1

MICROD

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC, 1970, c. C-30, et ses amendements subséquents.

A VLSI Architecture for Linear Predictive Analysis of Speech

by

John Willis O'Leary, B.Sc.E.E.

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Engineering**

**Ottawa–Carleton Institute for Electrical Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario**

April 1989

©Copyright

1989, John Willis O'Leary

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-51180-X

**The undersigned recommend to the Faculty of Graduate studies
and Research acceptance of the thesis**

**“A VLSI Architecture for Linear
Predictive Analysis of Speech”**

**submitted by John Willis O'Leary, B.Sc.E.E.
in partial fulfilment of the requirements for
the degree of Master of Engineering**

Thesis Supervisor

Thesis Supervisor

Chair, Department of Systems and Computer Engineering

Carleton University

April 1989

Abstract

An architecture was developed to perform linear predictive analysis of speech using the autocorrelation method. Recursive windowing and the LeRoux–Gueguen algorithm were identified as techniques that will lead to an efficient hardware realization. The architecture was simulated at the system level and the spectral distance between the simulation results and the results generated by a floating point software implementation was 3.43 dB over 50 frames. Parts of the coder were implemented on silicon to demonstrate feasibility. The total complexity of the system is estimated to be less than 45,000 transistors, a size that can easily be realized in gate array or custom VLSI. The speech analysis system is suitable for incorporation into a speech coder for a digital mobile radio transceiver.

Acknowledgements

I would like to thank my thesis supervisors, Professor D. D. Falconer and Professor C. H. Chan, for their excellent guidance throughout the course of this research.

I would particularly like to thank Margaret Cooper for providing moral support, intellectual stimulation, and hot meals when they were most needed.

Table of Contents

Abstract	iii
Acknowledgements	iv
Table of Contents	v
List of Tables	viii
List of Illustrations	ix
List of Appendices	xii
1 Introduction and Motivation	1-1
1.1 Speech Coding for Land Mobile Radio	1-1
1.2 Objectives	1-3
1.3 Organization	1-5
2 Algorithms and Architectures for Speech Analysis	2-1
2.1 The Autocorrelation Method for Linear Predictive Analysis	2-1
2.1.1 <i>Recursive Autocorrelation Estimation</i>	2-4
2.1.2 <i>The LeRoux-Gueguen Algorithm</i>	2-6
2.1.3 <i>Summary of Algorithms</i>	2-10
2.2 Bit-Serial Architectures, Functional Design and a Hardware Design Strategy for VLSI	2-15
2.3 Design Tools	2-16
3 System Design and Simulation	3-1

3.1	Numerical Considerations	3-1
3.1.1	<i>Input Data Format, Analysis Order and Frame Size</i>	3-2
3.1.2	<i>Choice of System Wordlength</i>	3-2
3.1.3	<i>Numeric Overflow in the Input Correlator</i>	3-3
3.1.4	<i>Numeric Overflow in the Recursive Filter Sections</i>	3-3
3.1.5	<i>Numeric Overflow in the Non-Recursive Filter Section</i>	3-5
3.1.6	<i>Numeric Overflow in the LeRoux-Gueguen Algorithm</i>	3-9
3.2	A Set of Bit-Serial Simulation Elements	3-9
3.2.1	<i>Pseudo 2-Phase Clocking Scheme</i>	3-9
3.2.2	<i>Description of Elements</i>	3-9
3.3	System Organization	3-13
3.3.1	<i>Inter-Block Communication</i>	3-13
3.3.2	<i>Major Subsystems</i>	3-15
3.4	Circuit Design in Terms of Bit-Serial Primitives	3-17
3.4.1	<i>Input Register</i>	3-17
3.4.2	<i>Autocorrelation Block</i>	3-17
3.4.3	<i>LeRoux-Gueguen Algorithm</i>	3-22
3.4.4	<i>Control Circuitry</i>	3-25
3.5	System Simulation	3-31
3.5.1	<i>Algorithm Verification</i>	3-31
3.5.2	<i>Hardware Simulation</i>	3-32
3.6	Summary	3-33
4	Implementation of Test Circuits	4-1
4.1	Choice of Circuits for Trial Realization	4-1
4.2	Layout Notes	4-2

	vii
4.3 Input Correlator Block	4-2
4.3.1 <i>Clock Buffer</i>	4-3
4.3.2 <i>Serial/Parallel Multiplier</i>	4-9
4.3.3 <i>Delay Line</i>	4-14
4.3.4 <i>Control Circuitry</i>	4-14
4.3.5 <i>Multiplexer and Delay Compensation Circuitry</i>	4-14
4.4 Recursive Filtering Block	4-21
4.4.1 <i>Serial Fixed-Coefficient Multiplier</i>	4-21
4.4.2 <i>Subtractor</i>	4-26
4.4.3 <i>Delay Compensation Circuitry</i>	4-26
4.5 Assembly of Test Chip	4-26
4.6 System Complexity Estimate	4-33
5 Conclusions and Recommendations	5-1
5.1 Thesis Summary	5-1
5.2 Suggestions for Further Research	5-4
Serial Arithmetic Components for SPLICE Simulator	A-1
SPLICE Netlist for Simulation of Recursive Autocorrelator	B-1
SPLICE Netlist for Simulation of LeRoux-Gueguen Block	C-1
References	R-1

List of Tables

2-1 Comparison of Lattice and LeRoux-Gueguen Methods 2-14

3-1 Nonrecursive Filter Coefficients 3-8

4-1 Summary of Simulation Results for Clock Buffer 4-8

4-2 System Parts List and Complexity Estimate 4-34

List of Illustrations

1-1	REL _P Speech Coder – Block Diagram	1-2
1-2	RPE-LPC Coder Block Diagram	1-4
2-1	Digital Model for Speech Production	2-2
2-2	Linear Predictive Analysis System	2-5
2-3	Recursive Autocorrelator Block Diagram	2-7
2-4	Signal Flowgraph for LeRoux-Gueguen Algorithm	2-9
2-5	Adaptive Lattice Method	2-11
2-6	Block-by-block (LeRoux-Gueguen) Method	2-13
3-1	Recursive Filter Section	3-4
3-2	Nonrecursive Filter Section	3-6
3-3	Pseudo 2-Phase Clocking Scheme	3-10
3-4	Major Elements in the Simulation Library	3-11
3-5	Inter-Block Communication Signals	3-14
3-6	Overall Organization of System	3-16
3-7	Input Register Implementation	3-18
3-8	Input Correlator Implementation	3-20
3-9	Signal Scaling Block Implementation	3-21
3-10	Recursive Filter Implementation	3-23
3-11	Nonrecursive Filter Implementation	3-24
3-12	Coefficient Calculation Block	3-26, 3-27

3-13 Lattice Block	3-28
3-14 FSM Activated by Sample Clock	3-29
3-15 FSM Activated by Frame Clock	3-30
4-1 Clock Buffer Schematic	4-4
4-2 Clock Buffer Layout	4-5
4-3 FACTS Simulation Results for Clock Buffer	4-6
4-4 SPICE Simulation Results for Clock Buffer	4-7
4-5 Serial/Parallel Multiplier	4-10, 4-11
4-6 Layout of Serial/Parallel Subtracting Stage	4-12
4-7 Layout of Serial/Parallel Multiplier	4-13
4-8 Delay Line Element	4-15
4-9 Layout of 9 × 32-bit Delay Line	4-16
4-10 Correlator Control Circuit Schematic	4-17
4-11 Correlator Control Circuit Layout	4-18
4-12 Multiplexer and Delay Compensation Schematic	4-19
4-13 Multiplexer and Delay Compensation Layout	4-20
4-14 Fixed Coefficient Multiplier	4-22, 4-23
4-15 Layout of Fixed Coefficient Subtracting Stage	4-24
4-16 Layout of Fixed Coefficient Multiplier	4-25
4-17 Subtractor Schematic	4-27
4-18 Subtractor Layout	4-28
4-19 Delay Compensation Schematic	4-29
4-20 Delay Compensation Layout	4-30

4-21 Chip Floorplan	4-31
4-22 Chip Plot	4-32

List of Appendices

A	Serial Arithmetic Components for SPLICE Simulator	A-1
B	SPLICE Netlist for Simulation of Recursive Autocorrelator	B-1
C	SPLICE Netlist for Simulation of LeRoux-Gueguen Block	C-1

Chapter 1

Introduction and Motivation

Recently, there has been much interest in digital transmission of speech over the land mobile channel. It is perceived that digital systems provide a cost advantage over analog systems, and are less susceptible to the effects of multipath propagation on the mobile radio channel. The need for secure communications is also an important factor; encryption can be performed more easily on digital data. As an additional benefit, digital systems will allow for the integration of voice and data signals, and possibly provide the capability to connect to the Integrated Services Digital Network (ISDN). The digital speech must be encoded, however, because of the limited bandwidth of the mobile channel.

1.1 Speech Coding for Land Mobile Radio

The requirements for speech coding techniques are determined chiefly by the limitations of the land mobile channel. Previous research carried out by Robert Hoyle at Carleton University identified the following requirements for a speech coding technique [11].

- Firstly, the channel spacing is specified as 30 kHz in the 150 MHz band and 25 kHz in the 450 MHz and 850 MHz bands. To transmit digitally encoded speech over these channels, it is desirable that the output data rate of the coder be 16 kbits/sec or less.
- Secondly, a mobile station will receive a rapidly fading signal due to multipath propagation. Electromagnetic emissions from various sources may also degrade the quality of the signal that is received. Therefore the coding technique must be able to transmit toll-quality speech over a noisy channel.
- Finally, the speech coding algorithms should be as simple as possible to allow for VLSI implementation.

(a) RELP coder

(b) RELP decoder

Figure 1-1: Basic RELP Speech Coder – Block Diagram

The conclusion of Hoyle's research was that the residually excited linear prediction (RELP) coder best fits the requirements for a speech coder for mobile radio. The RELP coder was initially devised by Un and Magill [32]. The block diagram of a basic RELP coder and decoder is shown in Figure 1-1. In the coder, linear predictive analysis is performed on the input speech signal $s(n)$ to obtain the LPC coefficients. These coefficients are used to generate a predicted speech signal $\hat{s}(n)$ which is subtracted from the input speech signal to produce the prediction residual. This residual is low-pass filtered, decimated and quantized; the result is multiplexed with the encoded LPC coefficients and transmitted over the digital channel. In the decoder, the decimated prediction residual is interpolated and filtered to recover the full-band prediction residual. The prediction residual is then used as excitation to a synthesis filter; the result is the decoded digital speech signal.

Since this research was carried out, a variation of the Regular-Pulse Excited LPC (RPE-LPC) coder was selected for the European digital mobile radio system [23]. Unlike the RELP coder, the RPE-LPC coder models the prediction residual by a low bit rate excitation sequence and does not transmit a decimated version of the prediction residual. Referring to Figure 1-2 (a), the residual $r(n)$ is generated by passing $s(n)$ through a whitening filter. The difference between the prediction residual and the model residual $v(n)$ is passed through an error weighting filter. The weighted difference $e(n)$ is used to update the parameters of the excitation model. The LPC coefficients and parameters of the excitation model are transmitted over the channel. Decoding is a simple matter of separating the LPC coefficients from the parameters of the excitation model, and using the model residual $v(n)$ to drive the synthesis filter.

Linear predictive analysis of speech is an essential element of the RELP coder, the RPE-LPC coder, and other promising coders such as the adaptive predictive coder. This thesis will concentrate on this crucial element of the RELP and RPE-LPC coders, with special emphasis on hardware implementation.

1.2 Objectives

The primary objective of this thesis is to develop an architecture for linear predictive

(a) RPE-LPC Coder

(b) RPE-LPC Decoder

Figure 1-2: RPE-LPC Coder Block Diagram

analysis of speech that is suitable for incorporation into a speech coder for a digital mobile radio transceiver. This is accomplished in the following way:

- 1) The autocorrelation method of linear predictive analysis is examined in detail, with a strong bias towards hardware implementation. Some techniques that will lead to an efficient hardware realization are identified.
- 2) The linear predictive analysis system is partitioned into subsystems, and various design choices are made that will affect the performance of the analysis algorithm. The design is "fleshed out" to the point that implementation can begin.
- 3) The most compute-intensive elements of the system are chosen for trial implementation; they are laid out and fabricated by Northern Telecom Electronics through the Canadian Microelectronics Corporation.
- 4) The feasibility of the linear predictive analysis system for hardware implementation can now be assessed. As well, number of recommendations can be made based on this experience.

