the residue of the integrand is found by differentiating

$$residue = \left.\frac{\partial[(v-\alpha)^2 H(v) H(\frac{z}{v}) v^{k+1}]}{\partial v}\right|_{v=\alpha} = \frac{(k+1)\alpha^k + (1-k)\alpha^{k+2} z^{-1}}{(1-\alpha^2 z^{-1})^3}.$$

This residue equals $H'_k(z)$ since the $2\pi j$ which multiplies the integral in (2.8) cancels the $2\pi j$ of the residue theorem.

It is worth noting that if $h(n)=w(-n)$ is chosen to be the impulse response of a one pole filter, the resulting output filters $H'_k(z)$ are also one pole filters. That is, if $h(n)=w(-n)=\alpha^n$ for $n\geq 0$ so that

$$H(z) = \frac{1}{1-\alpha z^{-1}},$$

then

$$H'_k(z) = \frac{\alpha^k}{1-\alpha z^{-1}}.$$

The question might arise, "If a one pole window corresponds to a one pole output filter, and if a two pole window corresponds to a three pole output filter, then what type of window corresponds to a two pole output filter?" If we take for $H'_0(z)$ the two pole filter

$$H'_0(z) = \frac{1}{(1-\alpha z^{-1})^2},$$

which has the corresponding impulse response

$$h'_0(n) = (n+1)\alpha^n,$$

and recall that the output filter for $k=0$ has impulse response $h'_0(n)=h^2(n)=w^2(-n)$, then the window can be found from

$$h(n) = \sqrt{h'_0(n)} = \sqrt{n+1}\alpha^{\frac{n}{2}}.$$

Such a window has no discontinuities and more heavily weights the present frame than the past frames as desired, but computation of $H'_k(z)$ for $k\neq 0$ is impossible because $h(n)$ is irrational in n and therefore does not have a closed form $H(z)$ representation to use in equation (2.8). $H'_k(z)$ is the z-transform of $h'_k(n)=h(n)h(n+k)$ which can be written

$$h'_k(n) = h(n)h(n+k) = \sqrt{(n+1)(n+k+1)}\alpha^{\left(n+\frac{k}{2}\right)}.$$

The inability to find $H'_k(z)$ as a rational polynomial which can be realized as a recursive sampled-data filter eliminates this window from consideration.

APPENDIX B

CATALOG OF ANALOG COMPUTATIONAL CIRCUITS

The autocorrelator employs the basic computational building blocks for signal processing - delay, multiplication, and filtering. There are many ways to implement each of these blocks, and some alternatives are mentioned briefly below. For a more detailed discussion of each circuit, please see the references cited.

1.1.1. DELAY LINES

Digital delay lines are basically RAM and all are functionally the same. A number of different RAM cells are available for constructing a digital delay line [35]. Analog delay lines are of two types - cascaded sample-and-hold circuits or multiplexed sample-and-hold circuits.

The simplest MOS sample-and-hold circuits suffer from offset and signal dependent errors (i.e. distortion) due to overlap capacitance and channel charge redistribution. An interesting, feedback-corrected sample-and-hold was presented recently [36]. Dynamic range of 80dB was reported for this sample-and-hold which was constructed with a polysilicon gate process.

A multiplexing sample-and-hold (S/H) scheme, employing one op amp and p capacitors, was used in an LPC lattice analyzer [14]. The delayed signals are stored on different capacitors, and when a particular delayed signal is desired, the corresponding capacitor is connected into the circuit. The accuracy of this multiplexed S/H was limited by the fixed-pattern noise associated with the switching. A dynamic range of only 45dB was reported, but the

circuit was constructed with a metal gate process; so direct comparison with the 80dB figure above is not possible.

1.1.2. ANALOG MULTIPLIER

A four-quadrant analog MOS multiplier has been fabricated by Soo [37]. This multiplier provided 77dB of signal-to-noise ratio, 1.5MHz bandwidth, and 0.3% non-linearity while occupying 450mil^2 of chip area. Such a circuit, if combined with a high quality analog SC delay line, would work very well in an autocorrelator.

A four-quadrant analog multiplier/divider in bipolar technology was reported by Gilbert [38]. The non-linearity was an excellent 0.01% with a ±10 volt input signal swing. This circuit might be used with SC delay line and filters in a bipolar compatible CMOS process. The delay lines (cascaded S/H circuits) and filters for the autocorrelator in a purely bipolar technology would be prohibitively large.

1.1.3. MDACs

An MDAC is just a DAC with its analog reference input connected to an analog voltage equal to the multiplicand. No attempt will be made to discuss all the possible resistive DACs, capacitive DACs, and combinations thereof which might function as an MDAC; see references [39] and [30] for such a discussion. The MDAC chosen for the autocorrelator is a capacitive DAC compatible with a standard CMOS process.

1.1.4. FILTERS

Various implementations of a real pole and real zero are available. The low frequency cutoff of 25Hz eliminates RC filters from considerations due to the large area required. Some clever implementations of continuous time

filters with low cutoff frequencies using FETs have been reported [40],[41]. SC filters allow low cutoff frequencies and have become very popular for audio range filtering [39]. Implementation of a negative real zero (in the z-domain, not the s-domain) poses a problem in some SC configurations.

1.1.5. ABSOLUTE VALUE CIRCUITS

An MOS absolute value circuit can be constructed using a diode-connected enhancement mode transistor to restrict current flow to one direction only, figure 4.13. Accurately matched resistors are required in such a circuit. Alternatively, a sampled-data SC absolute value circuit can be used to compute the absolute value of the input. This requires a comparator to determine the sign of the present sample and an amplifier with a programmable gain of ±1. In a SC amplifier, the sign of the gain is easily inverted by changing the switch phasing of the input sampling switch.

1.1.6. LONG DELAY LINES

Long delay lines, with lengths on the order of a hundred samples, are not feasibly realized as a cascade of sample-and-hold circuits because the number of op amps becomes large, and the area required for them becomes prohibitive. The multiplexed S/H mentioned above might work for large delays. But the natural choice for a long analog delay line is the charge-coupled device or the bucket-brigade device. They are compatible with MOS circuitry. A digital approach would employ a large RAM.

APPENDIX C

SPEECH PROGRAMS AND SENTENCES

Computer simulation results presented in chapter 4 were performed by passing digitized speech through custom written programs. The speech was sampled at 8kHz and digitized by a 12 bit ADC. All programs were written in the C programming language. When required, synthesized speech was played back by a 12 bit DAC operating at an 8kHz rate. The sentences used for the simulations are listed in table C.1 , followed by the source code for the programs.

2.1. GENERATEAUTO.C

The program Generateauto.c simulates the autocorrelation system and allows optional inclusion of the various system modifications - downsampling, AGC, multiplier quantization (MDAC), pre-emphasis, and quantization of the autocorrelation values. It takes speech as input and generates the autocorrelation values for each frame. The source for Generateauto.c is listed on the following pages.

Digitized Speech Used for Simulations		
File Name	Sentence or Phrase Spoken	Speaker
Adam.t	Adam.	George White
Baker.t	Baker.	George White
Charlie.t	Charlie.	George White
David.t	David.	George White
Dr.Bob.t	Doctor Bob is on vacation again.	Paul Hurst
Hello.t	Hello, how are you?	Paul Hurst
Susan.t	Susan kicked the goat on Sunday.	Paul Hurst
Suzie.t	Suzie sat on the sandwich.	Paul Hurst
Thieves.t	Thieves who rob friends deserve jail.	Peter Chu
We.t	We were away a year ago.	Bob Brodersen
Why.t	Why do I owe you a letter?	Peter Chu
ah.t	/ah/	synthetic sound
ahsh.t	/ah/,/sh/	synthetic sounds
audiocritic.t	We publish part one of the transcript of our all day seminar on the state of the art.	Bob Brodersen
clock.t	Clock,slope,field.	John Fattaruso (RSB)
cursor.t	Move the cursor to the clock slope field.	John Fattaruso
demo.t	This is a demonstration of synthetic speech.	Bob Brodersen
ee.t	/ee/	synthetic sound
eh.t	/eh/	synthetic sound
handel.t	Behold, I tell you a mystery.	John Fattaruso
is.t	Is waiting to see you.	Steve Love
mike.t	The two powering modules available are fitted with a battery and a balancing transformer.	Steve Love
oak.asa.t	The oak trees are strong.	Asa Romberger
oak.ellen.t	The oak trees are strong.	Ellen Szeto
poweramp.t	The two most interesting power amplifiers for the audio purist.	Barry Hyman
room_noise.t	(background room noise)	small fan running
sh.t	/sh/	synthetic sound
shah.t	/sh/,/ah/	synthetic sounds
ss.t	/ss/	synthetic sound
trends.t	Trends and perspectives in signal processing.	Steve Love
xyz.t	X Y Z.	Paul Hurst