1.3 Organization

Chapter two contains a review of the autocorrelation method of linear predictive analysis, and explains several techniques which can make the autocorrelation method more suitable for hardware implementation. A functional design methodology is mentioned briefly and a general strategy for the design is outlined. The design tools that were available at Carleton at the time are described.

Chapter three begins by examining some important system design issues such as system wordlength, analysis window length, overflow prevention, inter-block communication and clocking. A partitioning of the system into major sub-blocks is presented. Each of these major sub-blocks is then examined in detail. Finally, the results of the system-level simulation are presented.

Chapter four marks the transition from system design to implementation. First, the most compute-intensive elements of the system are chosen for trial implementation. A number

of useful circuit primitives are synthesized and assembled into test circuits. These circuits were submitted to the Canadian Microelectronics Corporation for eventual fabrication by Northern Telecom Electronics.

In chapter five the research is summarized and recommendations are made.

Chapter 2

Algorithms and Architectures for Speech Analysis

Section 2.1 formulates the autocorrelation method of linear predictive analysis and outlines the major subunits of a speech analysis system. Subsections 2.1.1, and 2.1.2 describe several modifications to the autocorrelation method which are particularly suitable for VLSI implementation. Subsection 2.1.3 provides a summary of the algorithms and assesses their suitability for hardware implementation. Section 2.2 introduces bit-serial architectures for VLSI, and a functional design methodology. Section 2.3 discusses the hardware design strategy that will be followed. Section 2.4 summarizes the design tools that were available, and where they fit into the design strategy.

2.1 The Autocorrelation Method for Linear Predictive Analysis

A digital model for the production of speech signals is shown in Figure 2-1. The digital filter approximates the properties of the vocal tract. For voiced speech the filter is excited by an impulse train in which the spacing between impulses varies according to pitch. For unvoiced speech the filter is excited by a random number generator which produces white noise [29].

The object of linear predictive analysis is to estimate the parameters $a_1, \dots, a_p$ of the digital filter which represents the vocal tract.

The characteristics of the vocal tract change relatively slowly during speech and it is a reasonable assumption that the digital filter of Figure 2-1 has fixed characteristics over a short (10ms – 20 ms) time interval. The first step in speech analysis is to divide the speech waveform into analysis intervals called frames. The parameters $a_1, \dots, a_p$ will then

Figure 2-1: Digital Model for Speech Production

be estimated once per analysis frame. Therefore, the speech signal $s(n)$ is first multiplied by a data window to form a new time waveform

$$s_w(n, j) = s(n)w(n - j) \quad (2-1)$$

where j accounts for the position of the window.

We shall estimate the values of the vocal tract parameters $a_1, \dots, a_p$ by $\alpha_1, \dots, \alpha_p$, such that the mean-squared error between the current sample $s_w(n, j)$ and the predicted sample $\hat{s}_w(n, j)$ is minimized:

$$E = \sum_{n=-\infty}^{\infty} [s_w(n, j) - \hat{s}_w(n, j)]^2 = \sum_{n=-\infty}^{\infty} [s_w(n, j) - \sum_{i=1}^p \alpha_i s_w(n - i, j)]^2 \quad (2-2)$$

which expands to

$$E = \sum_{n=-\infty}^{\infty} \{ s_w^2(n, j) - 2s_w(n, j) \sum_{i=1}^p \alpha_i s_w(n - i, j) + [\sum_{i=1}^p \alpha_i s_w(n - i, j)]^2 \} \quad (2-2a)$$

We can obtain $\alpha_1, \dots, \alpha_p$ by setting $\frac{\partial E}{\partial \alpha_k}$ to zero for each α_k :

$$\frac{\partial E}{\partial \alpha_k} = \sum_{n=-\infty}^{\infty} \{ -2s_w(n, j)s_w(n - k, j) + 2s_w(n - k, j) \sum_{i=1}^p \alpha_i s_w(n - i, j) \} = 0, \quad 1 \leq k \leq p \quad (2-3)$$

which can be rewritten as

$$R(k, j) = \sum_{i=1}^p \alpha_i R(k - i, j), \quad 1 \leq k \leq p \quad (2-3a)$$

where

$$R(i, j) = \sum_{n=-\infty}^{\infty} s_w(n, j)s_w(n - i, j) \quad (2-4)$$

is the autocorrelation of the windowed speech signal $s_w(n, j)$. This system of equations can then be solved to yield $\alpha_1, \dots, \alpha_p$.

The system of equations in (2-3a) has the useful property of being symmetric and Toeplitz. That is, the elements along any diagonal of the matrix are identical. There are efficient techniques for the solution of such systems such as the Levinson-Durbin method and the LeRoux-Gueguen method [19]. The LeRoux-Gueguen method offers some definite

advantages for hardware implementation. These will be discussed further in Subsection 2.1.2.

A block diagram of a linear predictive analysis system appears in Figure 2-2. Subsections 2.1.1, 2.1.2 and 2.1.3 discuss efficient methods for implementing the various blocks.

2.1.1 Recursive Autocorrelation Estimation

The most computationally intensive elements of the system outlined in Figure 2-2 are the data windowing and calculation of the autocorrelation coefficients. The data windowing in particular requires storage of many windowing coefficients in a lookup table and the buffering of entire frames of data. It is possible, however, to eliminate this problem by filtering the autocorrelation coefficients rather than windowing the data.

In equation (2-4) we had:

$$R(i, j) = \sum_{n=-\infty}^{\infty} s_w(n, j) s_w(n - i, j)$$

Since $s_w(n, j) = s(n)w(n - j)$ equation (2-4) can be rewritten as:

$$R(i, j) = \sum_{n=-\infty}^{\infty} s(n)w(n - j)s(n - i)w(n - i - j) \quad (2 - 4a)$$

or

$$R(i, j) = \sum_{n=-\infty}^{\infty} s(n)s(n - i)w(-(j - n))w(-(j - n) - i) \quad (2 - 4b)$$

Now, by defining $W(n, i) = w(n)w(n - i)$ and $S(n, i) = s(n)s(n - i)$ equation (2-4b) can be rewritten as:

$$R(i, j) = \sum_{n=-\infty}^{\infty} S(n, i)W(j - n, i) \quad (2 - 4c)$$

We can see from this equation that the autocorrelation $R(i, j)$ is the convolution of $s(n)s(n - i)$ with $w(-n)w(-n - i)$. Therefore if we produce $s(n)s(n - i)$ and pass it through a filter with impulse response $h_i(n) = w(-n)w(-n - i)$ the filter's output at time j will be $R(i, j)$.

Barnwell shows that a recursive window can be used for this purpose [2]. In particular, he demonstrates that the window

$$w(n) = (1 - \beta)\beta^{-n} \quad -\infty < n \leq 0$$

Figure 2-2: Linear Predictive Analysis System

$$w(n) = 0 \quad 0 < n < \infty$$

with $\beta = 0.98$ yields performance comparable to a 240-point Hamming window. This gives us the impulse response for the filter:

$$h_k(n) = (1 + n)\beta^n(1 + n + k)\beta^{n+k} \quad (2-5)$$

Taking the z -transform yields

$$H_k(z) = \frac{(k+1)\beta^k + (1-k)\beta^{k+2}z^{-1}}{(1 - \beta^2 z^{-1})^3} \quad (2-6)$$

Figure 2-3 (a) shows the structure of the recursive autocorrelator, and figure 2-3 (b) is a close-up of the k 'th filter as described by equation (2-6), broken into cascaded first-order sections. There are several important points which make this structure ideal for VLSI implementation:

- It operates identically on every sample, eliminating the requirement for buffering of entire analysis frames.
- The two multiplications in the non-recursive section of the filter need only be done once per analysis frame, not once per sample.
- The structure is extremely modular and easily partitioned. Furthermore, the three recursive sections are entirely identical.
- All of the windowing information is contained in β so there is no need for extensive ROM storage to support the windowing function.

Hurst and Broderson report an attempt to realize such an autocorrelator [12].

2.1.2 The LeRoux-Gueguen Algorithm

Roundoff noise, parameter sensitivity and stability are important issues in digital filtering. The lattice filter has several features related to these concerns that make it most suitable for fixed-point (hardware) implementations. In a RELP or RPE-LPC coder, the lattice filter would likely be used for the whitening filter shown in Figure 1-1 because of its superior numerical properties.

(a) recursive autocorrelator structure

(b) block diagram of k'th window filter

Figure 2-3: Recursive Autocorrelator Block Diagram

Friedlander's paper [5] points out these advantages of the lattice structure which make it suitable for use in hardware implementations:

- The transfer function of a lattice filter is determined by the values of the coefficients K_i . The output noise of lattice filters is independent of the values of these coefficients. That is, the effects of roundoff noise and truncation noise in the filter do not change with changes in the filter's transfer function. The result is better-quality speech at the output of the decoder.
- The transfer function of the filter is relatively insensitive to perturbations in the parameters K_i . This means that a shorter word length can be used for the parameters, yielding some hardware savings.
- The stability of a lattice filter is guaranteed when $|K_i| < 1$. This makes it possible to guarantee filter stability in finite wordlength implementations.

The LeRoux–Gueguen algorithm calculates the reflection coefficients $K_1, \dots, K_p$ directly from the autocorrelation coefficients $R(0), \dots, R(p)$ [17]. The basic algorithm is as follows:

```

DO 10 H = 1, P
    EP(H) = R(H+1)
    EN(H) = R(H)
10    CONTINUE

DO 20 H = 1, P
    K(H) = -( EP(H) / EN(1) )
    DO 30 I = 1, P-H+1
        TEMP      = EP(I+H-1) + EN(I) * K(H)
        EN(I)      = EN(I)      + EP(I) * K(H)
        EP(I+H-1) = TEMP
    30    CONTINUE
20    CONTINUE

```

Figure 2-4 shows the signal flowgraph of the L-G algorithm (from [25]).

Figure 2-4: Signal Flowgraph for LeRoux-Gueguen Algorithm

2.1.3 Summary of Algorithms

When we try to assess the suitability of our chosen algorithms, it is essential that we keep our goal firmly in mind. The goal of linear predictive analysis in the context of RELP and RPE-LPC coding is to obtain the reflection coefficients $K_1, \dots, K_p$ and the *whitened prediction residual* for further processing. In the early stages of the research, several methods for speech analysis were examined with the requirement that they be suitable for fixed-point hardware implementation. In the following paragraphs the length of the analysis frame will be denoted by N , and the analysis order by p .

The first method that was examined was the adaptive lattice method [20]. A signal flow graph for this method and an implementation of the correlator are shown in Figure 2-5. The low-pass filters in the correlator are essential because each reflection coefficient will only be transmitted once per frame, so an average of the recent values is required. The transfer function of the filters used for averaging is

$$W(z) = \frac{1}{(1 - \beta z^{-1})^3}, \quad 0 < \beta < 1$$

Now, it is clear from Figure 2-5 that each stage of the lattice analysis filter requires two multiplications and two additions. Furthermore, the correlator requires two multiplications and four additions plus an additional six multiplications and six additions for the averaging operation. These operations must occur in all p stages of the lattice for each of N samples in a frame, so the total number of operations per frame is $10pN$ multiplications and $12pN$ additions. A new set of reflection coefficients is calculated once per frame, so p division operations are required as well. Note that this assumes that it is permissible to use the reflection coefficients of the previous frame to compute the prediction residual for the current frame. If such an action is not permissible, there are two alternatives: update the reflection coefficient once per sample (then pN divisions will be required per frame) or buffer each frame until its reflection coefficients are computed and pass it through an additional lattice filter (a buffer large enough to hold a frame of speech is required, plus $2pN$ additions and $2pN$ multiplications per frame for the lattice analysis filter). For purposes of comparison, we assume that a suitable prediction residual can be computed using the

(a) adaptive lattice method

(b) correlator for adaptive lattice

Figure 2-5: Adaptive Lattice Method

reflection coefficients from the previous frame – without this assumption the cost of implementing the adaptive lattice method rises considerably. The delay between input speech and output prediction residual is essentially zero.

The second technique that was examined is the method that was eventually adopted: block-by-block calculation of the reflection coefficients *via* the LeRoux–Gueguen algorithm, followed by lattice filtering of the input speech signal to obtain the prediction residual. This is shown in Figure 2–6. For each sample, the recursive windowing operation requires $p + 1$ multiplications to calculate the correlation lags $s(n)s(n - k)$, as well as $p + 1$ multiplications and $p + 1$ additions in each of the three recursive filtering stages. These operations occur for each of N samples in a frame. Once per frame, the nonrecursive filtering operation must perform $2(p + 1)$ multiplications and $p + 1$ additions, and the LeRoux–Gueguen algorithm must perform p^2 multiplications, p^2 additions and p divisions. The prediction residual is generated at a cost of $2pN$ multiplications and $2pN$ additions per frame. The delay between input speech and the output prediction residual is approximately one frame ($10ms - 20ms$).

The numbers above are summarized in Table 2–1, along with figures for typical values of analysis frame size N and analysis order p .

The bottom line is that the Leroux–Gueguen method has significant computational advantages over the adaptive lattice method. The LeRoux–Gueguen method requires a data buffer large enough to accomodate somewhat more than one frame of data (see Figure 2–5) and has a somewhat more complex control structure. Note, however, that if we make the same assumption that we made for the adaptive lattice method (namely, that it is permissible to generate this frame's prediction residual using last frame's reflection coefficients) the data buffer can be eliminated. Flexibility is increased with the LeRoux–Gueguen method: we could opt for a mixed ASIC/off-the-shelf implementation, with the recursive autocorrelation and extraction of the prediction residual performed by custom LSI and the computation of reflection coefficients performed as an interrupt service by a general purpose microprocessor or a DSP microprocessor. Taking these factors into consideration, block-by-block analysis *via* the LeRoux–Gueguen method is the best choice.

Figure 2-6: Block-by-block (LeRoux-Gueguen) Method

		ADAPTIVE LATTICE	BLOCK-BY-BLOCK LEROUX-GUEGUEN
COMPUTE $R(0), \dots, R(p)$	+	N/A	$3N(p+1) + p + 1$
	×	N/A	$4N(p+1) + 2(p+1)$
	+	N/A	0
COMPUTE $K_1, \dots, K_p$	+	$10pN$	p^2
	×	$8pN$	p^2
	+	p	p
OBTAIN PREDICTION RESIDUAL	+	$2pN$	$2pN$
	×	$2pN$	$2pN$
	+	0	0
TOTAL	+	$12pN$	$p^2 + 5pN + 3N + p + 1$
	×	$10pN$	$p^2 + 6pN + 4N + 2p + 2$
	+	p	p
EXAMPLE $N = 120$ $P = 9$	+	12,960	5,851
	×	10,800	6,579
	+	9	9
DELAY		0 (see section 2.1.3)	1 frame (see section 2.1.3)

Table 2-1: Comparison of Lattice and LeRoux-Gueguen Methods

2.2 Bit–Serial Architectures, Functional Design and a Hardware Design Strategy for VLSI

Bit–serial computation architectures are distinguished by a communications strategy which features data transmission over single wires, bit–sequentially. Conventional bit–parallel architectures see the transmission of a word at a time over parallel buses. In many cases bit–serial architectures offer area–efficient VLSI implementations of signal processing functions. This is particularly true at the moderate throughputs required for a speech analysis application. Moreover, a serial system is often composed of many fine–grained components making it easy to partition the system across chips. Pins are precious on VLSI chips; serial architectures reduce on–chip and inter–chip routing requirements.

A functional design methodology for serial signal processing systems has been proposed and studied [3]. Functional design is based on the idea that systems will be built out of operators of a common generic type. The principle is to find an algorithm that enables the designer to reduce the required system function into an interconnection of lower–level functions, and ultimately to instances of primitive functional cells.

In the context of digital signal processing, the signal flowgraph is a natural decomposition of a processing function. The primitive functions are arithmetic operations, unit delays, constant generators, and so on. An advantage of this method of decomposition is that algorithm and architecture design and hardware design are largely separated. The design of the algorithm and architecture to realize a DSP function can be carried out at a data–flow level in terms of primitive functional cells. The algorithm and architecture can then be simulated to verify that design targets are met. Implementing the system as an ASIC is then a relatively simple problem of logic design.

The following are the main steps of the hardware design strategy that will be followed in chapters 3 and 4 of this thesis. The progression from algorithm design to architecture design to hardware design is clear.

- 1) Implement the algorithms using a sequential programming language and floating point arithmetic. This will give us a “baseline” against which to compare

the results of later stages of design.

- 2) Partition the system into blocks that seem to belong together naturally. A number of system-level design decisions must be made at this stage. Determine the mechanisms by which control and data is communicated between these blocks. This is the end of algorithm design and the beginning of architecture design.
- 3) Determine the set of bit-serial primitives that are required to implement the system. Implement these primitives as user defined circuit elements in the SPLICE simulator. This forms the basis for simulations at the data-flow level.
- 4) Merge and multiplex functional units to obtain area savings while satisfying throughput requirements. Design and simulate the system in terms of the primitive elements. The architecture design is now complete.
- 5) Choose several of the blocks for trial fabrication. Determine what primitives must be provided at the layout cell level. Design and implement the hardware primitives such that they fulfil the interface behaviour assumed in step 3.
- 6) Layout the primitives and assemble them into the test circuits.

Chapter 3 covers the first four steps. Those remaining are dealt with in Chapter 4.

2.3 Design Tools

The CAD tools that are available affect the partitioning of the system and the meeting point between high-level and low-level design efforts. This section gives a short summary of what tools were available at Carleton at the time.

SPLICE is a mixed-mode circuit simulation program developed at the University of California at Berkeley [26]. Carleton University has a locally augmented version of SPLICE which offers the user the capability of defining additional electrical and logical models [8]. Models are defined in RATFOR [15] and compiled and linked with the simulator. These user defined elements can then be used in a circuit netlist.

The VIVID design system is an integrated set of tools for the layout and simulation

of VLSI circuits, offered by the Microelectronics Center of North Carolina. Its primary elements are:

- ICE – a symbolic layout system,
- HCOMPACT – a hierarchical compactor which converts symbolic layouts generated with ICE into mask layers,
- FACTS – a fast circuit level simulator similar to SPICE.

Some other tools are available in addition to VIVID and SPLICE. Most notably, several mask editors were available (KIC, VALE) as well as several circuit and switch-level simulators (SPICE, Hg).

With this in mind, here is how the design tools fit into the design process.

- 1) SPLICE will be used to simulate the system in terms of bit-serial primitive elements. The primitives will be user defined logical models specified in RAT-FOR.
- 2) The primitives required to implement the test circuits will be laid out using VIVID's ICE symbolic layout editor.
- 3) VIVID's FACTS simulator will be used to verify that the primitives operate correctly. This is done by choosing a suitable set of test inputs and ensuring that the FACTS outputs and SPLICE outputs for this element agree.
- 4) SPICE will be used as a "sanity check" for FACTS. Since FACTS works from a symbolic layout and not a mask layout its estimations of circuit node capacitances may be inaccurate. Post-layout simulation can be done on some of the primitives by converting the mask description from VIVID's LLAMA format into Metheus's PHL format, running PHLEX on the Metheus workstation to extract a SPICE netlist and then running SPICE.
- 5) Positioning the blocks on the pad frame and wiring them together will be done using either KIC on the Sun workstations or VALE on the Metheus workstation.

Chapter 3

System Design and Simulation

This chapter addresses the high-level design of the speech analysis system. It is intended to proceed up to the point at which implementation on silicon can begin.

In section 3.1, we consider the numerical accuracy required, and cite the results of previous research to define the input data format, analysis order, sampling rate, frame length, and other critical parameters. Numerical overflow in the various parts of the algorithms is examined, and appropriate measures are taken to prevent it.

A set of bit-serial simulation elements is introduced in section 3.2. Clocking policy is determined.

Section 3.3 discusses the strategy for communication between major subsystems, and gives an overview of the system and its partitioning. The design of the major subsystems in terms of the bit-serial primitives is covered in section 3.4.

Finally, the results of the system-level simulation are presented in section 3.5.

3.1 Numerical Considerations

In a speech decoder, the spectral accuracy and therefore the quality of the restored speech depends on the accuracy with which the received LPC parameters characterize the speech spectrum. The following are the most important design issues that affect the fidelity of the LPC parameters.

- Quantization of the sampled speech at the input of the LPC coder.
- Analysis order p .
- Analysis frame length.
- Rounding and truncation in finite-word-length implementations.

- Overflow or underflow of intermediate quantities during processing. Numerical overflow in any stage of processing yields undesirable results. This is an important issue to contend with. We shall consider each of the major subblocks in turn.

These issues shall be addressed in the next six subsections.

3.1.1 Input Data Format, Analysis Order and Frame Size

The input speech is quantized to 12 bits and is in two's complement format. We shall consider the scale to be such that $-1 \leq s(n) < 1$.

Previous work by Viswanathan, *et al* and Hoyle indicates that an analysis order p of 9, a speech sampling rate of 6.67 kHz and a frame length of 18 ms (120 samples) is sufficient for good quality speech analysis [33], [11]. The recursive autocorrelator introduced in chapter 2 gives performance comparable to a 240-point Hamming window, so our window effectively spans two analysis frames.

3.1.2 Choice of System Wordlength

The system wordlength determines the dynamic range of the system. Synchronization between elements of a bit-serial system is simplified considerably if the wordlength is deemed to be fixed throughout the system.

A study of linear predictive analysis of speech in additive white and coloured noise using fixed-point arithmetic concludes that 16-bit accuracy is required of the reflection coefficients and 32-bit accuracy should be maintained in the intermediate calculations [31]. We shall therefore choose the system wordlength to be 32 bits. The reflection coefficients are the output of a division operation which produces a result in bit-parallel format. We will restrict these to 16-bit accuracy.

It should be mentioned in passing that preemphasis filtering should be used to condition the input speech before analysis [21], [29]. Firstly, preemphasis filtering may remove the effects of glottal waveshape from the spectrum, leaving it to more truly represent the shape of the vocal tract. Secondly, proper preemphasis reduces numerical error in the analysis

algorithms. These mean both a reduction in the analysis order and reflection coefficient accuracy required to achieve acceptable performance.

Preemphasis can be accomplished using a filter of the form

$$H(z) = 1 - \gamma z^{-1}$$

where $0 \leq \gamma \leq 1$. Of course, the reconstructed speech must be deemphasized at the receiver by passing it through the inverse filter $\frac{1}{H(z)}$. A more detailed discussion of this topic is beyond the scope of this thesis.

3.1.3 Numeric Overflow in the Input Correlator

The input signal is limited in magnitude to $|s(n)| \leq 1$ by definition. Therefore overflow in the calculation of $s(n)s(n-k)$ cannot occur.

3.1.4 Numeric Overflow in the Recursive Filter Sections

We must guarantee that the multiplier and the adder in the recursive filter section never overflow. This may be accomplished by employing a scaling factor as shown in Figure 3-1. The input to the filter is $x(n)$, the input signal after pre-scaling is $x_1(n)$, the output signal is $y(n)$ and the scaling factor is λ .

We shall assume that the input signal is constrained to $|x(n)| \leq M$. Now, if

$$H(z) = \frac{Y(z)}{X_1(z)}$$

is the transfer function between the input signal after pre-scaling and the output signal, it can be shown that

$$\lambda \leq \frac{1}{M \sum_{i=0}^{\infty} |h(i)|} \quad 3-1$$

is sufficient to ensure that $|y(n)| \leq 1$ [1], [14].

In the case of the recursive sections, the impulse response of the filter without pre-scaling is

$$h(n) = \beta^{2n}$$

(a) Recursive Filter Section

(b) Recursive Filter Section with Prescaling

Figure 3-1: Recursive Filter Section

so

$$\lambda \leq \frac{1}{M \sum_{i=0}^{\infty} \beta^{2i}} = \frac{1 - \beta^2}{M} \quad 3-2$$

since $|\beta| = 0.98 < 1$. For $\beta = 0.98$, $\lambda = 0.0396$. This scaling is accomplished by introducing a right shift of 5 bits before each recursive filtering stage. Note that, in fact, a right shift of 5 bits corresponds to $\lambda = 0.0313$ and the output is constrained to $|y(n)| \leq 0.789$. When the signal $|s(n)s(n-k)| \leq 1$ is passed through three cascaded recursive stages, the output of the third stage will be limited to $|y(n)| \leq 0.491$.