Table C.1. Speech used in the simulations of chapter 4

GENERATEAUTO.c

```
#define SOURCE "/brodersen/hurst/bin"
#ifdef M
#undef SOURCE
#define SOURCE "/mb/audio/hurst/bin"
#endif
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

/* This program generates the first 10 autocorrelation values
 * for the input file. The autocorrelation values are stored in
 * files. 'output1.xx' contains the autocorrelatin values determined
 * using Barnwell's 10 filters exactly. 'output2.xx' contains the
 * autocorrelation values determined using a lower sampling rate
 * on the last section of each filter.
 *
 *      to compile:  cc  -O Generateauto.c -lm -lNS
 *
 * This is the VAX version.
 */

#define AD_BITS 11 /* number of bits in A/D converter (excluding sign bit) */
#define AD_full_scale 2047
#define POINTS 400 /* sets maximum frame size (in samples) */
#define ALPHA 0.98
#define pi 3.1415927
#define NO_AUTOC 10
#define MAX_FRAMES 500 /* limits maximum number of frames to 500 */
#define sign(x) (((x) >= 0) ? 1 : -1)  /* nice line of code, huh? */
                                       /* see 'C manual', pg. 47  */

/* all external variables are initialized to 0 */

int frame_rate       =80;
int samp_freq =8000;
int contsampflag =0;
int dwnsampflag  =0;
int gainflag     =0;
int preempflag   =1;
int agcflag      =0;
int clipflag     =1;
int quant_flag         =0;
int q_bits       =0; /* This value is never used. If quant_flag==1,q_bits is
                      * read from command line.
                      */
int r_full_scale=2047;
int rfs_bits     =11; /* rfs_bits is 'r_full_scale_bits'. # bits which correspond
                      * to r_full_scale.
                      */
float R[MAX_FRAMES][NO_AUTOC];
```

```
struct stat stbuf;
FILE *fopen(),*fp_scaled,*fp_sec1,*fp_sec2,*fp_sec3,*fp_quant;

main (argc,argv)
int argc;
char *argv[];

{       int i,l,rate,row,frames,atoi(),framelength,agc[MAX_FRAMES],power();
        long filesize;
        float cos(),pow(),low_pass(),mult();
        float max_R0,s[POINTS],prod[POINTS];
        char *temp_file,*mktemp();
        FILE *fopen(),*fp_input;


        max_R0=0.0;

        if (argc == 1) instructions();

        i=2;
        while(++i <= argc)
        {       if (argv[i-1][0] != '-')
                        ex ("bad parameter");
                else switch(argv[i-1][1])
                {       case 'c': contsampflag=1;
                                  break;
                        case 'd': dwnsampflag=1;
                                  rate=atoi(argv[i++]);
                                  if(rate == 0) ex("-d ?");
                                  break;
                        case 'r': frame_rate=atoi(argv[i++]);
                                  break;
                        case 'g': gainflag=1;
                                  break;
                        case 'b': rfs_bits=atoi(argv[i++]);
                                  r_full_scale=power(2,rfs_bits) - 1;
                                  gainflag=1;
                                  printf("gainflag is now on (i.e. -g flag assumed)\n");
                                  if((rfs_bits > 15) || (rfs_bits < 5))
                                        ex("bad number of bits");
                                  break;
                        case 'a': agcflag=1;
                                  break;
                        case 'k': clipflag=0;
                                  break;
                        case 'q': quant_flag=1;
                                  q_bits=atoi(argv[i++]);
                                  break;
                        case 'p': preempflag=0;
                                  break;
                        case 's': samp_freq=atoi(argv[i++]);
                                  break;
                        default:  ex("bad parameter");
```

```
        }
}

if((agcflag==1)&&(gainflag==1))
        printf("Warning: Using both the -g and -a options \n\twill give
        unpredictable synthetic speech due to incorrect agc gain. \n");
if ((contsampflag==0)&&(dwnsampflag==0))
{       printf("Should have specified -c or -d flag: '-c' assumed\n");
        contsampflag = 1;
}
if ((contsampflag==1)&&(dwnsampflag==1))
        ex("can't use both -c and -d flags");
if ((quant_flag == 1) && (q_bits < 0))
        ex("can't have q_bits < 0. Try again.");

if(agcflag == 1)
        if((fp_scaled=fopen("scaled","w")) == NULL)
                ex("cannot create file 'scaled'");

if(quant_flag == 1)
        if((fp_quant=fopen("quantized","w")) == NULL)
                ex("cannot create file 'quantized'");

if((fp_sec1=fopen("out_section1","w")) == NULL)
        ex("cannot create file 'out_section1'");
if((fp_sec2=fopen("out_section2","w")) == NULL)
        ex("cannot create file 'out_section2'");
if((fp_sec3=fopen("out_section3","w")) == NULL)
        ex("cannot create file 'out_section3'");

/* temp_file = mktemp("PrEeMpXXXXX"); */

if (preempflag == 1)
{       printf("PRE-EMPHASIZING INPUT\n");
        /* sys_call("%s/preemp %s>%s",SOURCE,argv[1],temp_file); */
}
/* else         sys_call("cp %s %s",argv[1],temp_file); */

if ((fp_input = fopen(argv[1],"r")) == NULL)
        ex("can't open input file \n");

stat(argv[1],&stbuf);           /*compute length of file fp_input*/
filesize = stbuf.st_size/2;
printf("file size is %ld words\n",filesize);

framelength = samp_freq/frame_rate;
printf("frame length=%d samples          frame rate=%d Hz\n",
framelength,frame_rate);
frames = filesize/framelength;
printf("there are %d frames \n",frames);
if(frames > MAX_FRAMES)
{       printf("Program can handle only %d frames. Sorry.",MAX_FRAMES);
        exit();
```

```
        }

        /* s[i] will hold the speech samples */
        for (i=0;i<POINTS;i++)
                s[i] = 0.0;
        for(i=0;i<POINTS;i++)
                prod[i] = 0.0;

        for (row=0;row<frames;row++)
        {
                printf("\nframe %4d          ",row+1);
                for (i=0;i<NO_AUTOC;i++)
                        s[i] = s[i+framelength];
                for (i=NO_AUTOC;i<framelength+NO_AUTOC;i++)
                        s[i] = getsh(fp_input);

                if (preempflag == 1) preemp(s,framelength);
                if (agcflag==1) do_agc(s,agc,framelength,row);

                for (l=0;l<NO_AUTOC;l++)
                {
                        for(i=0;i<framelength+1;i++)
                                prod[i]=mult(s[i+NO_AUTOC-1],s[i-l+NO_AUTOC-1],
                                q_bits,l)/AD_full_scale;

                        if (contsampflag == 1)
                                R[row][l]=low_pass(prod,l,1,framelength);
                        if (dwnsampflag == 1)
                                R[row][l]=low_pass(prod,l,rate,framelength);
                }
        }

        for (i=0;i<frames;i++)
                if (R[i][0] > max_R0) max_R0=R[i][0];

        if (contsampflag == 1)
                norm_store(max_R0,frames,1,agc);

        if (dwnsampflag == 1)
                norm_store(max_R0,frames,2,agc);

        fclose(fp_input);
        fclose(fp_sec1);
        fclose(fp_sec2);
        fclose(fp_sec3);
        /* sys_call("/bin/rm %s",temp_file); */
        if(agcflag == 1) fclose(fp_scaled);
        if(quant_flag == 1) fclose(fp_quant);
        printf("\n");
}

/* low_pass() is the bank of low pass filters required in the autocorrelation
 * computation. Each autocorrelation coefficient has a different filter.
```

```
 */

#define fc 51.453     /*51.453 is the -3db point for filter #3 when fs=8kHz*/
#define p1 0.9604
#define p2 0.9604
#define gain1 2.5      /*want filter section 1 to have dc gain = 2.5       */
#define gain2 1.38     /*gain of filter section 1 (due to b0 & b1) is 1.9604*/
#define gain3 1.3     /*so gain1*1.9604 = 1.27*1.9604=2.5                 */

#define GAIN0 1.9604        /*This is the dc gain for filter section 0 for R[0]  */

float low_pass(prod,l,rate,framelength)
int l,rate,framelength;
float prod[POINTS];
{       float input2,input3;
        float pow(),cos(),sqrt(),x,b0,b1,p3,g1,g2,g3,z1,d0;
        int i,section;
  /* hopefully, all static variables are initialized to zero (seems to be true) */
static float output1[2][NO_AUTOC],output2[2][NO_AUTOC],output3[2][NO_AUTOC];

        if (rate == 1) section = 0;
        else           section = 1;

        b0=(l+1)*pow(ALPHA,l+0.0);
        b1=(l-1)*pow(ALPHA,l+2.0);
        z1=b1/b0;
        d0=b0/GAIN0;    /* what the heck is d0 ? */

        x=cos(2.0*pi*fc/(8000/rate));     /* compute p3 (it depends on the */
        p3=(2-x) - sqrt((2-x)*(2-x)-1);   /* down sampling of last section.
                                          * 8000 is sampling rate which is
                                          * assumed in window parameters
                                          */

        g1=gain1*(1-p1)/GAIN0;
        g2=gain2*(1-p2);
        g3=gain3*(1-p3);

        for(i=1;i<framelength+1;i++)
        {       input2=output1[section][l];
                input3=output2[section][l];
                output1[section][l]=p1*output1[section][l]+
                (b0*prod[i]-b1*prod[i-1])*g1;
                if (l == 0) putsh((short)output1[section][l],fp_sec1);
                output2[section][l]=p2*output2[section][l]+input2*g2;
                if (l == 0) putsh((short)output2[section][l],fp_sec2);
                if(i==(i/rate)*rate)
                {       output3[section][l]=p3*output3[section][l]+input3*g3;
                        if (l == 0) putsh((short)output3[section][l],fp_sec3);
                }
        }

        return(output3[section][l]);
```

```
}

/* norm_store() scales autocorrelation results (if asked) and stores
 * the results in a file.
 */

norm_store(max,rows,outflag,agc)
float max;
int rows,outflag,agc[MAX_FRAMES];
{       int i,j,norm,scale,quantize(),round_off();
        float fscale,sqrt();
        char  *filename;
        FILE *fopen(),*fp_output;

        if (outflag == 1)
        {
                if (gainflag == 1)      filename = "output1.g";
                else if (agcflag == 1)  filename = "output1.agc";
                   else                 filename = "output1.ng";
        }
        else
        {
                if (gainflag == 1)      filename = "output2.g";
                else if (agcflag == 1)          filename = "output2.agc";
                   else                 filename = "output2.ng";
        }

        if ((fp_output = fopen(filename,"w")) == NULL)
                ex("can't open output file");

        if (max < 0.01) max = 0.1; /* in case max is zero */
        for (i=0;i<rows;i++)
        {       if (gainflag == 1)
                        if (R[i][0] < 0.01) fscale = 1;
                        else fscale = max / R[i][0];

                for(j=0;j<NO_AUTOC;j++)
                {       if (gainflag == 1)
                        {       norm = (r_full_scale*(R[i][j]/max)*fscale);
                                /* assure R[0] = r_full_scale */
                                if(j == 0) putsh(r_full_scale,fp_output);
                                else       putsh(norm,fp_output);
                        }
                        else            /*if agcflag or no gain options*/
                                putsh(limit(R[i][j],r_full_scale),fp_output);

                }
                if (gainflag == 1)
                {       scale=limit(sqrt((r_full_scale/255.0)*fscale),32767);
                        putsh(scale,fp_output);
                }

                else if (agcflag == 1)
```

```
                        putsh(agc[i],fp_output);

                    else putsh(1,fp_output); /* output a 1 for the gain
                                              * if 'no gain' option
                                              */
        }

        fclose(fp_output);
        return;
}

#define ZERO 0.7

preemp(s,framelength)
float s[];
int framelength;
{
        int i;
        static int fopenflag = 0;
        static float x0, x1, output;
        static FILE *fp_precheck;

        /*
        if(fopenflag++ == 0)
                if((fp_precheck = fopen("PREcheck","w")) == NULL)
                        ex("cannot create PREcheck");
        */

        for(i = NO_AUTOC; i < framelength + NO_AUTOC; i++)
        {
                x0 = s[i];
                s[i] = (x0 - ZERO*x1)/(1. + ZERO); /* DC gain = 1 */
                /* putsh((short)s[i],fp_precheck); */
                x1 = x0;
        }
}

#define ROOT2 1.414
#define A    0.99

do_agc(s,agc,framelength,row)
float s[POINTS];
int agc[MAX_FRAMES],framelength,row;
{       int j,limit();
        float flimit(),fabs(),sqrt(),sigma;
        static FILE *fp_agcout;
        static float out1,out2;  /* out1 and out2 are initialized to 0.0
                                  * first time 'do_agc()' is called.
                                  */
        if (row == 0)
                if ((fp_agcout = fopen("agc.out","w")) == NULL)
                        ex("cannot open 'agc.out' file");
```

```
        for(j=NO_AUTOC;j<framelength+NO_AUTOC;j++)
        {       out1 = ALPHA*out1 + (1.0-ALPHA)*fabs(s[j]);
                out2 = ALPHA*out2 + (1.0-ALPHA)*out1;
        }

/* The output of this |(.)| and LPF circuit is an estimate of the rms value
 * of the latest speech frame. (Actually, we must multiply the result by
 * ROOT2 to get an estimate of sigma.)
 * LPF should have the same impulse response as the window used in
 * autocorrelaton computation (impulse response of t.f. w/ two equal poles).
 */

        sigma = ROOT2 * out2;
        agc[row] = limit(4 * sigma,AD_full_scale);
        if (agc[row] == 0) agc[row] = 1; /* can't allow 0 as an agc value.
                                          * agc = 0 screws up synthesizer */
        putsh(agc[row],fp_agcout);
        printf("agc output is %d",agc[row]);

        for(j=NO_AUTOC;j<framelength+NO_AUTOC;j++)
        {       s[j] = (s[j] * AD_full_scale)/agc[row];
                if (clipflag == 1) s[j] = flimit(s[j],(float)AD_full_scale);
                putsh(limit(s[j],AD_full_scale),fp_scaled);
        }
        return;
}


instructions()
{       printf("usage: generateauto file {-c} {-d dwnsamp} {-r frame_rate}
{-s samp_rate}  \n\t\t{-g} {-a} {-k} {-q q_bits} {-p} {-b bits}\n\n");
        printf("\t file must contain at least 500 samples of speech \n");
        printf("\t -c: requests constant sampling rate to be used\n\t\t
        (output is in output1)\n");
        printf("\t -d dwnsamp: sets down sampling rate for the last filter
        section \n");
        printf("\t\t(dwnsamp=2 means the sampling rate drops by a factor
        of 2)\n");
        printf("\t\t(output is in output2)\n");
        printf("\t -r frame_rate: sets the rate at which the autocorrelation
        values \n\t\tare to be output (in Hz). Default is 80 Hz\n");
        printf("\t -s samp_rate: sets the sampling rate (Default is 8k Hz)\n");
        printf("\t -g: causes program to output a scale factor after each
        set\n\t\tof autocorrelation values (R[0]=full_scale)\n");
        printf("\t -a: causes an agc circuit to pre-process the signal\n\t\t
        Value of gain is output every frame\n");
        printf("\t\tagc scaled time waveform is stored in file 'scaled'\n");
        printf("\t -k: stops clipping from occuring at output of agc ckt.\n");
        printf("\t -q q_bits: causes multiplication to be done with multiplier
        \n\t\tquantized to q_bits (see comment below; keep q_bits <= 11)\n");
        printf("\t -p: disables pre-emphasis \n");
        printf("\t -b bits: sets R[0] = 2**bits when used with -g option\n\t
        \tProgram accepts 8 <= bits <= 15 \n\t
```

```
        \t(default is bits=11 -> full_scale=2047)\n");
        printf("\t Comment: 'bits' is the number bits used to store the\n\t
        \t results NOT INCLUDING the sign bit.\n");
        printf("\t Therefore, bits=8 allows numbers to range \n\t
        \tbetween -255 and +255\n");
        exit();
}

/* limit(x,max) takes inputs x (float) and max (int) and returns an integer
 * whose value is between -max and +max inclusive. This is a 'clipper' for x.
 */

limit(x,max_value)
float x;
int max_value;
{       int output;
        float fabs();

        if (fabs(x) > max_value)
        {       /*printf("\nlimiting\t");*/
                return(output = sign(x) * max_value);
        }

        else return(output = x);
}

/* flimit(x,max) takes inputs x (float) and max (float) and returns an float
 * whose value is between -max and +max inclusive. This is a 'clipper' for x.
 */

float flimit(x,max_value)
float x,max_value;
{
        float fabs();

        if (fabs(x) > max_value)
        {       /*printf("\nlimiting\t");*/
                return(sign(x) * max_value);
        }

        else return(x);
}

/* mult(x,y,bits) does floating point multiplication. If asked, the function
 * will quantize y and return the product.
 */

float mult(x,y,bits,l)
float x,y;
int bits,l;
{       float pow(),y_quantized;
        int quantize(),power();
```

```
        if(quant_flag == 0) return(x * y);

        else
        {       y_quantized = quantize(y,bits);
                /* printf("y=%f\ty_quantized=%f\n",y,y_quantized); */
                if (l == 0) putsh((int)y_quantized,fp_quant);/* output quantized
                                                        * speech once only*/
                return(x * y_quantized);
        }
}

/* quantize(y,b) takes y (float); clips it to AD_full_scale; and then
 * quantizes its value to b bits (not including sign). Returns an int.
 * Quantization simulates an A/D converter (see t.f., pg. 106 of Analog
 * Devices 'Analog to Digital Conversion Handbook')
 */

quantize(y,b)
int b;
float y;
{       float y_temp,y_norm,flimit();
        int limit();

        y_temp = flimit(y,(float)AD_full_scale);        /* clip y */
        b = b + 1;
        y_temp = ((((int) (y_temp+(1<<(AD_BITS-b)))) >> ((AD_BITS+1)-b))
        << (AD_BITS+1)-b);          /* quantize to b bits */

        return(limit(y_temp,AD_full_scale));                    /* clip y_temp */
}

/* round_off(x) takes x (float) and returns the integer closest to x
 */

round_off(x)
float x;
{
        return((int) (x + sign(x)*0.5));
}

Abs(t)
int t;
{
        return(sign(t) * t);
}

/* power(a,b) is an integer version of pow(). Returns a to the b power.
 * a,b, and result are all integers.
 */

power(a,b)
int a,b;
{       int result = 1;
```

```
        while(b > 0)
        {       result = a * result;
                b--;
        }

        return(result);
}

/* ex() prints a message, then exits. */

ex(str)
char *str;
{       printf("%s\n",str);
        exit();
}

/* sys_call(string,arg) does the following:
 *
 * char command[];
 * sprintf(command,string,arg1,arg2,..);
 * system(command);
 *
 * Only %s, %c, %d and %h are allowed in the string. %h is replaced
 * with the user's home directory path. (i.e. HOME of 'printenv')
 */

#define MAX_LINE 500
#define NULL_CHAR '\0'
#define BLANK ' '
#define RETURN '\n'

sys_call(string,arg)
char string[MAX_LINE],*arg;
{       char command[MAX_LINE],*getenv(),**ptr;
        int i = 0;

        ptr = &arg;
        sprintf(command,"%c",BLANK);

        while( string[i] != NULL_CHAR )
                if( string[i] == '%' )
                {       switch(string[++i])
                        {       case 's':
                                        sprintf(command,"%s%s",command,*(ptr++));
                                        break;
                                case 'h':
                                        sprintf(command,"%s%s",command,
                                        getenv("HOME"));
                                        break;
                                case '%':
                                        sprintf(command,"%s%c",command,string[i]);
                                        break;
                                case 'd':
```

```
                                sprintf(command,"%s%d",command,*(ptr++));
                                break;
                        default:
                                sprintf(command,"unknown conversion type
                                '%%%c'",string[i]);
                                ex(command);
                                break;
                        }
                        i++;
                }
                else sprintf(command,"%s%c",command,string[i++]);

        system(command);
        return;
}
```