3.1.5 Numeric Overflow in the Non-Recursive Filter Section

We shall take a slightly different approach to dealing with overflow in the nonrecursive filter section. Once again, assume that the input signal is constrained to be $|x(n)| \leq M$.

$$\lambda_k \leq \frac{1}{M \sum_{i=0}^{\infty} |h_k(i)|} \quad 3-3$$

The input of the nonrecursive section is the output of the third recursive section which, in fact, is limited to $|x(n)| \leq 0.491$. Rather than applying a prescaling factor λ , however, we can scale the filter coefficients. Consider a scaling factor ζ_k applied to all of the coefficients as shown in Figure 3-2 (b). It's easy to see that the impulse response

$$h_k(n) = Z^{-1}\{H(z)\} = Z^{-1}\left\{\frac{Y(z)}{X_1(z)}\right\}$$

of the filter with scaled coefficients is

$$h_k(n) = (k+1)\zeta_k\beta^k\delta(n) + (1-k)\zeta_k\beta^{k+2}\delta(n-1)$$

Now, we have

$$\lambda_k \leq \frac{1}{M\zeta_k \left[|(k+1)\beta^k| + |(1-k)\beta^{k+2}| \right]}$$

For the case in which no prescaling factor is applied (that is, $\lambda_k = 1$), the relation

$$\zeta_k \leq \frac{1}{M \left[|(k+1)\beta^k| + |(1-k)\beta^{k+2}| \right]}$$

(a) Nonrecursive Filter Section

(b) Nonrecursive Filter Section with Coefficient Scaling

Figure 3-2: Nonrecursive Filter Section

is sufficient to ensure that $|y(n)| \leq 1$.

Before choosing ζ , a few things should be noted.

- It is intended that one nonrecursive filter be multiplexed among the ten autocorrelation coefficients. This means that to prevent arithmetic overflow, ζ should be less than or equal to the worst case (or smallest) value of ζ_i .

$$\zeta \leq \min(\zeta_0, \zeta_1, \dots, \zeta_9)$$

Table 3-1 (a) contains a list of ζ_i for $i = 0, \dots, 9$. The worst case value of ζ occurs at $i = 9$.

- For the sake of simplicity, inputs to bit-serial multipliers should be restricted to values between -1 and 1.

Referring to Table 3-1 (a), our largest unscaled coefficient is $10\beta^9$. If all coefficients are scaled by the factor

$$\zeta = \frac{1}{10\beta^9}$$

all coefficients will be less than 1 in magnitude, and the constraint

$$\zeta \leq \frac{1}{M \sum_{i=0}^{\infty} |h_i(n)|}$$

is satisfied for all i . Therefore, a value $\zeta = \frac{1}{10\beta^9}$ was chosen as the coefficient scaling factor.

Barnwell's paper on the recursive autocorrelator indicates that small variations in the value of β may be allowed [2]. More specifically, a variation of 0.001 (that is, $\beta = 0.98 \pm 0.001$) has no effect on the quality of the resulting reflection coefficients. For this allowed variation of 0.001, $\lceil \log_2(0.001) \rceil = 10$ fractional bits are required to represent the coefficients with adequate precision. Scaling by a factor of $\zeta = \frac{1}{10\beta^9}$ means that an additional $\lceil \log_2(\frac{1}{10\beta^9}) \rceil = 4$ bits are needed.

Table 3-1 (b) summarizes the "ideal" values of the scaled coefficients and the values quantized to fourteen fractional bits of accuracy in two's complement notation. The leftmost (most significant) bit is a sign bit.

k	$(1+k)\beta^k$	$(1-k)\beta^{k+2}$	ζ_k
0	1	0.9604	0.5100999
1	1.96	0	0.510204
2	2.8812	-0.9223681	0.262911
3	3.764768	-1.8078416	0.1794491
4	4.6118408	-2.6575271	0.1375635
5	5.4235248	-3.4725021	0.1124097
6	6.2008967	-4.2538151	0.0956508
7	6.9450043	-5.0024866	0.0836995
8	7.6568672	-5.7195096	0.0747586
9	8.3374776	-6.4058508	0.0678272

(a) scaling factors ($\beta = 0.98$)

k	$(1+k)\zeta\beta^k$		$(1-k)\zeta\beta^{k+2}$	
	Ideal	Quantized	Ideal	Quantized
0	0.1199403	00.00011110101101	0.1151907	00.00011101011111
1	0.235083	00.00111100001011	0	00.00000000000000
2	0.3455721	00.01011000011101	-0.1106291	11.11100011101100
3	0.4515477	00.01110011100110	-0.2168331	11.11001000100000
4	0.5531458	00.10001101100110	-0.3187447	11.10101110011010
5	0.6504994	00.10100110100001	-0.4164931	11.10010101011001
6	0.7437377	00.10111110011001	-0.510204	11.01111101011001
7	0.8329862	00.11010101001111	-0.6	11.01100110011010
8	0.9183673	00.11101011000110	-0.686	11.01010000011001
9	1	01.00000000000000	-0.76832	11.00111011010100

(b) scaled coefficients ($\beta=0.98$, $\zeta=1/10\beta^9$)**Table 3-1: Nonrecursive Filter Coefficients**

3.1.6 Numeric Overflow in the LeRoux–Gueguen Algorithm

In the LeRoux–Gueguen algorithm, all reflection coefficients K_i that the algorithm produces are less than $R(0)$ in magnitude, as are all intermediate values in the calculations. Furthermore, so are $R(1), \dots, R(p)$. Therefore, ensuring that no overflow occurs in previous stages of the analysis is sufficient to ensure that no overflow occurs in the LeRoux–Gueguen stage.

3.2 A Set of Bit–Serial Simulation Elements

3.2.1 Pseudo 2–Phase Clocking Scheme

We shall adopt a 2–phase clocking scheme for simulation. For CMOS implementation, a pseudo 2–phase clocking scheme will be used [35]. The pseudo 2–phase scheme takes the two non–overlapping clock phases ϕ_1 and ϕ_2 and adds their complements $\bar{\phi}_1$ and $\bar{\phi}_2$. By convention, the inputs of an element are sampled during the first clock phase (ϕ_1) and the outputs of an element are asserted during the second clock phase (ϕ_2). Figure 3–3 shows the clock waveforms for 2–phase and pseudo 2–phase clocking schemes, a delay element constructed using pseudo 2–phase clocking, and a simple finite state machine constructed using pseudo 2–phase clocking.

For clarity in future figures in this chapter, only one clock signal (ϕ) will be shown. It is understood that a 2–phase clock consisting of ϕ_1 and ϕ_2 (and their complements in CMOS implementations) is implied.

3.2.2 Description of Elements

This section briefly describes the major elements in the bit–serial simulation library. In the figures that follow, these elements shall be denoted by the symbols shown in Figure 3–4. For more complete information, consult Appendix A which contains the RATFOR listings for these SPLICE user–defined elements.

(a) Adder/Subtractor

(a) 2-phase clock waveforms

(b) pseudo 2-phase clock waveforms

(c) delay element

(d) simple FSM

Figure 3-3: Pseudo 2-Phase Clocking Scheme

(a) Adder/Subtractor

(b) Serial/Parallel Multiplier

(c) Serial Multiplier

(d) Serial Divider

(e) Fixed Coefficient Multiplier

(f) Bit Serial ROM

(g) Clock Buffer

(h) Latch

(i) Bit Delay

(j) Word Delay

Figure 3-4: Major Elements in the Simulation Library

The adder/subtractor will perform addition or subtraction on 32-bit words supplied at inputs **A** and **B**. The operation is determined by the **Mode** input which is "0" for addition and "1" for subtraction. The input data words must be supplied concurrently with a word synchronization pulse on **WDin**. The **Word** output is a word synchronization pulse for **Sum**.

(b) Serial/Parallel Multiplier

This component is offered in two flavours: one with a twelve-bit parallel input and one with a sixteen-bit parallel input. The twelve-bit serial/parallel multiplier forms the product of **DataI** supplied in bit-serial format and **C11**, ..., **C0** supplied in bit-parallel format. **WordI** is the word synchronization pulse for **DataI**, **Word0** is the synchronization pulse for **Data0** and **Product**. The sixteen-bit version works similarly.

(c) Serial Multiplier

The fully serial multiplier forms the product of **DataI** and **CoefI** supplied in bit-serial format. **WordI** is the synchronization pulse for **DataI** and **CoefI**, and **Word0** is the synchronization pulse for **Coef0**, **Data0** and **Product**.

(d) Serial Divider

The serial divider forms to 16-bit quotient of the 32-bit operands **Divisor** and **Dividend**. **WordI** is the synchronization pulse for **Divisor** and **Dividend**, which are supplied in bit-serial format. The quotient is output in bit-parallel format on **Q15**, ..., **Q0**.

(e) Fixed Coefficient Multiplier

The fixed coefficient multiplier computes the product of **DataI** and a hardwired coefficient. This coefficient is assigned on the **SPLICE** model statement. **WordI** is the word synchronization pulse for **DataI**, **Word0** is the synchronization pulse for **Data0** and **Product**.

(f) Serial ROM

The serial ROM supplies a bit stream of length 512 at its output **Data0**. The ROM's internal counter is reset by the **Reset** input. For simulation purposes, the ROM contents are read from a disk file and initialized at the beginning of each run.

(g) Clock Buffer

The clock buffer supplies the internal clock to major blocks. The clock is enabled when **Enable** is high. Both the ϕ input and the **Clock** output are both two-phase signals (see section 3.2.1).

(h) Latch

The latch is transparent when its *T* input is high, and the output is latched otherwise.

(i) Bit Delay

The bit delay delays the input signal **In** one bit-time.

(j) Word Delay

The word delay delays the input signal **In** one word-time, or 32 bit-times.

3.3 System Organization

3.3.1 Inter-Block Communication

Control and data will be passed among blocks by exchanging control signals. Two control lines are required between blocks in addition to the single data line. The first control line, **WORD**, is for word synchronization. It is required by all bit-serial arithmetic components. **WORD** is high during the least significant bit time of each data word. The second control line, **EXEC**, is for data transfer synchronization. It is high during data transfer between blocks. A group of several words passed between block and accompanied by **EXEC** shall be referred to as a *packet*.

Figure 3-5 clarifies the situation for a typical block. When **EXECI** is asserted at the input of the block, data transfer is initiated. Three data words are read on **DATAI**, each demarcated by a word synchronization pulse on **WORDI**. These three data words form the input packet. Data transfer into the block ends when **EXECI** goes low. Some time after data transfer occurs at the input, the block's processing is completed and data becomes available

(a) Control and Data Signals for a Typical Block

(b) Control and Data Timing for a Typical Block

Figure 3-5: Inter-Block Communication Signals

at the output. The block asserts **EXEC0** to begin the transfer, and the output packet is shifted out on **DATA0**. The output words are accompanied by synchronization pulses on **WORD0**.

3.3.2 Major Subsystems

Figure 3-6 shows the overall organization of the system.

First, the inputs are latched into the input register. The input register is essentially a parallel-to-serial shift register which will latch the parallel inputs and also shift them out serially so that they are in the proper format for the input correlator.

The input correlator block multiplies the current input by itself and by the previous nine inputs to form the correlation lags $s(n)s(n)$, ..., $s(n)s(n-9)$. Referring to Figure 2-3, these are the inputs to the filters $H_0(z)$, ..., $H_9(z)$ respectively.

Three recursive filtering blocks follow the input correlator block. Each recursive filtering block is preceded by a scaling block to prevent overflow of the arithmetic elements inside the filter.

The nonrecursive filtering block follows the three recursive filtering blocks. The output of the nonrecursive filter is the ten autocorrelation coefficients $R(0)$, ..., $R(9)$ (see Figure 2-3).

Finally, the autocorrelation lags are transformed into K_1 , ..., K_9 , the reflection coefficients, by the LeRoux-Gueguen block. This block is partitioned into the coefficient calculation block and the lattice block.

The coefficient calculation block accepts input data from the autocorrelator or a set of intermediate data from the lattice block. It produces the reflection coefficients plus some intermediate data which it ships to the lattice block.

The lattice block accepts a set of intermediate data from the coefficient calculation block and a reflection coefficient and uses these to generate a new set of intermediate data. The new intermediate data is passed back to the coefficient calculation block.

Synchronization of all arithmetic elements in the system is accomplished by the **WORD** synchronization pulse. Since the system wordlength is 32 bits, this pulse may be generated from the system clock by a simple "divide-by-32" circuit.

Figure 3-6: Overall Organization of System

The first control block is activated by the sample clock and the WORD synchronization pulse. This block generates the EXEC signal required by the input correlator. It also generates the LOAD signal which triggers a parallel load of the input register.

The second control block is activated by the frame clock and generates the control signals required by the LeRoux-Gueguen block (WORD and EXEC).

3.4 Circuit Design in Terms of Bit-Serial Primitives

3.4.1 Input Register

The input register block latches the input speech samples (12-bit parallel format) and converts them to 32-bit serial format for the input correlator.