2.2. SPEAKPITCH.C

To synthesize speech which was analyzed by Generateauto.c, a simple direct-form LPC analyzer was written. All computations are floating point.

SPEAKPITCH.c

```
#define SOURCE "/brodersen/hurst/bin"
#define PITCH_FILE "/brodersen/hurst/source/8msec_pitch"
#ifdef M
#undef SOURCE
#define SOURCE "/mb/audio/hurst/bin"
#undef PITCH_FILE
#define PITCH_FILE "/mb/audio/hurst/source/8msec_pitch"
#endif
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#define NO_AUTOC 10
#define NO_POLES 9
#define LOTS 2000
#define UV 0  /* indicator variable-UV means unvoiced frame */
#define V 1  /* indicator variable- V means voiced frame  */
#define MAX_24BITS 8388607 /* max value for a 24 bit integer:(2^23)-1 */

int frame_rate =80;
int samp_freq=8000;
int floatflag  =1; /* float version of Durbin's Recursion is default */
int pitchflag  =0;
int gainflag   =0;
```

```
int agcflag    =0;
int unstableflag=1;
int deempflag  =1;
int unstble_frames=0;
char *pitch_file;
struct stat stbuf;

/* This program is a refinement of Sammy Lum's Speak.c. It runs much faster.
 *
 * Autocorrelation data and pitch estimates are fed into this program to
 * produce synthetic speech. The output speech is stored in a file.
 *
 *  To compile: cc -O Speakpitch.c -lm -lNS  (VAX program)
 */

main(argc,argv)

int argc;
char **argv;
{
        int i,atoi();
        char *temp_file,*mktemp(),string[200];

        pitch_file = PITCH_FILE;

      if (argc == 1) instructions();
        if ((argv[1][0] == '-') || (argv[2][0] == '-'))
                ex("bad file name");

                i=3;
                while (++i <= argc)
                {       if (argv[i-1][0] != '-')
                                ex("bad parameter");
                        else switch(argv[i-1][1])
                        {       case 'r':frame_rate = atoi(argv[i++]);
                                        break;
                                case 'f':floatflag = 1;
                                        break;
                                case 'p':pitch_file = argv[i++];
                                        break;
                                case 'g':gainflag = 1;
                                        break;
                                case 'a':agcflag = 1;
                                        break;
                                case 's':samp_freq = atoi(argv[i++]);
                                        break;
                                case 'u':unstableflag = 0;
                                        break;
                                case 'd':deempflag = 0;
                                        break;
                                default: ex("bad parameter\n");
                        }
                }
```

```
                if ((gainflag == 1)&&(agcflag == 1))
                        ex("can't use both -g and -a options");
        if (filter(argv) < 0) ex(" ");

                if(deempflag == 1)
                {
                        printf("DE-EMPHASIZING\n");
                        sys_call("%s/deemp %s > %s",SOURCE,argv[2],
                        temp_file=mktemp("Spk_TemPXXXXXX"));
                        printf("SCALING\n");
                        sys_call("%s/scale %s %s;/bin/rm %s",SOURCE,
                        temp_file,argv[2],temp_file);
                }

                if(deempflag == 0)
                {
                        printf("SCALING\n");
                        sys_call("%s/scale %s %s",SOURCE,argv[2],
                        temp_file=mktemp("Spk_TemPXXXXXX"));
                        sys_call("/bin/mv -f %s %s",temp_file,argv[2]);
                }
}

instructions()
{       printf("usage: speakpitch file1 file2 {-p file3} {-r auto_rate}
          {-s samp_freq} {-f} \n\t\t{-g} {-a} {-u} {-d}\n\n");
        printf("file1 should contain the autocorrelation values\n");
        printf("file2 will contain the output samples\n");
        printf("-p file3 should contain pitch information\n
          \tdefault is a constant 8msec pitch period\n");
          printf("-r auto_rate is the frequency at which the autocorrelation
          \n\tvalues were sampled (in Hz);default is 80Hz\n");
          printf("-s samp_freq is the sampling frequency (default is 8kHz)\n");
          printf("-f flag causes Durbin's Recursion to be done in floating point\n");
          printf("-g causes program to read gain input from autocorrelation file\n");
          printf("-a causes program to read agc gain from autocorrelation file\n");
          printf("-u causes program to terminate Durbin's recursion whenever\n
          \t\ta reflection coeff. exceeds ONE\n");
          printf("-d turns off deemphasis \n");
          printf("   results are scaled and ready to output to the D/A converter
          in LSI\n");
          exit();
}

filter(argv)
char *argv[];
{
        int fpe();
        int count,samples,framelength,atoi(),last_frame;
      int i,j,k,l,output,rows,poles,scale,scaleflag = 1,agc;
        long filesize,r[NO_AUTOC],overflow_count,pitch_period,r_full_scale;
float y[NO_AUTOC],a[NO_AUTOC],newa[NO_AUTOC],*x,gain,sqrt(),fabs(),magn,out,fscale,fagc;
        float noise[LOTS],nat_excit[256],simple_excit[256],Gain,comp_gain();
```

```
FILE *fopen(),*fp_auto,*fp_output,*fp_pitch,*fp_excitation;

  signal(8,&fpe); /* this is a function which allows programmer to jump
                   * to a subroutine when a floating point error occurs
                   */

if ((fp_excitation=fopen("excitation","w")) == NULL)
        ex("cannot create file 'excitation'");

if ((fp_auto=fopen(argv[1],"r")) == NULL)
  ex("cannot open file of autocorrelation data");

if ((fp_output = fopen(argv[2],"w")) == NULL)
  ex("can't create output file");

  if ((fp_pitch = fopen(pitch_file,"r")) == NULL)
  ex("pitch file cannot be opened");


/* initialization */

  overflow_count = 0;
gain = 0.0;
count = -1;
for ( i= 0; i<NO_AUTOC; i++)
{ y[i] = 0.0;
  a[i] = 0.0;
  r[i] = 0;
  }
  a[0] = 1.0;

/* compute length of the file containing auto values */

  stat(argv[1],&stbuf);
  filesize = stbuf.st_size;

/* determine number of rows */

  framelength = samp_freq/frame_rate;
  printf("framelength=%d samples \t\tframe_rate=%d Hz\n",
  framelength,frame_rate);
rows = filesize/(2*(NO_AUTOC + scaleflag));
printf("frames = %d\n",rows);

  /*set up arrays of excitations*/

  set_excit(noise,nat_excit,simple_excit,framelength);

/* BEGIN SYNTHESIS */

for ( l=0; l<rows; l++)
{     printf("frame %4d     ",l+1);
```

```
    for (i=0;i<NO_AUTOC;i++)
        r[i] = getsh(fp_auto);  /* read in autocorrelation values */

    if (gainflag == 1)
      {   fscale = getsh(fp_auto);
          if (fscale == 0.0) fscale = 1.0; /* can't have fscale==0 */
          if(l == 0) r_full_scale = r[0];
          fscale = fscale * sqrt(255.0/2047.0);/* scale factor */
          /*
          printf("fscale is %f\t",fscale);
          fscale = fscale * fscale * sqrt(r_full_scale / 2047.0);
          printf("fscale is %f\n",fscale);
          */
      }
      else if (agcflag == 1)
              fagc = 0.0001 + (agc = getsh(fp_auto));
          else getsh(fp_auto); /* just skip over the scale factor
                               * (which is 1) if no gain option
                               */

/* get pitch */

    pitch_period=getsh(fp_pitch);
    if (feof(fp_pitch) != 0)   /*check for EOF in pitch file*/
    { printf("\n\npitch file %s is too small. Need at
      least %d pitch periods\n",pitch_file,rows);
          return(-1);
    }
    samples=(pitch_period * samp_freq)/10000; /*convert pitch period
      * to # of samples at sampling freq; pitch_period is in units of
      * 0.1msec (assumes pitch period is from Monte's bread-board or
      * Herb's span program)
      */

    printf("Pitch period is %4.1fmsec\n",pitch_period/10.0);

 /*set x to point to the start of the desired excitation array*/
    if (samples > 0)
      {       x = nat_excit;  /* voiced */
              if (last_frame == UV) count = -1; /* reset count on a */
              last_frame = V;                   /* uv/v transition  */
      }
    else
      {       x = noise;      /*unvoiced*/
              samples = LOTS;
              last_frame = UV;
      }

 /* do durbin's recursion on new data. Use the new values
  * obtained from durbin's recursion only if it works (i.e.
  * number of poles = 9)          */

    if (floatflag == 1)         /*use floating point version*/
```

```
    { if ((poles=durbinf(r,newa,&gain,0)) == NO_POLES)
                    for (i=0;i<NO_AUTOC;i++)
                            a[i] = newa[i];
    }

    else                            /*use integer version*/
    { if ((poles=durbin(r,newa,&gain,0,0)) == NO_POLES)
                    for (i=0;i<NO_AUTOC;i++)
                            a[i] = newa[i];
    }

  printf("gain = %7.3f     poles = %d\n",Gain=
      comp_gain(gain,fagc,fscale),poles);

      /* compute filter outputs for a frame. Synthesis filter
       * has the form: y(n)=a[1]*y(n-1)+a[2]*y(n-2)+..+G*x(n).
       * y(n) {present output} is stored in y[0], y(n-1) is
       * in y[1],...etc. x(n)=x[n]=excitation
       */
  for ( k=0; k < framelength; k++)
  {    if(count >= samples) count=0;
       else              count++;

       for (i=1; i<NO_AUTOC; i++)
        y[0] = a[i]*y[i] + y[0];   /*y[0] is the present output*/

      /* TESTING VAX random number generator */
          putsh((short)x[count],fp_excitation);

          y[0] = y[0] + x[count]*Gain/856.0;
      output = (out = y[0]*500.0);
          if((magn=fabs(out)) > 32767.0)
          {  output = 32767.0 * out/magn;
                  overflow_count++;
                  printf("**overflow in computation. y[0]=%f\n",out);
           }

       putsh(output,fp_output);

       for (j=0;j<NO_POLES;j++)   /*shift data*/
          y[NO_POLES-j] = y[NO_POLES-1-j];

       y[0] = 0.0;
  }

}
fclose(fp_excitation);
fclose(fp_auto);
fclose(fp_output);
fclose(fp_pitch);
  if (overflow_count != 0L)
       printf("%ld overflows occured in the synthesis filter\n",
       overflow_count);
```

```
        if (unstble_frames != 0)
            printf("%d unstable frames \n",unstble_frames);
        return(0);

}

set_excit(noise,nat_excit,simple_excit,framelength)
float noise[LOTS],nat_excit[256],simple_excit[256];
int framelength; /* Why is this arguement listed ? */

{       int i,rand(),Rand();

/* expected value of noise squared = 1791393   */
        for(i=0;i<LOTS;i++)
                noise[i] = (Rand(0) - 16383.5) * 0.02002;

/* this input waveform is not used in this version of Speakpitch.c */
/* energy in simple_excit = 1782150   */
        for(i=0;i<256;i++)
                simple_excit[i] = 0;
                simple_excit[28] = -545;
                simple_excit[29] = 1090;
                simple_excit[30] = -545;


/* energy in nat_excit = 1791393      */
              nat_excit[0] = 37;
            nat_excit[1] = 83;
            nat_excit[2] = 96;
            nat_excit[3] = 107;
            nat_excit[4] = 76;
            nat_excit[5] = 84;
            nat_excit[6] = 90;
            nat_excit[7] = 77;
            nat_excit[8] = 89;
            nat_excit[9] = 63;
              nat_excit[10] = 80;
            nat_excit[11] = 62;
            nat_excit[12] = 64;
            nat_excit[13] = 49;
            nat_excit[14] = 44;
            nat_excit[15] = -11;
            nat_excit[16] = 9;
            nat_excit[17] = -65;
            nat_excit[18] = -63;
            nat_excit[19] = -21;
            nat_excit[20] = -4;
            nat_excit[21] = -43;
            nat_excit[22] = -159;
            nat_excit[23] = -239;
            nat_excit[24] = -320;
            nat_excit[25] = -276;
            nat_excit[26] = -211;
```

```
            nat_excit[27] = -238;