The input register is composed of 32 identical shift register cells, one of which appears in Figure 3-7 (a). These cells operate in two distinct phases. When LD is high (phase 1), data from the parallel input PI is loaded into the cell (that is, appears on the output of the mux). When LD is low (phase 2), data from the serial input SI is loaded into the cell synchronized with ϕ_1 and data is shifted out the serial output SO synchronized with ϕ_2 . The timing diagram for one of the cells is shown in Figure 3-7 (b).

The cells are connected together as in Figure 3-7 (c). The parallel inputs and outputs of cells 1 to 12 are connected to the "outside world" while the serial inputs and outputs are connected to the cells' neighbours. Cell 0 duplicates the sign bit of the input word into the highest-order bit of the serial word. The remaining cells are provided to pad the 12-bit input word up to the system wordlength of 32 bits. For simplicity, the connection of Load and ϕ to every shift register cell was omitted from the diagram.

A short pulse on Load from Control Block #1 loads all of the cells from their parallel inputs. The resulting word will then be shifted down through the serial output and into the input correlation block while Load is low.

3.4.2 Autocorrelation Block

The recursive autocorrelator can be partitioned easily into the following units:

- Input correlator, which produces the sequence $s(n)s(n), \dots, s(n)s(n-9)$ from the input speech samples $s(n)$.
- Scaling block, which scales the data stream prior to each recursive filtering block.
- Recursive filtering block, which performs a first order recursive filtering on the data.
- Nonrecursive filtering block, which performs a nonrecursive filtering on the data.

One input correlation block and one nonrecursive filtering block are required. Three scaling blocks and three recursive filtering blocks are required. The internal construction of these blocks is detailed in the following sections.

3.4.2.1 Input Correlation

The input correlation block forms the products $s(n)s(n), \dots, s(n)s(n-9)$ for each speech sample $s(n)$ that is presented at its inputs. Its block diagram is in Figure 3-8 (a) and its timing diagram is in Figure 3-8 (b).

The current sample $s(n)$ is received from the input register in parallel form at the inputs C11, ..., C0 and in serial form at the DataI input. Previous samples $s(n-1), \dots, s(n-9)$ are stored in the nine word delay blocks. These are retained in between activation cycles.

The multiplexer transmits the serial data word to the multiplier during the first data word time of the current packet, resulting in $s(n)s(n)$. The multiplexer feeds the stored previous samples to the multiplier during subsequent data word times. $s(n)$ remains latched on the parallel input of the multiplier, allowing the products $s(n)s(n-1), \dots, s(n)s(n-9)$ to be calculated.

3.4.2.2 Signal Scaling

The scaling circuit scales the data before input to each recursive filtering stage to prevent arithmetic overflow in the recursive filters. Its block diagram and timing diagram are shown in Figures 3-9 (a) and 3-9 (b). The circuit depends on the word synchronization pulse of

(a) Block Diagram

(b) Control and Data Timing Diagram

Figure 3-8: Input Correlator Implementation

(a) Block Diagram

(b) Control and Data Timing Diagram

Figure 3-9: Signal Scaling Block Implementation

the following word in the data packet to latch its sign bit and thus accomplish a five-bit right shift with sign extension.

Sign extension of the last data word in a packet is a special case – in this situation there is no subsequent word and therefore no word synchronization pulse. Latching in this case occurs when the input *ExecI* goes low (that is, the end of the packet has been reached).

3.4.2.3 Recursive Filtering

The block diagram of the recursive filtering block is shown in Figure 3-10 (a) and the timing diagram in Figure 3-10 (b). One multiplier and one adder are multiplexed to handle all of the necessary operations. The ten previous output values are stored in the nine word delays and in the multiplier's internal shift register between activation cycles.

3.4.2.4 Non-Recursive Filtering

The nonrecursive filter's block diagram appears in Figure 3-11 (a) and its timing diagram is shown in Figure 3-11 (b).

The filter coefficients are obtained from the two ROMs (please see Table 3-1 (b) for the values of these coefficients). Each ROM must hold ten 32-bit coefficients.

The ten data words from the previous packet are stored in the ten word delays between activation cycles. When a data packet arrives at the input of the block, the new data and the stored data from the previous packet are multiplied by the appropriate coefficients and added to produce ten data words at the output.

3.4.3 LeRoux-Gueguen Algorithm

The LeRoux-Gueguen algorithm is less easily partitioned than the recursive autocorrelator, but a reasonable breakdown seems to be:

- Coefficient calculation block, which performs the division required to calculate the reflection coefficients and stores the intermediate values calculated in each iteration
- Lattice block, which accepts a set of intermediate values from the coefficient calculation block and generates the next set of intermediate values.

(a) Block Diagram

(b) Control and Data Timing Diagram

Figure 3-10: Recursive Filter Implementation

(a) Block Diagram

(b) Control and Data Timing Diagram

Figure 3-11: Nonrecursive Filter Implementation

These are detailed in the following sections.

3.4.3.1 Coefficient Calculation Block

The coefficient calculation block performs the division necessary to calculate the reflection coefficients and stores the intermediate values calculated in the LeRoux–Gueguen iteration. Its block diagram and timing diagram are shown in Figures 3–12 (a) and 3–12 (b) respectively.

This block receives data packets from either the autocorrelator and Control Circuit #2 (*via* the **DataI**, **WordI**, **ExecI** inputs) or from the lattice block (*via* the **DataI1**, **DataI2**, **WordI1**, **ExecI1** inputs). The packets coming from the autocorrelator are the raw autocorrelation values $R(0), \dots, R(9)$ for a frame of speech. The packets coming from the lattice block are intermediate values to be used in the calculation of the next reflection coefficient.

As each reflection coefficient is made available at the outputs **C0**, **C1**, ..., **C15** the **Result** output is pulsed high for one cycle of the system clock.

3.4.3.2 Lattice Block

The lattice block accepts a set of intermediate values and a reflection coefficient from the coefficient calculation block on the inputs **DataI1**, **DataI2**, **WordI**, **ExecI**, **C0**, **C1**, ..., **C15**. It then passes the intermediate data through a lattice section to obtain the next set of intermediate values required by the coefficient calculation block. The output data packet is routed from the **DataO1**, **DataO2**, **WordO**, **ExecO** outputs back to the **DataI1**, **DataI2**, **WordI**, **ExecI** inputs of the coefficient calculation block.

The block diagram and timing diagram of the lattice block appear in Figures 3–13 (a) and 3–13 (b).

3.4.4 Control Circuitry

Two small finite state machines (FSMs) are required to produce control signals for the various blocks.

The first FSM generates the **Latch** signal to load the input register and the **ExecO** signal which is routed to the **ExecI** input of the input correlator. These signals are produced in

Figure 3-12: Coefficient Calculation Block

(b) Control and Data Timing Diagram

Figure 3-12: Coefficient Calculation Block

(a) Block Diagram

(b) Control and Data Timing Diagram

Figure 3-13: Lattice Block

(a) State Diagram

(b) Timing Diagram

Figure 3-14: FSM Activated by Sample Clock

(a) State Diagram

(b) Timing Diagram

Figure 3-15: FSM Activated by Frame Clock

response to the sample clock and the word synchronization pulse **WordI** which is derived from the system clock. The state diagram and timing diagram for this FSM are shown in Figures 3-14(a) and 3-14(b).

The second FSM generates the **Exec0** and **Word0** signals which are connected to the **ExecI** and **WordI** inputs of the coefficient calculation block. These signals are produced in response to the frame clock and **EXEC** and **WORD** control outputs from the nonrecursive filtering block. The state diagram and timing diagram for this FSM are shown in Figures 3-15(a) and 3-15(b).

For simulation purposes the FSMs were implemented as **SPLICE** components in **RATFOR** (see Appendix A). The hardware realization of these would be accomplished with a small amount of random logic or a small PLA. Automatic generation of the netlist from a description in DDL [4] or another synchronous hardware description language is commonplace.

3.5 System Simulation

This section describes the steps that were taken to verify the design presented in this chapter. First, the basic algorithms (recursive autocorrelation and LeRoux-Gueguen) were implemented using a sequential programming language. The output from these algorithms was compared with coefficients calculated using a traditional analysis algorithm. Second, the hardware designed in this chapter was simulated using **SPLICE**. The results of this simulation were then compared with the results of the algorithmic simulation to verify that the design of the hardware was, indeed, correct.

3.5.1 Algorithm Verification

The software procedure **AUTO** written by Gray and Markel [7] was used to perform a ninth-order linear predictive analysis on 36,000 samples of speech uttered by a male speaker, and quantized to 12-bit accuracy. A frame length of 120 samples was used, and the speech was windowed using a sliding 240-point rectangular window. The recursive

autocorrelation and LeRoux–Gueguen algorithms were implemented in RATFOR and used to analyze the same 36,000 samples of speech.

Both of the resulting sets of reflection coefficients were converted to so-called predictor coefficients $a(1), \dots, a(9)$ and spectra were estimated for each frame according to

$$V(k) = \left| \frac{1}{1 - \sum_{p=1}^9 a(p)e^{-j\frac{2k\pi p}{128}}} \right|^2$$

Now, if $V_{GM}(k, l)$ is the k 'th spectral sample of the l 'th frame calculated using the Gray and Markel algorithms, $V_r(k, l)$ is the k 'th spectral sample of the l 'th frame calculated using the recursive autocorrelation and LeRoux–Gueguen methods and M is the number of frames, a spectral distance measure (in dB) can be computed using

$$E = \left[\frac{\sum_{l=1}^M \frac{1}{128} \sum_{k=0}^{127} \left| 10 \log \frac{V_{GM}(k, l)}{V_r(k, l)} \right|^2}{M} \right]^{\frac{1}{2}}$$

This test was conducted over $M = 50$ frames and $M = 300$ frames of speech. The resulting spectral distances between the two methods were 3.45 dB over 50 frames and 3.36 dB over 300 frames. Similar results are quoted in Barnwell's paper [2].

3.5.2 Hardware Simulation

The extremely wide time scale in this system was a hindrance to simulation. At the low end of the scale, the system clock frequency is 10 MHz, so the clock period is 100 ns. At the high end of the scale, the sample clock has a frequency of 6.67 kHz and the frame clock has a frequency of 55.55 Hz (their periods are 150 μ s and 18 ms, respectively). A simulation of the system for 300 frames would be a simulation for 5.4 s, or 5,400,000,000 ns of simulation time. Such a simulation would require weeks of real time on the VAX 11/750. The system was split into two sections for this reason. The sections were simulated separately.

First, the assembly composed of the recursive autocorrelator, input register and control circuit # 1 was isolated. It was observed that the input correlator requires 3.2 μ s to

perform its calculations. The period of the sample clock was therefore decreased to $4\ \mu\text{s}$ to eliminate the "dead time" between samples. The autocorrelator was simulated for 480 samples and the resulting autocorrelation coefficients were compared with those computed as intermediate values in the RATFOR implementation (section 3.5.1). Small discrepancies were noted – over 480 samples of speech the worst case noise in the autocorrelation coefficients due to finite-word-length effects was -132 dB (calculated with respect to the dynamic range of the signal). To put this another way, the effects of using finite precision arithmetic were confined to the least significant eight bits of the 32-bit word. The average noise on the autocorrelation coefficients was -159 dB (the least significant four bits were affected).

Second, the LeRoux–Gueguen block and control circuit # 2 were simulated. The LeRoux–Gueguen block requires $313.6\ \mu\text{s}$ to produce nine reflection coefficients from ten autocorrelation coefficients. The period of the frame clock was decreased to $329.6\ \mu\text{s}$ to eliminate "dead time" between activation cycles. This block was simulated for 50 frames of speech, using autocorrelation coefficients obtained from the RATFOR simulation (section 3.5.1) as input. The reflection coefficients thus obtained were compared to those produced by Gray and Markel's algorithm. The spectral distance over 50 frames of speech was 3.43 dB. When compared to the reflection coefficients computed by the RATFOR implementation of this algorithm, the spectral distance over 50 frames was 0.12 dB.

The SPLICE netlist used for the simulation of the autocorrelator is in Appendix B, and the netlist for the LeRoux–Gueguen block is in Appendix C. The simulation of the recursive autocorrelator for 480 samples took 49.68 CPU hours on a VAX 11/750 under VMS version 4.4. The simulation of the LeRoux–Gueguen block for 50 frames required 24.16 CPU hours on the same VAX.