            nat_excit[28] = 155;
            nat_excit[29] = 856;
            nat_excit[30] = -264;
            nat_excit[31] = -404;
            nat_excit[32] = 233;
            nat_excit[33] = -266;
            nat_excit[34] = 293;
            nat_excit[35] = -189;
            nat_excit[36] = 177;
            nat_excit[37] = -131;
            nat_excit[38] = 105;
            nat_excit[39] = -134;
            nat_excit[40] = 109;
              nat_excit[41] = -71;
              nat_excit[42] = 44;
        for(i = 43;i<256;i++)
            nat_excit[i] = 0;

        return;
}

/* Rand() bridges the incompatability gap between the 11/40 and VAX. This
 * function returns a random number between 0 and 32767 on both machines.
 */

#define MASK 0x00007fffL /* masks all but last 15 bits on a long integer */
Rand(seed)
int seed;
{       int rand(),noise; /* want 'noise' to be a positive random
                           * number between 0 and 32767
                           */

        noise = (int) ((long)rand(seed) & MASK);
        return(noise);
}

#define MAX_32BITS ((1 << 31) - 1)

/* This function accepts 10 autocorrelation values as input and computes
 * the filter coefficients (the a(i)'s). */

/* Blows up on frame 323 of 'gen audiocritic.t -a'
 * Because of this, this function is no longer the default.
 */

durbin(r,newa,gain,delta,shiftcnt)
int delta,shiftcnt;
long r[NO_AUTOC];
float newa[NO_AUTOC],*gain;

{       long r0,e;
        long a[NO_AUTOC][NO_AUTOC];
```

```
        short i,j,k,poles,ONE,shiftflag;
        float g,sqrt(),newk[NO_POLES],pow();
        float tempsum,fabs();

/*printf("integer version of Durbin's Recursion\n");*/

        /* ONE = 16384; this works fine with long integers */
        ONE = 32767;
        shiftflag = 0;
        if(delta == 0) printf("R[0]=%5ld   ",r[0]);
        else           printf("             ");

        r0 = r[0];
        while(r0 < (1 << 14))
        {       r0 = r0 << 1;
                shiftcnt++;
                shiftflag = 1;
        }

        if(shiftflag)
                for(i=0;i<NO_AUTOC;i++) /* minimize roundoff error */
                        r[i] = r[i] << shiftcnt;

        for(i=1;i<NO_AUTOC;i++)
                newa[i]=0.0;
        newa[0] = 1.000;

   /*this is Durbin's recursion. See J.Makhoul paper. a[j][i]=(a sup j)sub i */

        r[0] = r[0] + delta;   /*try to fix ill-conditioned matrix*/

        e = r[0];
        printf("R[0]=%5ld    ",r[0]);
        if (e <= 0L) i=1;

        else for(i=1;i<NO_AUTOC;i++)
        {       tempsum = 0;

                for(j=1;j<=i-1;j++)
                {       tempsum = tempsum + (int) (a[i-1][j] * r[i-j]);
                        if(fabs(tempsum) > MAX_32BITS)
                                fprintf(stderr,"TEMPSUM OVERFLOW %f",tempsum);
                }

                a[i][i] = ((r[i]*(long)ONE - tempsum)/e);
                        printf("K[%d]=%ld\t",i,a[i][i]);

                if (( a[i][i]  >=  ONE) || ( a[i][i] <= -1*ONE))
                        if (unstableflag == 1)
                        {       printf("refl. coeff >= ONE !!  k[%d] = %7.2f\n",
                        i,a[i][i]/(float)ONE);
                                if(delta == 0) unstble_frames++;
                                if ((poles=durbin(r,newa,gain,compute_delta(r[0]),
```

```
                        shiftcnt))==NO_POLES) return(poles);
                        /* if ((poles=durbin(r,newa,&gain,compute_delta(r[0])
                        shiftcnt))==NO_POLES) return(poles); old line -
                        wrong (I think) */
                        /* increment R[0] by .1% and redo durbin if unstable */
                }
                else { unstble_frames++; break;}

        for(j=1;j<=i-1;j++)
        {       a[i][j] = a[i-1][j] - ((a[i][i] * a[i-1][i-j])/ONE);
                /* a[i][i] * a[i-1][i-j] overflows on frame 323 */
                printf("a[%d][%d] = %ld ",i,j,a[i][j]);
                if(abs(a[i][j]) > MAX_24BITS)
                {       fprintf(stderr,"a[%d][%d] = %ld\n",i,j,a[i][j]);
                        fprintf(stderr,"^ a[][] exceeds 24 bits!!\n");
                }
        }

        /* e = ((ONE - ((a[i][i] * a[i][i])/ONE)) * e)/ONE; */
        /* e = (((ONE * (long)ONE - a[i][i] * a[i][i])/ONE) * e)/ONE; */
        e = e - (int) (a[i][i] * a[i][i] / ONE) * e / ONE; /* has the nice
        feature that e >= (theoretical value of e) */

        if (e == 0) { fprintf(stderr,"e == 0\n");e = 1;}
}

poles = i-1;
if (poles < NO_POLES) return(poles); /* don't compute a new gain value wih
                        * bad data--just return and use old data*/
for(j=1;j<=poles;j++)
{
        newa[j] = a[poles][j]/((float)ONE);
        if(delta != 0)
                printf("newa[%d] = %f ",j,newa[j]);
        newk[j-1] = a[j][j]/((float)ONE);
        if(delta != 0)
                printf("newk[%d] = %f ",j-1,newk[j-1]);
}

/* note that lpc_gain^2 = lpc_error */
/* note: If R[0] has been incremented a number of times, R[0] and
 * therefore e are too large -> gain is too large.
 */
*gain = sqrt((float)e/pow(2.0,(float)shiftcnt));
/* scale gain to undo '<< shiftcnt' above */

return(poles);
}

old_durbin(r,newa,gain,delta)
int delta;
long r[NO_AUTOC];
float newa[NO_AUTOC],*gain;
```

```
{       long a[NO_AUTOC][NO_AUTOC],e,tempsum;
        int i,j,k,poles,ONE;
        float g,sqrt();

/*printf("integer version of Durbin's Recursion\n");*/

        ONE = 10000;

        for(i=1;i<NO_AUTOC;i++)
                newa[i]=0.0;
        newa[0] = 1.000;

   /*this is Durbin's recursion. See J.Makhoul paper. a[j][i]=(a sup j)sub i */
        r[0] = r[0] + delta;   /*try to fix ill-conditioned matrix*/

        e = r[0];
        printf("R[0]=%5ld    ",e);
        if (e <= 0L) i=1;

        else for(i=1;i<NO_AUTOC;i++)
        {       tempsum = 0;

                for(j=1;j<=i-1;j++)
                        tempsum = tempsum + (a[i-1][j] * r[i-j]);

                a[i][i] = (r[i]*ONE - tempsum)/e;
                /*printf("k[i]=%ld\t",a[i][i]);*/

                if (( a[i][i]  >=  ONE) || ( a[i][i] <= -1*ONE))
                        if (unstableflag == 1)
                        {       printf("refl. coeff >= ONE !!  k[%d] = %ld\n",i,a[i][i]);
                                if(delta == 0) unstble_frames++;
                                if ((poles=old_durbin(r,newa,&gain,compute_delta (r[0]
                                ==NO_POLES) return(poles);
                                /* increment R[0] by .1% and redo durbin if unstable */
                        }
                        else { unstble_frames++; break;}

                for(j=1;j<=i-1;j++)
                        a[i][j]  = a[i-1][j] - ((a[i][i] * a[i-1][i-j])/ONE);

                e = ((ONE - ((a[i][i] * a[i][i])/ONE)) * e)/ONE;

                if (e == 0L) e = 1L;
        }

        poles = i-1;
        if (poles < NO_POLES) return(poles);/* don't compute a new gain value wih
                                * bad data--just return and use old data*/
        g = 0.0;

        for(j=1;j<=poles;j++)
        {
```

```
                newa[j] = a[poles][j]/10000.0;
                g = g - newa[j] * r[j];
        }

        g = g + r[0];
        if (g < 0.0) g=0.0;
        *gain = sqrt(g);

        return(poles);
}

#define fepsilon 0.00001

durbinf(r,newa,gain,delta)
int delta;
long r[NO_AUTOC];
float newa[NO_AUTOC],*gain;

{       float a[NO_AUTOC][NO_AUTOC],e,tempsum,ONE;
        int i,j,k,poles;
        float g,sqrt();

/*printf("float version of Durbin's Recursion\n");*/

        ONE = 1.0000;

        for(i=1;i<NO_AUTOC;i++)
                newa[i]=0.0;
        newa[0] = 1.000;

   /*this is Durbin's recursion. See J.Makhoul paper. a[j][i]=(a sup j)sub i */
        r[0] = r[0] + delta;

        e = r[0];
        printf("R[0]=%5.0f   ",e);
        if (r[0] <= 0L) i=1;

        else for(i=1;i<NO_AUTOC;i++)
        {       tempsum = 0;

                for(j=1;j<=i-1;j++)
                        tempsum = tempsum + (a[i-1][j] * r[i-j]);

                a[i][i] = (r[i]*ONE - tempsum)/e;

                if (( a[i][i]  >=  ONE) || ( a[i][i] <= -1*ONE))
                        if (unstableflag == 1)
                        {       printf("refl. coeff >= ONE !!  k[%d] = %f\n",i,a[i][i]);
                                if (delta == 0) unstble_frames++;
                                delta = compute_delta(r[0]);
                              /*if ((poles=durbinf(r,newa,&gain,delta))==NO_POLES)
                                old line of code - wrong (I think) */
                                if ((poles=durbinf(r,newa,gain,delta))==NO_POLES)
```

```
                                return(poles);
                        }
                        else { unstble_frames++; break;}

                for(j=1;j<=i-1;j++)
                        a[i][j]  = a[i-1][j] - ((a[i][i] * a[i-1][i-j])/ONE);

                e = ((ONE - ((a[i][i] * a[i][i])/ONE)) * e)/ONE;
                /* if (e == 0L) e = 1L; */
        }

        poles = i-1;
        if (poles < NO_POLES) return(poles);

        g = 0.0;

        for(j=1;j<=poles;j++)
        {
                newa[j] = a[poles][j]/1.0000;
                g = g - newa[j] * r[j];
        }

        g = g + r[0];
        if (g < 0.0) g=0.0;
        *gain = sqrt(g);

        return(poles);
}

compute_delta(R0)
long R0;
{       int delta;

        delta = 0.001*R0;
        if (delta == 0) delta = 1;
        return(delta);
}

ex(str)
char *str;
{       printf("%s \n",str);
        exit();
}

float comp_gain(gain,fagc,fscale)
float gain,fagc,fscale;
{
        if (gainflag == 1)     return(gain/fscale);
        else if (agcflag == 1)  return(gain*fagc/800.0);/* 800 is the magic   */
           else                 return(gain);       /* number which gives   */
}                                                   /* Gain(agc)~Gain(fscale)*/

/* floating point error causes a jump to this function */
```

```
fpe()
{       printf("Floating point error\n");
        /*printf("Gain=%f\nCount=%d\nOutputn-1=%f\nOutputn=%f\n\n",Gain,
        Count,Outputn_1,Outputn);
        printf("Fagc=%f\nFscale=%f\n\n",Fagc,Fscale);
        */
        ex("");
}


#define MAX_LINE 500
#define NULL_CHAR ' '
#define BLANK ' '
#define RETURN '\n'

sys_call(string,arg)
char string[MAX_LINE],*arg;
{       char command[MAX_LINE],*getenv(),**ptr;
        int i;

        ptr = &arg;
        sprintf(command,"%c",BLANK);

        i=0;

        while( string[i] != NULL_CHAR )
                if( string[i] == '%' )
                {       switch(string[++i])
                        {       case 's':
                                        sprintf(command,"%s%s",command,*(ptr++));
                                        break;
                                case 'h':
                                        sprintf(command,"%s%s",command,
                                        getenv("HOME"));
                                        break;
                                case '%':
                                        sprintf(command,"%s%c",command,string[i]);
                                        break;
                                case 'd':
                                        sprintf(command,"%s%d",command,*(ptr++));
                                        break;
                                default:
                                        sprintf(command,"unknown conversion type
                                        '%%%c'",string[i]);
                                        ex(command);
                                        break;
                        }
                        i++;
                }
                else sprintf(command,"%s%c",command,string[i++]);

        system(command);
        return;
```

}

Pitch information is required to synthesize speech. To determine the pitch for the synthesized speech, a modified Gold-Rabiner pitch tracker was used [20]. The source code for this program is not included here.

2.3. SPEC_DEV.C

To quantify errors due to the various system modifications, an Itakura-Saito spectral distance measure was employed (equation 2.15) . The program spec_dev.c takes two sets of autocorrelation values and computes the spectral distance between them. In the simulations of chapter 4, one set of autocorrelation values was computed with high accuracy (no modifications), and the other set was computed with one or more modifications in effect. The spectral distance measured is referred to as a spectral deviation since it is the distance measured from a very accurate, reference frame.

SPEC_DEV.c:

```
#include <stdio.h>
#ifdef FORTY
#include "/usr/audio/hurst/source/VAX_header
#endif
#define NO_POLES 9
#define NO_AUTOC 10
#define OUT_SCALE 100
#define MAX_SHORT 32767
#define ERROR_FLAG -1000 /* number to output in place of rms_error if
                          * an unstable frame is encountered
                          */

int scaleflag =1;
int unstableflag=0; /* causes durbinx() to terminate on an unstable k */
int unstble_frames;
int quant_type=0; /* 0 -> use floating point arithmetic */
                  /* 1 -> use long arithmetic */
                  /* 2 -> use 16 bit integers wherever possible */
```

```
main(argc,argv)
int argc;
char **argv;
{
        /* short durbinsh(),durbinl(),durbinf(); */
        int atoi(),i,j,frame=0;
        long r[NO_AUTOC],r_ref[NO_AUTOC];
        float a[NO_AUTOC],a_ref[NO_AUTOC],gain,rms_error,numer,spec_dev;
        float durbinf(),error(),log();
        FILE *fopen(),*fp_ref_auto,*fp_other_auto,*fp_error;

        if(argc == 1)
                ex("usage: spec_dev ref other  [-fls] \n\t-f is the default");

        i=3;
        while(++i <= argc)
        {       if (argv[i-1][0] != '-')
                        ex ("bad parameter");
                else switch(argv[i-1][1])
                {       case 'f': quant_type = 0; /* use durbinf() */
                                break;
                        case 'l': quant_type = 1; /* use durbinl() */
                                break;
                        case 's': quant_type = 2; /* use durbinsh() */
                                break;
                        default:  ex("bad parameter");
                }
        }

        if((fp_ref_auto=fopen(argv[1],"r")) == NULL)
                ex("input file does not exist");
        if((fp_other_auto=fopen(argv[2],"r")) == NULL)
                ex("input file does not exist");

        if((fp_error=fopen("dev","w")) == NULL)
                ex("cannot create 'dev' ");

        while(1)
        {
                for(j=0;j<NO_AUTOC;j++)
                {       r_ref[j] = getsh(fp_ref_auto);
                        r[j] = getsh(fp_other_auto);
                        if(feof(fp_ref_auto) == 1)
                                if(j != 0) ex("bad data count in input file");
                                else exit(0);
                        if(feof(fp_other_auto) == 1)
                                ex("bad file size in other");
                }

                if(scaleflag == 1) getsh(fp_ref_auto); /* skip gain value */
                if(scaleflag == 1) getsh(fp_other_auto); /* skip gain value */

                rms_error = durbinf(r_ref,a_ref,&gain,0); /* !! not NORMALIZED !! */
```

```
                if(durbinf(r,a,&gain,0) < 0) /* durbinf failed */
                        numer = -10; /* indicates error */

                else numer = error(r_ref,a); /* !! not NORMALIZED !! */

                if(rms_error <= 0)
                        spec_dev = -(float) (MAX_SHORT / OUT_SCALE);
                        /* indicates an unstable frame in reference data */
                else if(numer <= 0)
                        spec_dev = (float) (MAX_SHORT / OUT_SCALE);
                        /* indicates an unstable frame in other data */
                   else spec_dev = 4.343 * log(numer / rms_error);
                printf("frame %3d spec_dev = %f db\n",++frame,spec_dev);
                /* printf("normalized rms_error = %f for frame %d\n",
                (rms_error * MAX_SHORT / r[0]),++frame);*/

                /* output spectral deviation in units of 0.01 db
                 */
                putsh((short) (spec_dev * OUT_SCALE),fp_error);

        }
}

/* This function accepts 10 autocorrelation values as input and computes
 * the filter coefficients (the a(i)'s). */

short durbinsh(r,newk,gain,delta)
int delta;
long r[NO_AUTOC]; /* ?? could be a short since R[0] <= 2047 */
float newk[NO_POLES],*gain;

{       long a[NO_AUTOC][NO_AUTOC],tempsum; /* e could be a short since
                                            * e <= R[0]. tempsum must
                                            * be a long. a[][] is up
                                            * in the air. See td_results
                                            * for limits on a[][]'s
                                            */
        short i,j,k,e,r_sh[NO_AUTOC],r_scale,poles,ONE;

        float g,sqrt(),newa[NO_AUTOC]; /* God forbid we should really need
                                        * a float. Better check this out.
                                        */

/*printf("integer version of Durbin's Recursion\n");*/

        ONE = 32767; /* Different value for ONE results in different rms_error */

        r_scale = ONE / r[0];
        for(i=0;i<NO_AUTOC;i++)
                r_sh[i] = r[i] * r_scale;
        r[0] = r_sh[0]; /* return r_sh[0] for proper scaling in print in main */
```

```
    for(i=1;i<NO_AUTOC;i++)
            newa[i]=0.0;
    newa[0] = 1.000;

/*this is Durbin's recursion. See J.Makhoul paper. a[j][i]=(a sup j)sub i */
    r_sh[0] = r_sh[0] + delta;   /*try to fix ill-conditioned matrix*/

    e = r_sh[0];
    /* printf("R[0]=%5d ",e); */
    if (e <= 0L) i=1;

    else for(i=1;i<NO_AUTOC;i++)
    {       tempsum = 0;

            for(j=1;j<=i-1;j++)
                    tempsum = tempsum + (a[i-1][j] * r_sh[i-j]);

            a[i][i] = (r_sh[i]*(long)ONE - tempsum)/e;
            printf("k[%d]=%f\t",i,a[i][i]/((float) ONE));

            if (( a[i][i]  >=  ONE) || ( a[i][i] <= -1*ONE))
                    if (unstableflag == 1)
                    {       printf("refl. coeff >= ONE !!  k[%d] = %7.2f\n",
                    i,a[i][i]/(float)ONE);
                            if(delta == 0) unstble_frames++;
                        if ((poles=durbinsh(r,newk,&gain,compute_delta(r[0])))
                            ==NO_POLES) return((short) e);
                            /*increment R[0] by .1% and redo durbin if unstable*/
                    }
                    else { unstble_frames++; break;}

            for(j=1;j<=i-1;j++)
                    a[i][j]  = a[i-1][j] - ((a[i][i] * a[i-1][i-j])/ONE);

            /* e = ((ONE - ((a[i][i] * a[i][i])/ONE)) * e)/ONE; */
            e = (((ONE * (long)ONE - a[i][i] * a[i][i])/ONE) * e)/ONE;


            if (e == 0) e = 1;
    }

    poles = i-1;
    if (poles < NO_POLES) return(ERROR_FLAG);/* don't compute a new gain
                    * value with bad data--just return and use old data*/
    g = 0.0;

    for(j=1;j<=poles;j++)
    {
            newa[j] = a[poles][j]/((float)ONE);
            newk[j-1] = a[j][j]/((float)ONE);
            g = g - newa[j] * r_sh[j];
    }
```

```
        g = g + r_sh[0];
        /* just found out (should have known) that g = e */
        /* printf("g = %f\te = %d\t",g,e); */
        if (g < 0.0) g=0.0;
        *gain = sqrt(g);

        return((short) e);
}


/* This function accepts 10 autocorrelation values as input and computes
 * the filter coefficients (the a(i)'s). */

short durbinl(r,newk,gain,delta)
int delta;
long r[NO_AUTOC]; /* ?? could be a short since R[0] <= 2047 */
float newk[NO_POLES],*gain;

{       long a[NO_AUTOC][NO_AUTOC],e,tempsum; /* e could be a short since
                                        * e <= R[0]. tempsum must
                                        * be a long. a[][] is up
                                        * in the air. See td_results
                                        * for limits on a[][]'s
                                        */
        int i,j,k,poles,ONE; /* On the VAX, these are the same as longs.
                              * Change to shorts wherever possible.
                              */

        float g,sqrt(),newa[NO_AUTOC]; /* God forbid we should really need
                                        * a float. Better check this out.
                                        */

/*printf("integer version of Durbin's Recursion\n");*/

        ONE = 10000;

        for(i=1;i<NO_AUTOC;i++)
                newa[i]=0.0;
        newa[0] = 1.000;

   /*this is Durbin's recursion. See J.Makhoul paper. a[j][i]=(a sup j)sub i */
        r[0] = r[0] + delta;  /*try to fix ill-conditioned matrix*/

        e = r[0];
        /* printf("R[0]=%5ld       ",e); */
        if (e <= 0L) i=1;

        else for(i=1;i<NO_AUTOC;i++)
        {       tempsum = 0;

                for(j=1;j<=i-1;j++)
                        tempsum = tempsum + (a[i-1][j] * r[i-j]);
```

```
                a[i][i] = (r[i]*ONE - tempsum)/e;
                printf("k[i]=%f\t",a[i][i]/((float) ONE));

                if (( a[i][i]  >=  ONE) || ( a[i][i] <= -1*ONE))
                        if (unstableflag == 1)
                        {       printf("refl. coeff >= ONE !!  k[%d] = %7.2f\n",
                                i,a[i][i]/(float)ONE);
                                if(delta == 0) unstble_frames++;
                                if ((poles=durbinl(r,newk,&gain,compute_delta(r[0])))
                                ==NO_POLES) return((short) e); /* increment R[0]
                                by .1% and redo durbin if unstable */
                        }
                        else { unstble_frames++; break;}

                for(j=1;j<=i-1;j++)
                        a[i][j]  = a[i-1][j] - ((a[i][i] * a[i-1][i-j])/ONE);

                e = ((ONE - ((a[i][i] * a[i][i])/ONE)) * e)/ONE;

                if (e == 0L) e = 1L;
        }

        poles = i-1;
        if (poles < NO_POLES) return(ERROR_FLAG);/* don't compute a new gain vi
                                * bad data--just return and use old data*/
        g = 0.0;

        for(j=1;j<=poles;j++)
        {
                newa[j] = a[poles][j]/((float)ONE);
                newk[j-1] = a[j][j]/((float)ONE);
                g = g - newa[j] * r[j];
        }

        g = g + r[0];
        if (g < 0.0) g=0.0;
        *gain = sqrt(g);

        return((short) e);
}

#define fepsilon 0.00001

float durbinf(r,newa,gain,delta)
int delta;
long r[NO_AUTOC];
float newa[NO_AUTOC],*gain;

{       double a[NO_AUTOC][NO_AUTOC],e,tempsum,ONE;
        int i,j,k,poles;
        double g,newk[NO_POLES];
        float sqrt();
```

```
/*printf("float version of Durbin's Recursion\n");*/

        ONE = 1.0000;

        for(i=1;i<NO_AUTOC;i++)
                newa[i]=0.0;
        newa[0] = 1.000;

    /*this is Durbin's recursion. See J.Makhoul paper. a[j][i]=(a sup j)sub i */
        r[0] = r[0] + delta;

        e = r[0];
        /* printf("R[0]=%6.0f        ",e); */
        if (r[0] <= 0L) i=1;

        else for(i=1;i<NO_AUTOC;i++)
        {       tempsum = 0;

                for(j=1;j<=i-1;j++)
                        tempsum = tempsum + (a[i-1][j] * r[i-j]);

                a[i][i] = (r[i]*ONE - tempsum)/e;
                /* printf("k[%d]=%f\t",i,a[i][i]); */

                if (( a[i][i]  >=  ONE) || ( a[i][i] <= -1*ONE))
                        if (unstableflag == 1)
                        {       printf("refl. coeff >= ONE !!  k[%d] = %f\n",i,a[i][i]);
                                if (delta == 0) unstble_frames++;
                                delta = compute_delta(r[0]);
                                if ((poles=durbinf(r,newa,&gain,delta))==NO_POLES)
                                        return(e);
                        }
                        else { unstble_frames++; break;}

                for(j=1;j<=i-1;j++)
                        a[i][j]  = a[i-1][j] - ((a[i][i] * a[i-1][i-j])/ONE);

                e = ((ONE - ((a[i][i] * a[i][i])/ONE)) * e)/ONE;
                /* if (e == 0L) e = 1L; */
        }

        poles = i-1;
        if (poles < NO_POLES) return((float)ERROR_FLAG);

        g = 0.0;

        for(j=1;j<=poles;j++)
        {
                newa[j] = -a[poles][j]/ONE;  /* fixing sign for error()  */
                newk[j-1] = a[j][j]/ONE;
                g = g - newa[j] * r[j];
        }
```

```
        g = g + r[0];
        if (g < 0.0) g=0.0;
        *gain = sqrt(g);

        return(e);
}

compute_delta(R0)
long R0;
{       int delta;

        delta = 0.001*R0;
        if (delta == 0) delta = 1;
        return(delta);
}

float error(r,a)
long r[];
float a[];
{
        int j;
        float err = 0.0;
        float R(),B();


        for (j= -NO_POLES;j<=NO_POLES;j++)
                err = err + B(j,a) * R(j,r);

        return(err);
}

/* R(j,.) returns autocorrelation value of input. Function is necessary
 * because error() uses R(j,.) and R(-j,.)
 */
float R(j,r)
long r[];
int j;
{
        if(j < 0) j = -1 * j;

        return((float)r[j]);
}

/* B(j,a) computes autocorrelation values for the LPC coeffs. Reference is
 * G. White paper (Xerox PARC) 'Automatic Speech Recognition, LPC Residual
 * vs. BP Filtering
 */
float B(j,a)
float a[];
int j;
{
        int i;
        float b = 0.0;
```

```
        if(j < 0) j = -1 * j;

        for(i=0;i<=(NO_POLES - j);i++)
                b = b + a[i] * a[i+j];

        return(b);
}

ex(str)
char *str;
{       fprintf(stderr,"%s \n",str);
        exit();
}
```