3.6 Summary

This chapter dealt with the top-level design of the speech analysis system. In summary, the following activities were carried out:

- The input data format and the system wordlength were determined.

- Steps were taken to deal with the problem of numeric overflow in the system
- A set of primitive arithmetic elements was developed for simulation. The elements were compiled into the SPLICE simulator and are available to any SPLICE netlist.
- Clocking and communication policies were established.
- The system was partitioned into subsystems of a convenient size. The subsystems, in turn, were implemented in terms of the primitive arithmetic elements designed in this chapter.
- The system was simulated in two steps. First, the chosen algorithms were implemented in software and the reflection coefficients thus calculated were compared with those calculated using algorithms published elsewhere. Second, the system as designed in this chapter was simulated and the resulting reflection coefficients were compared with those calculated by the software implementation.

The top-level design is now completed. Chapter four contains details of the implementation of portions of the system on silicon.

Chapter 4

Implementation of Test Circuits

The low-level design of some subunits of the speech analysis system will be considered in this chapter. The system-level design issues were covered in Chapter Three.

In section 4.1, two blocks are chosen for trial implementation because they are moderately complex and perform computationally demanding functions. Some layout conventions are introduced in section 4.2. The layout of the input correlator is detailed in section 4.3. The recursive filtering circuit is laid out in section 4.4. Sections 4.5 and 4.6 concern the assembly of the test chip and complexity estimate for the entire system.

4.1 Choice of Circuits for Trial Realization

Neither the time nor the facilities were available to attempt to realize the entire linear predictive coder on silicon. Those parts of the system which perform the most computationally intensive functions were chosen for trial realization.

Of the major blocks in the design shown in Figure 3-6, the lattice block, the coefficient calculation block and the non-recursive filter block need only perform their operations once per analysis frame. These were assigned low priority for realization because they do not represent computationally demanding aspects of the calculation.

The input correlator and recursive filter blocks, on the other hand, are active every sample and are moderately complex. They were assigned the highest priority for realization.

The scaling blocks, input register and the finite-state machine controllers are active every sample and should be extremely trivial to realize. They were assigned medium priority, and it was intended that these be added to the test chip if time allowed.

4.2 Layout Notes

The layout was done using VIVID, the Microelectronics Center of North Carolina's virtual-grid symbolic layout system. VIVID ICE is a graphical editor through which virtual-grid symbolic layouts are created. VIVID HCOMPACT converts the virtual-grid symbolic layouts into mask geometries. HCOMPACT works best when a gate-matrix layout style is adhered to [34]. Briefly, the guidelines are

- Polysilicon wires run vertically
- Diffusion wires run horizontally
- Metal wires may run vertically or horizontally
- Transistors may only be placed on polysilicon columns

VIVID ABSTRACT extracts a simulation netlist from the virtual-grid symbolic layout. It also attempts to estimate nodal capacitances. VIVID FACTS may then be used for electrical simulation of the circuit. ABSTRACT's guesses were quite good; the netlist so obtained proved adequate for verifying the functions of the circuits and determining if timing constraints were met. After compaction, a SPICE netlist may be extracted from the mask geometry and the circuit may be simulated using SPICE. This step will yield somewhat more accurate timing figures.

4.3 Input Correlator Block

This block can be further subdivided into these functional units:

- Clock buffer
- Serial/parallel multiplier
- Delay line (9 32-bit words)
- Control circuitry
- Input multiplexer and a few delay elements required to pad out the total delay of the multiplier path to be 32 bits.

These are dealt with in sections 4.3.1 through 4.3.5.

4.3.1 Clock Buffer

The clock buffer was designed to easily drive a large on-chip load at the target system clock rate of 10 MHz. The gate capacitance of a minimum-size ($W = L = 3\mu m$) transistor was estimated to be $6.37 fF$ from the formula

$$C_g = \frac{\epsilon_o \epsilon_{SiO_2}}{t_{ox}} W L$$

where ϵ_o is the permittivity of vacuum ($8.854 \times 10^{-12} F/m$), ϵ_{SiO_2} is the relative permittivity of SiO_2 (taken to be 4) and t_{ox} is the thickness of the gate oxide (50nm for the Northern Telecom CMOS3 process [10]). A reasonable value for a "large" on-chip load is 1 pF. Transistors will generally turn out to be two or three times the minimum size – a 1 pF load is approximately 50 to 75 such transistors. For a system clock rate of 10 MHz, a clock pulse width in the range of 30 ns would be appropriate. It is reasonable to design the clock buffer to drive 1 pF with rise and fall times of less than 10 ns.

This performance goal was met by using $\frac{W}{L}$ ratios of 20 for the n-channel devices and 40 for the p-channel devices in the large inverters shown in Figure 4-1. It is advisable to build such large devices using several parallel transistors, as in the layout shown in Figure 4-2. The benefits are reduced parasitic capacitance due to reduced size of the drain diffusions, reduced resistance in the polysilicon gates and reduced susceptibility to latchup.

As this particular primitive is crucially important to the overall performance of the circuit, extra care was taken during simulation. First, the circuit netlist was extracted from the virtual-grid symbolic layout and the circuit was simulated using VIVID FACTS. The results of this simulation are shown in Figure 4-3. Numerical values of rise time (t_r), fall time (t_f), the delay times for rising and falling outputs (t_{dr}), (t_{df}), and the pulse width (PW) were obtained from the FACTS simulation results.

Then, after the mask layout was obtained from VIVID HCOMPACT, a SPLICE netlist was extracted using ph1ex on the Metheus-CV workstation and the simulation was repeated using SPICE Level 2. These results appear in Figure 4-4. Once again, numerical values for rise time, fall time and so on were obtained from the simulation results.

Figure 4-1: Clock Buffer Schematic

Figure 4-2: Clock Buffer Layout

Figure 4-3: FACTS Simulation Results for Clock Buffer

Figure 4-4: SPICE Simulation Results for Clock Buffer

	FACTS		SPICE		
	NO LOAD	1 pF LOAD	NO LOAD	1 pF LOAD	2 pF LOAD
t_r	1.60 ns	2.71 ns	8.01 ns	8.97 ns	10.67 ns
t_f	1.41 ns	2.05 ns	6.06 ns	6.79 ns	7.47 ns
t_{dr}	4.34 ns	5.15 ns	15.16 ns	16.39 ns	17.46 ns
t_{df}	3.15 ns	3.83 ns	9.82 ns	10.73 ns	11.38 ns
PW	28.81 ns	28.68 ns	25.66 ns	25.34 ns	24.92 ns

Table 4–1: Summary of Simulation Results for Clock Buffer

The results observed from both simulations are summarized in Table 4-1. The SPICE simulation shows that there is considerable inertia associated with the buffer itself: the delay time is significant and the output clock pulse is noticeably shaped. However, the output pulse width, measured between the points where the rising and falling edges have the values 2.5 V, suffers little degradation even at a load capacitance of 2 pF. The results indicate that the driver should perform well.

The FACTS results were considerably more optimistic than SPICE results (by a factor of 3 or 4). FACTS runs many orders of magnitude faster than SPICE, however, and provides verification of logical correctness.

4.3.2 Serial/Parallel Multiplier

A slightly simplified version of the pipelined multiplier reported by Lyon [18] was chosen to realize the serial/parallel multiplier. This multiplier is build up in stages – one stage is required for each bit in the coefficient. Three distinct types of stages can be identified. The first, or “adding,” stage is used for every coefficient bit except the last two. The second, or “last adding,” stage is used for the second-last coefficient bit. The third, or “subtracting,” stage is used for the last coefficient bit.

Schematic diagrams of the three stages appear in Figure 4-5(a), 4-5(b), and 4-5(c). The serial data word enters each stage *via* the X_i input and leaves through the X_o output. The partial product from the previous stage enters at the PP_i input and the resulting partial product is produced at the PP_o output. The word synchronization pulse enters and leaves at the R_i and R_o lines.

Figure 4-5(d) shows the floorplan of a serial/parallel multiplier with a twelve-bit coefficient. The broad “U” shape of the multiplier is intended to fit it more snugly with the other parts of this block.

The layout of the subtracting stage of the multiplier appears in Figure 4-6. All of the cells in the multiplier are constructed with four tiers of p-channel and n-channel transistors. Power (v_{ss} , v_{dd}) and clock (p1, p2, p1b, p2b) run in metal from east to west through the cell. The cells are designed to butt with adjoining multiplier cells at the east and west sides

(a) Schematic of adding stage

(b) Schematic of last adding stage

Figure 4-5: Serial/Parallel Multiplier

Figure 4-6: Layout of Serial/Parallel Subtracting Stage

Figure 4-7: Layout of Serial/Parallel Multiplier

– the signals x_i , x_o , ppi , ppo , ri , ro are available in metal at the cell boundaries. The coefficient bit y_i is input from the north.

Figure 4-7 shows the plot of the metal layer of the complete serial/parallel multiplier. Note that a clock buffer (section 4.3.1) is appended to the side of the multiplier.

4.3.3 Delay Line

A dynamic delay line was implemented using the cell shown in Figure 4-8(a). The layout of this cell is shown in Figure 4-8(b). Clocks and power are routed in metal from east to west. Longer delay lines are formed by butting a number of these cells along their east and west boundaries.

The delay line required for this block which is a series of 288 such cells, is shown in Figure 4-9. This yields a total delay of 9 32-bit words. Only the metallization layers were plotted in an attempt to maintain some clarity in this figure. A clock buffer is also included in the plot of this structure.

4.3.4 Control Circuitry

The schematic diagram of the required control circuitry appears in Figure 4-10. The word delay is composed of 32 delay elements, which were described in the previous section. Figure 4-11 shows the mask layout of this unit.

4.3.5 Multiplexer and Delay Compensation Circuitry

Figure 4-12 shows a schematic diagram of this unit.

The “delay compensation” function of this unit deserves some comment. It was assumed in the high-level simulations conducted in Chapter 3 that the delay or “latency” of the serial/parallel multiplier was 32 bits. In fact, the multiplier as realized in section 4.3.1 has a latency of 24 bits. The delay between the output of the multiplexer and the output of the multiplier must therefore be padded out to 32 bits by the addition of 8 delay elements in the data and control word paths. Note that the application of a pulse at the PFI input of the multiplier causes rounding of the next generated product to take place. The PPO output of this circuit exists to take advantage of this feature.

(a) Schematic of one delay cell

(b) Layout of one delay cell

Figure 4-8: Delay Line Element

Figure 4-9: Layout of 9 x 32-bit delay line

Figure 4–10: Correlator Control Circuit Schematic

Figure 4-11: Correlator Control Circuit Layout

Figure 4-12: Multiplexer and Delay Compensation Schematic

Figure 4–13: Multiplexer and Delay Compensation Layout

The mask layout of this unit is shown in Figure 4-13.

4.4 Recursive Filtering Block

The recursive filtering block requires that these functions be realized:

- Serial fixed-coefficient multiplier and delay compensation circuitry
- Delay line (9 32-bit words)
- Subtractor
- Clock buffer

It was mentioned in Chapter 3 that Barnwell's study of the recursive autocorrelator structure indicates that allowing a small variation in the value of β (say, $\beta = 0.98 \pm 0.001$) has no perceptible effect on the quality of the resulting reflection coefficients [2]. This fact was used to limit the number of bits in the coefficient of the fixed-coefficient multiplier. Since the allowed variation is 0.001, $\lceil \log_2(0.001) \rceil = 10$ fractional bits are required to represent the coefficient with adequate precision. If $\beta = 0.98027419633488$ the twelve-bit two's complement representation of $\beta^2 = 0.9609375$ is 00.1111011000. The most significant bit is the sign bit.

Representing the coefficient as a negative number results in space savings because "0" coefficient bits require somewhat less area to implement than "1" coefficient bits. Representing the coefficient as a negative value also means, of course, that a subtractor is needed rather than an adder. The negative coefficient $-\beta^2 = -0.9609375$ has 11.0000101000 as its twelve-bit two's complement representation.

The delay line and the clock buffer are essentially the same as those discussed for the input correlator. The construction of the serial fixed-coefficient multiplier and subtractor will be discussed in sections 4.4.1 and 4.4.2.