APPENDIX D

THE CMOS PROCESS SCHEDULE

The process schedule for the high voltage metal-gate CMOS process which was used to integrate the autocorrelator is presented here. This process is a modified version of the CMOS process used by Black [28]. More information on the process and layout rules can be found in [14].

This process uses positive photo-resist throughout. Alignment was done with a projection aligner to ~1μm precision. The p-channel devices were self aligned by an implant over the metal. The aluminum gates were used as the mask and the sintering step activated the implant.

Berkeley Metal Gate CMOS Process
November 18, 1981 (rev. March 5 1982)

1. Initial wafer cleaning - p-type, 5-7 ohm-cm, <100>
   a. TCE clean, 60 °C, 10 minutes (Degrease) (Watch temperature : Boiling TCE will shatter wafers)
   b. Dip in acetone
   c. Dip in methyl alcohol
   d. Rinse in De-ionized (DI) water
   e. Piranha etch (H_2SO_4:H_2O_2 - 5:1) for 10 min.
   f. Rinse thoroughly in running DI water
   g. Dip in HF:H_2O (1:10) for 20 seconds
   h. Rinse in DI water
   i. Blow dry
   j. Inspect under collimated light for dust. (If dust remains, repeat from h.; if that doesn't work, repeat from a.)

2. Initial oxidation - 4000A, Initial Oxide Furnace
   a. Wet O_2, 1100 °C, 27 min. (.4μm) (temperature setting=1095 °C)
   b. Dry N_2 (anneal), 900 °C, 20 min.

3. Positive Photoresist (PR) Step, Mask #1 : N- well mask.
   a. Prebake, 85-90 °C, 15 min. (if wafer just came from furnace then skip prebake)
   b. HMDS treatment
      1. N_2 purge, 10 min.
      2. Load wafers
      3. Bubble HMDS, 3 min.
      4. N_2 purge, 5 min.
   c. Spin AZ 1450J, 6000 rpm, 30 seconds. (PR thickness = 1.4μm)
   d. Softbake, 85-90 °C, 15-20 min.
   e. Align and expose (Blow off dust on top and bottom)
   f. Develop, AZ 351 developer (DI H_2O:Developer - 5:1), 1 min. (or regular AZ developer at 1:1 dilution)
   g. Rinse in DI water and blow dry
   h. Inspect
   i. Hardbake, 110-120 °C, 20 min.
   j. Oxide etch, buffered HF (NH_4F:HF - 5:1) etch rate .12μm/min. (.145μm/min if solution is just made) (etch time= 3.5-4 min. with 15% over etch) (prepare at least 4 hours before use)
   k. PR strip, acetone, 15 min.
   l. Piranha clean, H_2SO_4:H_2O_2 - 5:1, 15 min.

4. Implant step - Initial Oxide Furnace
   a. Grow implant oxide - 290A
      1. Wet O_2, 900 °C, 3 min., flowmeter setting=4 cm
      2. Dry N_2, 900 °C, 5 min., flowmeter setting=4 cm
   b. Implant Phosphorus (N-), Dose=$1x\,10^{13}$, 200keV (xj(well)=.462μm, Rsh=1.29kohm/square)

5. Piranha Clean, H_2SO_4:H_2O_2 - 5:1, 5 min.

6. Well oxidation - 7229A, Initial Oxide Furnace
   a. Wet O_2, 1150 °C, 65 min., (.76/.92μm) (setting=1144)
   b. Dry N_2, 900 °C, 30 min. (xj(well)=1.978μm, Rsh=943 ohms/square)

7. Well drive-in, Arsenic Furnace
   a. Dry N_2, 1200 °C, 24 hours
   b. Dry N_2 (anneal), 900 °C, 60 min. (xj(well)=9.055μm, Rsh=827 ohms/square)

8. Pos. PR Step, Mask #2 : N+ (N channel S/D) mask.
   a. Apply photoresist, expose, develop (follow 3.a-i)
   b. Oxide etch, buffered HF (NH_4F:HF - 5:1) etch rate .115μm/min. (etch time=7.5 min.)
   c. PR strip
   d. Piranha clean, 15 min.

9. Implant step - Initial Oxide Furnace
   a. Grow implant oxide - 290A
      1. Wet O_2, 900 °C, 3 min.
      2. Dry N_2, 900 °C, 5 min.
   b. Implant Phosphorus (N+), Dose=$5.84x\,10^{15}$, 160keV

10. Piranha Clean, 5 min.

11. Oxide growth over N+ - 3983A, N+ Drive-in Furnace
    a. Wet O_2, 950 °C, 60 min.
    b. Dry N_2, 900 °C, 30 min. (xj(N channel S/D)=0.476μm, Rsh=27.88 ohms/square)

12. Pos. PR Step, Mask #3 : P+ (P channel S/D) mask.
    a. Apply photoresist, expose, develop (follow 3.a-i)
    b. Oxide etch, buffered HF (NH_4F:HF - 5:1) etch rate .115μm/min. (etch time=7.5 min.)
    c. PR strip

d. Piranha clean, 15 min.

13. Implant step - P+ Drive-in Furnace
    a. Grow implant oxide - 290A
        1. Wet O_2, 900 °C, 3 min. (.29/.40μm)
        2. Dry N_2, 900 °C, 5 min.
    b. Implant Boron (P+), Dose=5.50x10^{15}, 32keV (xj(P channel S/D)=.598μm, Rsh=22.28 ohms/square, Rsh(well)=958 ohms/square)

14. Gettering step :
    Implant Boron Difluoride into back side of wafer
    Dose=5x10^{15}, 200 keV

15. Piranha Clean, 5 min.

16. Oxide growth over P+ and N+/P+ drive-in - 3200A, P+ drive-in Furnace
    a. Wet O_2, 1100 °C, 15 min. (setting=1110) (Calibrate furnace temp. first)
    b. Dry N_2, 1100 °C, 28 min. (27 min. then 2 min. pulling out) (xj(N channel S/D)=1.654μm, Rsh=18.39 ohms/square) (xj(P channel S/D)=1.589μm, Rsh=40.9 ohms/square)

17. Pos. PR Step, Mask #4 : Thin Oxide (P channel and N channel Gates) mask.
    a. Apply photoresist, expose, develop (follow 3.a-i)
    b. Oxide etch, buffered HF (NH_4F:HF - 5:1) etch rate .115μm/min. (etch through ~12000 A oxide) (etch time=7.5 min.)
    c. PR strip
    d. Piranha clean, 15 min.