4.4.1 Serial Fixed-Coefficient Multiplier

This multiplier is implemented as a much simplified case of the pipelined multiplier reported by Lyon [18]. The multiplier requires one stage for each bit in the coefficient. To

(a) Adding stage with "0" coefficient bit

(b) Adding stage with "1" coefficient bit

Figure 4-14: Fixed Coefficient Multiplier

(c) Last adding stage with "1" coefficient bit

(d) Subtracting stage with "1" coefficient bit

(e) Floorplan of multiplier with coefficient of -0.9609375

Figure 4-14: Fixed Coefficient Multiplier

Figure 4-15: Layout of Fixed Coefficient Subtracting Stage

Figure 4-16: Layout of Fixed Coefficient Multiplier

realize the multiplier needed here, four different leaf cells were designed. Figure 4-14(a) shows an "adding" cell with "0" coefficient bit. Figure 4-14(b) shows an "adding" cell with "1" coefficient bit. Figures 4-14(c) and 4-14(d) show "last adding" and "subtracting" cells, both with "1" coefficient bits.

The mask layout of the final subtracting stage of the multiplier appears in Figure 4-15. The floorplan of the multiplier with coefficient $-\beta^2 = -0.9609375$ and the mask layout of that multiplier are shown in Figures 4-14(e) and 4-16. As with the serial/parallel multiplier described earlier in this chapter, all elements of the fixed-coefficient multiplier are designed with four tiers of p-channel and n-channel devices with clocks and power routed through from east to west. x_1 , x_0 , ppi , ppo , r_1 and r_0 are all available at the east and west cell boundaries.

The total latency of the multiplier is only 24 bits, so delay compensation circuitry is required as in the input correlator. Once again, note that applying a pulse to the PPI input of the multiplier causes rounding of the next generated product to take place.

4.4.2 Subtractor

The schematic diagram of the subtractor is shown in Figure 4-17 and the layout in Figure 4-18. The subtractor is a simple combinational circuit with a carry feedback latch which is preset on the least significant bit of the input data word.

4.4.3 Delay Compensation Circuitry

Figure 4-19 shows the schematic and Figure 4-20 the layout of the delay compensation circuitry.

4.5 Assembly of Test Chip

The two circuits described in this chapter were wired into an A-sized pad frame and submitted to the Canadian Microelectronics Corporation for fabrication by NTE (Northern Telecom Electronics). Figure 4-21 shows the floorplan of the chip and Figure 4-22 shows a plot of the metallization layers.

2 of/de 2

MICROD

Figure 4-17: Subtractor Schematic

Figure 4-18: Subtractor Layout

Figure 4-19: Delay Compensation Schematic

Figure 4-20: Delay Compensation Layout

Figure 4-21: Chip Floorplan

Figure 4-22: Chip Plot

4.6 System Complexity Estimate

Based on the experience of implementing some of the primitive elements it is possible to make an estimate of the total complexity of the system. The first step is to estimate the complexity of the component parts, the bit-serial primitives. Some of these were actually implemented in this chapter, so the complexity can be obtained by counting the devices.

The adder/subtractor required 54 devices. The serial/parallel multiplier required 90 devices per stage (one stage is required for each bit in the coefficient). The fixed-coefficient multiplier is the simplest at approximately 70 devices per stage, on average. Clock buffers required 96 devices, bit delays required 8 devices, word delays required 256 devices, and muxes required 6 devices.

The sizes of the parts that were not implemented can generally be extrapolated from the sizes of the parts that were. The fully serial multiplier is slightly more complex than the serial/parallel multiplier and it should require approximately 110 devices per stage. The serial divider is best implemented using a convergence technique as described in [13]. Two fully serial multipliers and two adders would be required to achieve adequate throughput. The divider therefore would require 100 devices (for both adders) plus 220 devices for each stage in both multipliers. An array implementation of the divider was considered and dismissed because of excessive complexity – somewhat more than 17,000 devices would be required to implement the array divider. The complexity of the serial ROM is the most difficult to estimate because nothing similar in nature was implemented in this chapter. Two possible designs were considered: a ROM with 320 individually addressable cells plus a counter to generate the addresses, and a very long shift register which can be preloaded with the nonrecursive filter coefficients (something like the input register circuit designed in Chapter 3). The first design is estimated to require approximately 1500 devices per 320-bit ROM and is the less complex of the two. These 1500 devices are almost all used for address decoding.

The finite state machine controllers were not considered in the complexity estimate. They are likely implemented as PLAs and, as they each have about 11 product terms (PLA

	Input Reg.	Input Corr.	Scale	Rec. Filter	Nonrec. Filter	Coeff. Calc.	Lattice
Adder/Subtractor 54 devices				1 (54)	1 (54)		2 (108)
Serial/Par. Mult 90/stage		1 12-bit (1080)					2 16-bit (2880)
Serial Multiplier 110/stage					2 16-bit (3620)		
Serial Divider 100 + 220/stage						1 16-bit (3628)	
Fixed Coeff Mult 70/stage				1 12-bit (840)			
Serial ROM 1500/320 bits					2 (3000)		
Clock Buffer 96 devices		1 (96)		1 (96)	1 (96)		
Latch 6 devices	96 (768)		1 (8)				16 (128)
Bit Delay 8 devices		16 (128)	11 (88)	16 (128)	3 (24)		
Word Delay 256 devices		10 (2560)		9 (2304)	11 (2816)	41 (10496)	1 (256)
Mux 6 devices	32 (192)	1 (6)				3 (18)	
Miscellaneous		(10)	(26)		(8)	(54)	(2)
Total Devices	(960)	(3880)	(122)	(3422)	(9514)	(14196)	(3374)

Table 4-2: System Parts List and Complexity Estimate

rows) and 6 inputs and output (PLA rows), would add a few hundred devices to the total.

Table 4-2 summarizes these estimates along with the parts counts for the major subsystems. The parenthesized numbers in each column are transistor counts. The shaded areas are the parts which were not implemented in this chapter and whose size is therefore estimated. The complexity estimate for the system was obtained by summing the total number of devices in all of the major subsystems. Note that three scaling blocks and three recursive filter blocks are required in the system. The total number of devices required is 42,168 which is well within the reach of commercial gate array technology. This table also shows that the nonrecursive filter block, the coefficient calculation block and the lattice block account for over six tenths of the devices while performing the less demanding tasks.

Chapter 5

Conclusions and Recommendations

5.1 Thesis Summary

An architecture was developed that is suitable for incorporation into a speech coder for a digital mobile radio transceiver. The following steps were carried out to arrive at this goal:

- 1) The autocorrelation method of linear predictive analysis was examined in some detail. Recursive windowing and the LeRoux–Gueguen algorithm were identified as techniques that will lead to an efficient hardware realization.
- 2) The benefits of bit–serial architectures for low bandwidth digital signal processing were reviewed : bit–serial architectures offer area efficiency, straightforward system partitioning and reduced interconnect both on–chip and between chips.
- 3) A hardware implementation strategy composed of two main thrusts was devised. The first phase of the implementation proceeds in a top–down manner. The algorithms are specified in the form of a signal flowgraph. Major system decisions, such as system wordlength, handling of FWL effects, and so on, are made. The system is designed down to the level of bit–serial primitive elements. This phase is supported by the SPLICE simulator. The second phase of the implementation proceeds in a bottom–up manner. The bit–serial primitives required to realize the system are assembled from the ground up, and are then put together to form the complete system. The second phase is primarily supported by the VIVID design system.

4) A signal flowgraph of the entire system is obtained to begin the top-down implementation phase. Based on the results of previous research, the following system parameters were chosen:

- Input speech samples are 12-bit, linearly quantized, two's complement.
- Sampling rate of input speech is 6.67 kHz.
- Analysis order is 9.
- Analysis frame length is 18 ms.
- System wordlength is 32 bits.
- Accuracy required of reflection coefficients is 16 bits.

Numerical overflow in the various subsystems was considered and signal scaling was employed to avoid it.

- 5) A set of bit-serial arithmetic primitives was developed and integrated with the Carleton University SPLICE simulator, thus establishing a simulation environment for bit-serial systems.
- 6) Communication policies between system blocks were defined and the system was partitioned into subsystems. Functional units were multiplexed where appropriate to yield more efficient utilization of resources. Each subsystem was then implemented in terms of bit-serial primitive elements and the entire system was simulated using SPLICE. The results of this simulation were compared with the results of a high-level language implementation of the algorithms using a spectral distance measure. The performance of the system differed only slightly from that of a software implementation using floating point arithmetic.
- 7) Two major subcircuits were chosen for trial implementation and were realized in Northern Telecom's 3 micron CMOS technology. The complexity of the entire system was estimated to be less than 45,000 transistors. Circuits of this size can easily be realized as full custom chips or gate arrays. This

demonstrated the feasibility of the design.

The two major software packages used in this research were Carleton University's customized version of SPLICE, and the Microelectronics Center of North Carolina's VIVID symbolic layout system. It is worth making some remarks about the usefulness and quality of these tools.

SPLICE served its purpose well, all things considered. The ability to define models and integrate them with the simulator made it possible to perform a reasonable system level simulation before the internal details of the bit-serial arithmetic primitives were known. However, the disadvantage is that defining new simulation models is not straightforward. It is difficult, and sometimes impossible, to create parameterized models. SPLICE also ran rather slowly, but this reflects the inadequacies of the simulation platform (VAX 11/750) rather than any problem with SPLICE.

In VIVID's symbolic layout system, relative transistor placement, sizing and connectivity are established using symbolic representations of the devices. Another program automatically converts this symbolic layout into a mask layout in a process known as "compaction." The designer is spared the tedious chore of placing contact vias, guard rings, and so on – productivity increases by an order of magnitude. Unfortunately, VIVID's compacter allows no user intervention in the compaction process and thus many possible optimizations are overlooked. Furthermore, the compaction algorithms are not particularly clever. The mask layouts shown in Chapter 4 were produced by the VIVID compacter; almost without exception they contain considerable wasted area.

VIVID's circuit simulator, FACTS, was used to simulate the circuits designed. VIVID's documentation claims a large speed advantage over SPICE. Certainly, simulation timescales which required several seconds to cover using FACTS required on the order of minutes using SPICE. FACTS owes its speed to the comparatively simple MOSFET models used and very approximate treatment of layout-related capacitances. In short, FACTS does not provide nearly the simulation accuracy of SPICE, but is useful for verifying the function of circuits.

5.2 Suggestions for Further Research

There are several ways in which the work done in this thesis could be improved upon or extended.

The consideration of numerical accuracy (section 3.1) could be expanded upon. This is an important issue because the physical (chip) area consumed by a bit-serial design tends to vary as the system wordlength. The results of the system-level simulation conducted in Chapter 3 (0.12 dB difference between fixed and floating point implementations) suggest that the choice of system wordlength is too conservative. At least one relevant study exists (see [31]), but it is targeted toward vocoders rather than hybrid coders such as RELP and RPE-LPC. Determining the effect of system wordlength on the fidelity of a RELP or RPE-LPC coder would allow a system wordlength to be chosen to minimize chip area while maintaining adequate speech quality. Some specific issues to consider are:

- The use of saturating arithmetic elements. If the restriction that elements may never overflow is lifted and saturating arithmetic elements are implemented, several fewer bits may be required.
- The shape of the preemphasis filter (see section 3.1.2) may have an effect on the precision required.

As the introductory section of Chapter 4 pointed out, some of the subsystems are active every sample and some are active every frame. The greatest benefits accrue from implementing the most active subsystems as custom hardware. The tradeoffs between custom implementation and implementation on a standard DSP device must be investigated. A hybrid system in which the more intensive computations (input correlation, scaling, and recursive filtering) are performed by dedicated bit-serial circuits and the less intensive computations (nonrecursive filtering and LeRoux-Gueguen algorithm) are executed on a von Neumann or Harvard architecture is of particular interest. Such systems become increasingly attractive as many ASIC vendors begin to offer microprocessors as "macrocells." When implementing a speech coder or transceiver on a chip, several computationally light tasks could share such a microprocessor.

There is a great deal of room for improvement in CAD tools. Improving the simulation environment for bit-serial systems would involve reimplementing the primitives detailed in Appendix A, parameterizing them with respect to word length. A number of higher level primitive elements, such as integrators, differentiators, FFTs, noise generators, and so on would also be useful in the design of DSP functions [9]. If current industry trends continue, VHDL will be the language of choice for implementing these simulation primitives: most CAD vendors are committed to support VHDL on their simulators and several full VHDL simulators are on the market. A logical extension of this work would be a module generator which produces layout from the system-level simulation netlist (essentially a data flow graph).

Finally, implementing a speech coder or transceiver on a chip would be an ambitious project. The estimated complexity of the speech analysis system designed in this thesis indicates that this is an entirely reasonable goal. The size of this project dictates that it would be better carried out in a commercial setting.