18. Gate Oxide Growth - 840A/P+(well), 824A/P+(substrate), 1622A/N+ (or 1100A/gate, 1200A/N+) P+ drive-in furnace
    a. TCE clean P+ drive-in furnace at least 12 hours before use
    b. Wet O_2, 840 °C, 75 min. (Calibrate temp. first, check water after 1.5 hour) (setting=840) (or Dry O_2, 1000 °C, 145 min., init. ox. furnace)
    c. Dry N_2 (anneal), 840 °C, 15 min. (or Dry N_2 (anneal),1000 °C, 20 min. for Dry O_2 growth)

19. Threshold shift implant - Boron, Dose=5x10^{11}, 55 keV (75 keV alt.)

20. Piranha clean, 5 min.

21. Threshold shift anneal - P+ drive-in furnace Dry N_2, 1000 °C, 15 min. (load and unload wafers in SLOWLY)

22. Pos. PR Step, Mask #5 : Contact cut mask.
    a. Apply photoresist, expose, develop (follow 3.a-i) (spin on at 5000 rpm) (check very carefully for gaps)

b. Descum - Short plasma etch of resist
   1. N_2, 1 torr, 60 watts, 65 °C
   2. O_2, .76 torr, 10 watts, 5 min.
c. Oxide etch, buffered *HF* (NH_4F:*HF* - 5:1) etch rate .115μm/min. (etch time=1.5 min.) (etch through ~1700A of oxide)
d. PR strip
e. Piranha clean, 15 min.

23. Metal deposition - 8000A Aluminum
    a. *HF* dip (*HF*:water - 1:20), 10 sec. (just until back of wafer repels water)
    b. Bake under IR lamp, 15 min.
    c. Use following settings: 25μA,5.2 remote,1 staple,~$5x10^{-6}$ Torr,80 sec.,25 toward

24. Pos. PR Step, Mask #6 : Metal mask.
    Apply photoresist, expose, develop (follow 3.a-i) (spin on at 6200 rpm) (VERY CRITICAL ALIGNMENT to N+)

25. Metal etch - Al etchant type A, 45-50 °C, (Do not overetch) (~3.5 min.)

26. PR strip, acetone, 10 min.

27. Rinse in DI water

28. Pos. PR Step, Mask #1 : Self align implant. Apply photoresist, expose, develop (follow 3.a-i) (spin on at 6000 rpm)

29. Implant (self aligned PMOS) Boron, Dose=$3.81x10^{13}$, 45 keV (dose=40 μcoulombs)

30. PR strip
    a. Acetone, 30 min. (1 min. ultrasonic every 15 min.)
    b. DI rinse and blow dry
    c. Plasma etch, 150 watts, 1 torr, 5 min. or until done

31. Sinter Aluminum - 450 °C, 15 min. with forming gas,14 cm.

32. Spin on PR, 15 min. prebake(100 °C), 5000 rpm, 30 sec., 20min. softbake(90°C)

33. Dice - dicing saw, speed=2(hi), 4.0 mils not sawed, channel 1 = 4.250,2.500, channel 2= 4.630,1.750, distance to edge of chip= .250 mm.

this page for number sequence only

APPENDIX E

EXACT SIMULATION OF LOOP GAIN

The loop gain of a feedback circuit provides valuable stability and settling time information [27]. Often, loop gain simulations are based on approximations [27],[42]. For example, for the shunt-shunt feedback circuit of figure E.1 the exact loop gain is given by

$$T(s)=\frac{(G_m-y_f)(y_f-y_r)}{(y_f+y_o)(y_f+y_i)}$$

(following the terminology in [27], see figure E.1). Output loading and the output admittance of the op amp are included in y_o; the source admittance y_s and op amp input admittance are included in y_i. Usually, y_r, the reverse transmission through the op amp, is much smaller in magnitude than y_f in the frequency range of interest and can safely be neglected. But often y_f is assumed much smaller in magnitude than G_m and the loop gain is approximated as

$$T(s)=\frac{G_m y_f}{(y_f+y_o)(y_f+y_i)}.$$

Neglecting y_f is not always valid. For example, if $y_f=sC_{fb}$, then y_f and G_m may have comparable magnitudes near the frequency where $|T(s)|=1$. In that case, neglecting y_f would yield an incorrect phase margin and possibly false stability data. Circuit design text books introduce the "break-the-loop" loop gain analysis which has a nice intuitive feel, but relies on the above assumptions as well as the assumption that $|y_o| \gg \left|\frac{y_f y_i}{y_f+y_i}\right|$ [27],[42]. The "break-the-loop" loop gain is given by the voltage transfer function $T(s)=-\frac{V_r}{V_s}$

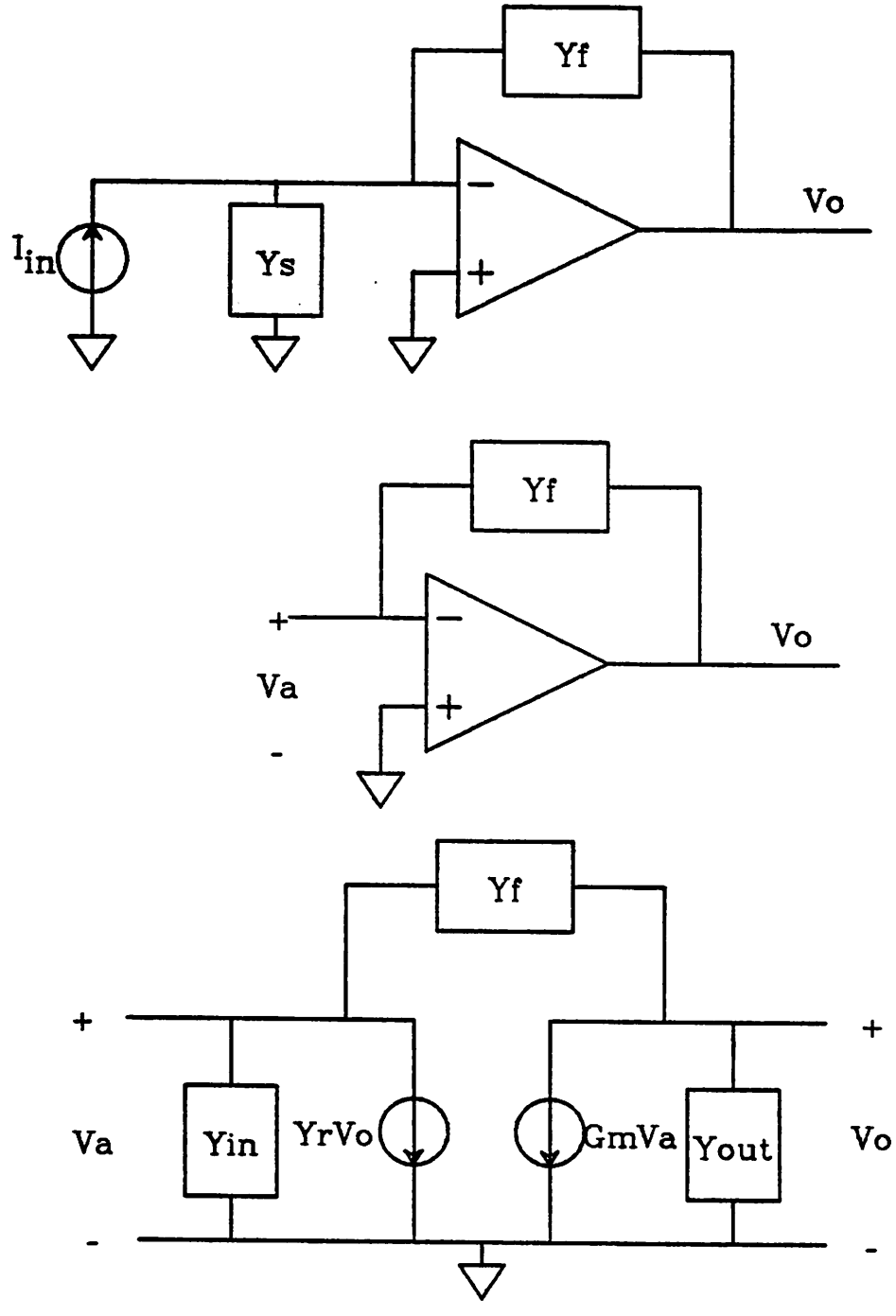


Figure E.1. top: Shunt-shunt feedback circuit
middle: Op amp with feedback element y_f
bottom: Same circuit with op amp replaced by its two-port representation

in figure E.2.

These approximations need not be made at all because direct, exact simulation of the loop gain is possible with circuit simulators like SPICE. For example, consider the shunt-shunt feedback circuit of figure E.1. If the input current source is replaced by a voltage source V_x as shown in figure E.3, the transfer function from V_x to V_o is

$$\frac{V_o}{V_x} = -\frac{(G_m - y_f)}{(y_f + y_o)}.$$

A dependent source, controlled by the output voltage, driving the output of the op amp in the reverse direction gives a voltage transfer relation

$$\frac{V_r}{V_o} = \frac{(y_f - y_r)}{(y_f + y_i)}.$$

And therefore the overall transfer function $-\frac{V_r}{V_x}$ is

$$-\frac{V_r}{V_x} = \frac{(G_m - y_f)\ (y_f - y_r)}{(y_f + y_o)\ (y_f + y_i)}$$

which equals the exact loop gain. No approximations were made.

It is often important, due to non-linear effects, to test the loop gain at different DC output voltages. These can easily be accommodated as is shown in figure E.4. The original circuit (figure E.4, top) is used to determine the DC operating points. Then controlled sources are connected as shown in figure E.4 to assure that all DC node voltages are correct. The loop gain is found by computing the small signal gain, $T(s) = -\frac{V_r}{V_x}$.

This exact loop gain analysis can be applied to all four of the feedback configurations. Here is the procedure for exact loop gain analysis for an arbitrary feedback configuration:

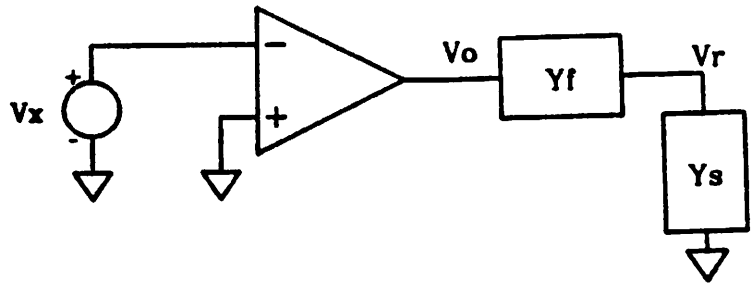


Figure E.2. Break-the-loop circuit for finding T(s)

Draw the circuit, labeling input and output sources correctly. The input source must be a current source if it is shunt feedback at the input, an input voltage source for series feedback at the input. Use Thevenin or Norton equivalences as required. Replace the input source with its dual (i → v, v → i), call the new source S_s. Now make a second copy of the circuit. Drive the output of the second circuit with a controlled source of the same type as the output variable, with value +1 times the output of the first circuit. In this circuit, replace the test source S_s with an infinite impedance mismatch (replace a voltage source with an open circuit, replace a test current source with a short circuit.) Label the location where S_s was as S_r with the same polarity. If S_s was a test voltage, S_r will measure an open-circuit return voltage. If S_s was a test current, S_r will measure a short-circuit return current. Have the circuit simulator calcu-

late the transfer functions $T(s)=-\frac{S_r}{S_s}$ which is the exact loop gain. An arbitrary DC input source can be accommodated as discussed above.

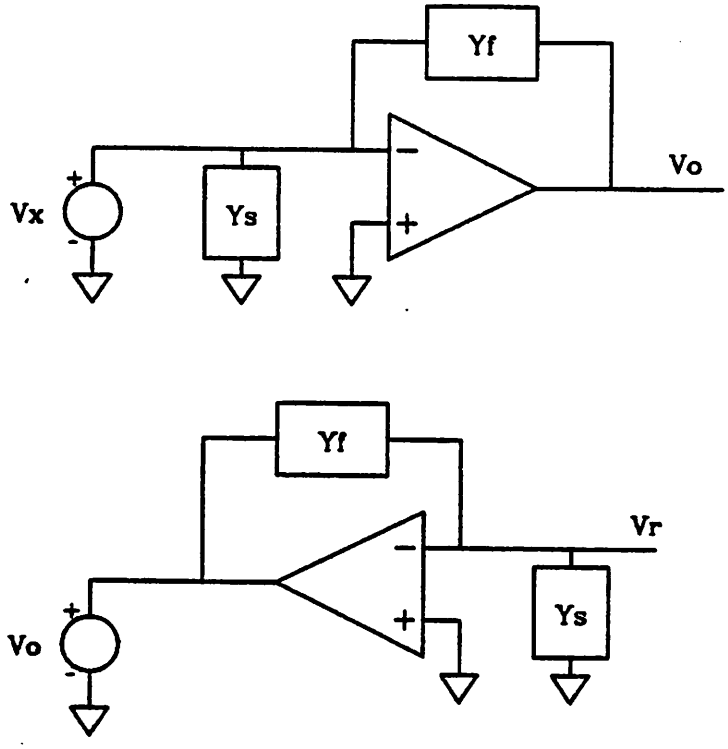


Figure E.3 Circuit for finding T(s) exactly

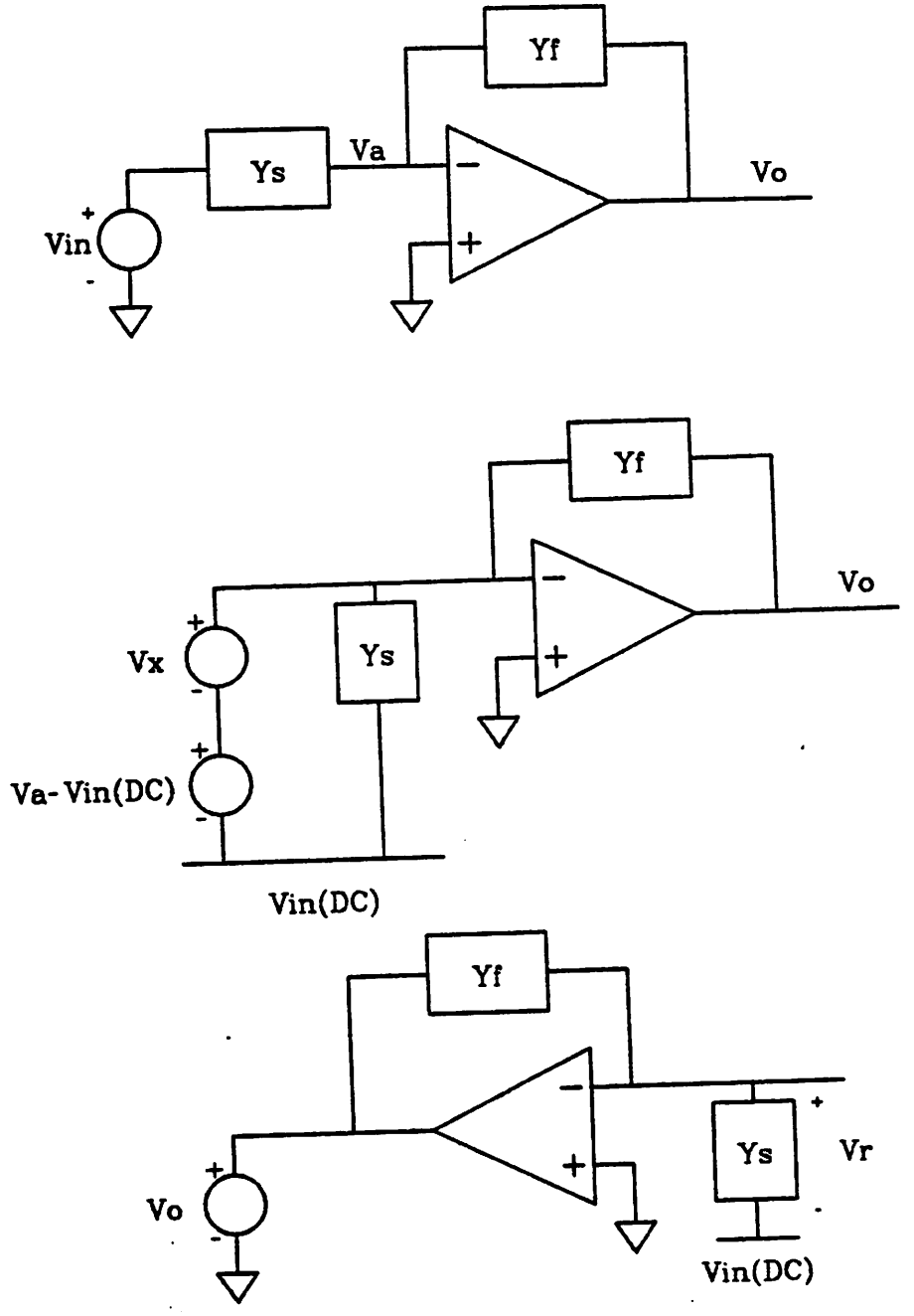


Figure E.4. Circuit for finding T(s) exactly, including DC biasing

BIBLIOGRAPHY

References

1. L.R. Rabiner and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Inc., Englewood Cliffs, N.J.(1978).

2. J.D. Markel and A.H. Gray, Jr., *Linear Prediction of Speech*, Springer-Verlag, New York, N.Y.(1980).

3. J.D. Markel and A.H. Gray, Jr., "A Linear Prediction Vocoder Simulation Based upon the Autocorrelation Method," *IEEE Trans. Acoust., Speech, and Signal Process.* **Vol. ASSP-22** pp. 124-134 (April 1974).

4. A.V. Oppenheim and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, Inc., Englewood Cliffs, N.J.(1975).

5. T. Barnwell, "Recursive Autocorrelation Computation for LPC Analysis," *Proc. Int'l. Conf. on Acoustics, Speech and Signal Processing*, pp. 1-4 (1977).

6. J. Durbin, "The Fitting of Time-Series Models," *Rev. Inst. Int. Statis.* **Vol. 28**(3) pp. 233-243 (1960).

7. N. Levinson, "The Wiener RMS Error Criterion in Filter Design and Prediction," *J. Math. Phys.* **Vol. 25**(4) pp. 261-278 (1947).

8. J. Makhoul, "Linear Prediction: A Tutorial Review," *Proc. IEEE* **Vol. 63** pp. 561-580 (April 1975).

9. J.L. Flanagan, "Automatic Extraction of Formant Frequencies from Continuous Speech," *J. Acoust. Soc. Am.* **Vol. 28** pp. 110-118 (January 1956).

10. H.K. Dunn and S.D. White, "Statistical Measurements on Conversational Speech," *J. Acoust. Soc. Am.* **Vol. 11** pp. 278-288 (January 1940).

11. F. Itakura, "Minimum Prediction Residual Principle Applied to Speech Recognition," *IEEE Trans. Acoustics, Speech and Signal Processing* **Vol. ASSP-23** pp. 67-72 (Feb. 1975).

12. J.D. Markel, "The SIFT Algorithm for Fundamental Frequency Estimation," *IEEE Trans. Audio Electroacoustics* **Vol. AU-20** pp. 367-377 (December 1972).

13. L.R. Rabiner and M.R. Sambur, "Application of an LPC Distance Measure to the Voiced-Unvoiced-Silence Detection Problem," *IEEE Trans. on Acoust., Speech, and Signal Process.* **Vol. ASSP-25**(4) pp. 338-343 (August 1977).

14. R.D. Fellman, *An MOS-LSI Adaptive Linear Prediction Filter For Speech Processing*, University of California, Berkeley(November 1982). Ph.D. Thesis

15. B. Gold Private communication,

16. J. Tierney, "A Study of LPC Analysis of Speech in Additive Noise," *IEEE Trans. Acoust., Speech, and Signal Process.* **Vol. ASSP-28**(4) pp. 389-397 (August 1980).

17. M.D. Paez and T.H. Glisson, "Minimum Mean Squared-Error Quantization in Speech," *IEEE Trans. Comm.* **Vol. Com-20** pp. 225-230 (April 1972).

18. S. Love, *A Preprocessor for Speech Analysis*, University of California, Berkeley, Ca(June 1981). M.S. Report

19. A.Vladimirescu and S. Liu, "The Simulation of MOS Integrated Circuits Using Spice2," Memorandum No. UCB/ERL M80/7, Electronics Research Laboratory, University of California, Berkeley, Ca (February 1980).

20. B. Gold and L. Rabiner, "Parallel Processing Techniques for Estimating Pitch Periods of Speech in the Time Domain," *J. Acoust. Soc. Am.* **Vol. 46** pp. 442-448 (August 1969).

21. A.H. Gray, Jr. and J.D. Markel, "Quantization and Bit Allocation in Speech Processing," *IEEE Trans. Acoust., Speech, and Signal Process.* **Vol. ASSP-24** pp. 459-473 (December 1976).

22. P. Chu Private communication,

23. A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill Book Company, San Francisco, Ca(1965).

24. R. Kaneshiro Private communication,

25. D.R. Alexander, R.J. Antinone, and Dr. G.W. Brown, "Spice2 MOS Modeling Handbook," Technical Report BDM/A-77-071-TR, The BDM Corporation, Albuquerque, NM (May 1977).

26. P.R. Gray, "Basic MOS Operational Amplifier Design - An Overview," pp. 28-49 in *Analog MOS Integrated Circuits*, ed. R.W. Brodersen,IEEE Press, New York, N.Y. (1980).

27. P.R. Gray and R.G. Meyer, *Analysis and Design of Analog Integrated Circuits*, John Wiley & Sons, Inc., New York, N.Y.(1977).

28. W.C. Black, Jr., *High Voltage Metal Gate CMOS Process for Large Scale Analog Circuit Applications*, University of California, Berkeley, CA(June 1977). M.S. Report

29. P.W. Bosshart, "A Multiplexed Switched Capacitor Filter Bank," *IEEE Journal of Solid-State Circuits* **Vol. SC-15**(6) pp. 939-945 (December 1980).

30. Engineering Staff of Analog Devices, Inc., *Analog-Digital Conversion Notes*, Analog Devices, Inc., Norwood, Mass.(1977).

31. R.H. McCharles, V.A. Saletore, W.C. Black, Jr., and D.A. Hodges, "An Algorithmic Analog-to-Digital Converter," *IEEE Int'l Solid-State Circuits Conf. Digest*, pp. 96-97 (1977).

32. J.L. McCreary and P.R. Gray, "All-MOS Charge Redistribution Analog-to-Digital Conversion Techniques- Part 1," *IEEE J. Solid State Circuits* **Vol. SC-10** pp. 371-379 (December 1975).

33. Ken Keller and Giles Billingsley, *KIC: A Graphics Editor for Integrated Circuits*, Electronics Research Laboratory, U.C. BerkeleyTechnical Memorandum

34. J.E. Marsden, *Basic Complex Analysis*, W.H. Freeman & Company, San Francisco, Ca(1973).

35. D.A. Hodges and H.G. Jackson, *Analysis and Design of Digital Integrated Circuits*, McGraw-Hill Book Company, San Francisco, Ca(1983).

36. E.J. Swanson, R.J. Starke, G.F. Cross, K.H. Olson, C.J. Waldron, R.A. Copeland, S.A. Surek, R.J. Ribble, and A.J. Vera, "A Fully Adaptive Transversal Canceler and Equalizer Chip," *IEEE Int'l. Solid-State Circuits Conf.*, pp. 20-21 (1983).

37. D.C. Soo and R.G. Meyer, "A Four-Quadrant NMOS Analog Multiplier," *IEEE Journal of Solid-State Circuits* **Vol. SC-17(6)** pp. 1174-1178 (December 1982).

38. B. Gilbert, "A Four-Quadrant Analog Divider/Multipler with 0.01% Distortion," *IEEE Int'l. Solid-State Circuits Conf.*, pp. 248-249 (1983).

39. P.R. Gray, D.A. Hodges, and R.W. Brodersen, *Analog MOS Integrated Circuits*, IEEE Press, New York, N.Y.(1980).

40. K.S. Tan and P.G. Gray, "Fully Integrated Analog Filters using Bipolar-JFET Technology," *IEEE Journal of Solid-State Circuits*, pp. 814-821 (1978).

41. M. Banu and Y. Tsividis, "Fully Integrated Active RC Filters in MOS Technology," *IEEE Int'l. Solid-State Circuits Conference*, pp. 244-245 (February 1983).

42. P.E. Gray and C.L. Searle, *Electronic Principles: Physics, Models, and Circuits*, John Wiley & Sons, Inc., New York, N.Y.(1969).