Appendix A

Serial Arithmetic Components for SPLICE Simulator

```

##-----
##
## Bit serial arithmetic components for SPLICE
##
## Copyright 1987, 1988, John W. O'Leary
##
## This file contains the RATFOR implementation of a set of
## bit--serial arithmetic components for use with the SPLICE
## simulator. These models can be made available for use in a
## SPLICE netlist by issuing the appropriate model statement:
##
## .model <some name> lgmod : model=<model #> ...
##
## For example,
##
## .model READ lgmod : model=1 pari=10    ;-- read from unit 10
## .model WRITE lgmod : model=3 pari=9    ;-- write to unit 9
##
## The following models are defined in this file:
##
## MODEL # | FUNCTION
## -----+-----
##      1  | Read data from file, convert to bit-serial format
##      2  | Read data from file, convert to bit-serial format
##      3  | Write data (in bit-serial format) to a file
##      4  | Write data (in bit-serial format) to a file
##      5  | Read data from file, convert to bit-parallel format
##      6  | Write data (in bit-parallel format) to a file
##     11  | Fixed-coefficient bit-serial multiplier
##     12  | 12 bit parallel-coefficient bit-serial multiplier
##     13  | Word delay
##     14  | Clock buffer - enable/disable the clock
##     15  | Bit delay
##     16  | Serial adder/subtractor
##     17  | Latch
##     19  | 32 bit serial coefficient, serial data multiplier
##     20  | 32 bit divider with bit parallel quotient output
##     21  | Constant generator (bit-serial ROM)
##     22  | Constant generator (bit-serial ROM)
##     23  | 16 bit parallel-coefficient bit-serial multiplier
##     31  | FSM to control the input reg. and input correlator
##     32  | FSM to control the coefficient calculation block

```

```

##          |
##
##-----
##-----
##
## COMPONENT      : Finite state machine to control the input
##                  register and supply control signals to the
##                  input correlator.
##
## FUNCTION       : This FSM generates the LATCH signal for the input
##                  register and the EXEC signal for the input corr-
##                  elator in response to the sample clock and WORD
##
## MODEL STATEMENT : .model fsm1 lgmod : model=31
##
## INSTANTIATION  : <name> LATCH,EXEC,WORD,SAMPLE,PHI1,PHI2 fsm1
##
## OUTPUTS       : LATCH - Latch control signal for input register
##                  EXEC  - Data transfer synchronization signal
##                  for input correlator
##
## INPUTS        : WORD   - Word synchronization pulse input
##                  SAMPLE - Sample clock
##                  PHI1   - Clock 1
##                  PHI2   - Clock 2 - outputs asserted on rising edge
##
##.DEFINE LGMOD.
##.NUMOUT 2
##.NUMINP 4
##.NUMMEM 5
##.*BEGIN
##
## Outputs
##
##      define (LATCH, LVOP1)
##      define (EXEC, LVOP2)
##
## Inputs
##
##      define (WORD, LVIN1)
##      define (SAMPLE, LVIN2)
##      define (PHI1, LVIN3)
##      define (PHI2, LVIN4)
##
## Memories
##
##      define (NEXTLATCH, IVAR1)
##      define (NEXTEIEC, IVAR2)
##      define (LASTPHI1, IVAR3)
##      define (STATE, IVAR4)
##      define (NEXTSTATE, IVAR5)
##

```

```

## FSM States
##
    define (IDLEST, 0)
    define (WORD1ST, 2)
    define (WORD2ST, 3)
    define (WORD3ST, 4)
    define (WORD4ST, 5)
    define (WORD5ST, 6)
    define (WORD6ST, 7)
    define (WORD7ST, 8)
    define (WORD8ST, 9)
    define (WORD9ST, 10)
    define (WORD10ST, 11)

##
## Executable Model
##
    if (PHI1==ONE & LASTPHI1!=ONE) {

        if (STATE==IDLEST) {
            if (SAMPLE==ONE & WORD==ONE) {
                NEXTLATCH = ONE
                NEXTEDEC = ONE
                NEXTSTATE = WORD1ST
            }
        }
        if (STATE==WORD1ST) {
            NEXTLATCH = ZERO
            NEXTEDEC = ONE
            if (WORD==ONE) {
                NEXTSTATE = WORD2ST
            }
        }
        if (STATE==WORD2ST) {
            NEXTEDEC = ONE
            if (WORD==ONE) {
                NEXTSTATE = WORD3ST
            }
        }
        if (STATE==WORD3ST) {
            NEXTEDEC = ONE
            if (WORD==ONE) {
                NEXTSTATE = WORD4ST
            }
        }
        if (STATE==WORD4ST) {
            NEXTEDEC = ONE
            if (WORD==ONE) {
                NEXTSTATE = WORD5ST
            }
        }
        if (STATE==WORD5ST) {
            NEXTEDEC = ONE

```

```

        if (WORD==ONE) {
            NEXTSTATE = WORD6ST
        }
    }
    if (STATE==WORD6ST) {
        NEXTEIEC = ONE
        if (WORD==ONE) {
            NEXTSTATE = WORD7ST
        }
    }
    if (STATE==WORD7ST) {
        NEXTEIEC = ONE
        if (WORD==ONE) {
            NEXTSTATE = WORD8ST
        }
    }
    if (STATE==WORD8ST) {
        NEXTEIEC = ONE
        if (WORD==ONE) {
            NEXTSTATE = WORD9ST
        }
    }
    if (STATE==WORD9ST) {
        NEXTEIEC = ONE
        if (WORD==ONE) {
            NEXTSTATE = WORD10ST
        }
    }
    if (STATE==WORD10ST) {
        NEXTEIEC = ONE
        if (WORD==ONE) {
            NEXTEIEC = ZERO
            NEXTSTATE = IDLEST
        }
    }
}

if (PHI2==ONE) {
    LATCH = NEXTLATCH
    EXEC = NEXTEIEC
    STATE = NEXTSTATE
}
LASTPHI1 = PHI1
.END

##-----
##
## COMPONENT      : Finite state machine to supply control signals
##                  to the coefficient calculation block
##
## FUNCTION       : This FSM generates the EXEC and WORD signals for
##                  the coefficient calculation block in response to
##                  the frame clock and the EXEC and WORD signals

```

```

**                      from the recursive autocorrelator.
**
** MODEL STATEMENT : .model fsm2 lgmod : model=32
**
** INSTANTIATION    : <name> EXECO,WORDO,EXECI,WORDI,FRAME,PHI1,PHI2 fsm2
**
** OUTPUTS          : EXECO - Data transfer synchronization signal
**                      for coeff. calculation block
**                      WORDO - Word sync. signal for coeff. calc. block
**
** INPUTS           : EXECI - Data transfer sync. pulse from autocorr.
**                      WORDI - Word sync. pulse from autocorrelator
**                      FRAME - Frame clock
**                      PHI1 - Clock 1
**                      PHI2 - Clock 2 - outputs asserted on rising edge
**
** .DEFINE LGMOD32
** .NUMOUT 2
** .NUMINP 5
** .NUMMEM 5
** .*BEGIN
**
** Outputs
**
**     define (EXECO, LVOP1)
**     define (WORDO, LVOP2)
**
** Inputs
**
**     define (EXECI, LVIN1)
**     define (WORDI, LVIN2)
**     define (FRAME, LVIN3)
**     define (PHI1, LVIN4)
**     define (PHI2, LVIN5)
**
** Memories
**
**     define (NEXTWORD, IVAR1)
**     define (NEXTEXEC, IVAR2)
**     define (LASTPHI1, IVAR3)
**     define (STATE, IVAR4)
**     define (NEXTSTATE, IVAR5)
**
** FSM States
**
**     define (IDLEST, 0)
**     define (WORD1ST, 2)
**     define (WORD2ST, 3)
**     define (WORD3ST, 4)
**     define (WORD4ST, 5)
**     define (WORD5ST, 6)
**     define (WORD6SI, 7)

```

```

define (WORD7ST, 8)
define (WORD8ST, 9)
define (WORD9ST, 10)
define (WORD10ST, 11)
**
** Executable Model
**
  if (PHI1==ONE & LASTPHI1!=ONE) {

    if (STATE==IDLEST) {
      if (FRAME==ONE & WORDI==ONE & EXECI==ONE) {
        NEXTWORD = ONE
NEXTEXEC = ONE
        NEXTSTATE = WORD1ST
      }
    }
    if (STATE==WORD1ST) {
NEXTWORD = WORDI
      NEXTEXEC = EXECI
      if (WORDI==ONE) {
        NEXTSTATE = WORD2ST
      }
    }
    if (STATE==WORD2ST) {
NEXTWORD = WORDI
      NEXTEXEC = EXECI
      if (WORDI==ONE) {
        NEXTSTATE = WORD3ST
      }
    }
    if (STATE==WORD3ST) {
NEXTWORD = WORDI
      NEXTEXEC = EXECI
      if (WORDI==ONE) {
        NEXTSTATE = WORD4ST
      }
    }
    if (STATE==WORD4ST) {
NEXTWORD = WORDI
      NEXTEXEC = EXECI
      if (WORDI==ONE) {
        NEXTSTATE = WORD5ST
      }
    }
    if (STATE==WORD5ST) {
NEXTWORD = WORDI
      NEXTEXEC = EXECI
      if (WORDI==ONE) {
        NEXTSTATE = WORD6ST
      }
    }
    if (STATE==WORD6ST) {

```

```

NEXTWORD = WORDI
  NEXTEXEC = EXECI
  if (WORDI==ONE) {
    NEXTSTATE = WORD7ST
  }
}
if (STATE==WORD7ST) {
NEXTWORD = WORDI
  NEXTEXEC = EXECI
  if (WORDI==ONE) {
    NEXTSTATE = WORD8ST
  }
}
if (STATE==WORD8ST) {
NEXTWORD = WORDI
  NEXTEXEC = EXECI
  if (WORDI==ONE) {
    NEXTSTATE = WORD9ST
  }
}
if (STATE==WORD9ST) {
NEXTWORD = WORDI
  NEXTEXEC = EXECI
  if (WORDI==ONE) {
    NEXTSTATE = WORD10ST
  }
}
if (STATE==WORD10ST) {
NEXTWORD = WORDI
  NEXTEXEC = EXECI
  if (WORDI==ONE) {
    NEXTEXEC = ZERO
    NEXTWORD = ZERO
    NEXTSTATE = IDLEST
  }
}
}
if (PHI2==ONE) {
  WORDO = NEXTWORD
  EXECO = NEXTEXEC
  STATE = NEXTSTATE
}
LASTPHI1 = PHI1
.END

##-----
##
## COMPONENT      : Read data from file, convert to bit-serial format
##
## FUNCTION       : When a rising edge is detected at the READ input
##                  a number I is read from an external file in real
##                  format (-1 < I < +1) and converted to bit-serial

```



```

**          format. The resulting bits are shifted out, LSB
**          first, through the SIGNAL output. A pulse is
**          asserted on the WORD output co-incident with the
**          LSB of SIGNAL.
**
** MODEL STATEMENT : .model readfile lgmod : model=1 PAR1=<integer>
**
** INSTANTIATION   : <name> SIGNAL,WORD,READ,PHI1,PHI2 readfile
**
** OUTPUTS         : SIGNAL - Data word in serial format, LSB first
**                   WORD   - Pulse co-incident with LSB of SIGNAL
**
** INPUTS          : READ   - Rising edge triggers read from file
**                   PHI1   - Clock 1
**                   PHI2   - Clock 2 - outputs asserted on rising edge
**
** PARAMETERS      : PAR1   - FORTRAN unit identifier to read from
**                   N.B. units 0,5,6,7 are used by SPLICE
**
** .DEFINE LGMOD1
** .NUMOUT 2
** .NUMINP 3
** .NUMMEM 6
** .*BEGIN
**
** ## Outputs
**
**     define (SIGNAL, LVOP1)
**     define (WORD, LVOP2)
**
** ## Inputs
**
**     define (READ, LVIN1)
**     define (PHI1, LVIN2)
**     define (PHI2, LVIN3)
**
** ## Parameters
**
**     define (KUNIT, PAR1)
**
** ## Memories
**
**     define (LASTREAD, IVAR1)
**     define (NEXTSIG, IVAR2)
**     define (NEXTWORD, IVAR3)
**     define (INPUT, IVAR4)
**     define (LASTPHI1, IVAR5)
**     define (LASTPHI2, IVAR6)
**
** ## Temporaries
**
**     real RINPUT

```

```

      integer UNIT
**
** Executable model
**
      if (PHI1==ONE & LASTPHI1!=ONE) {  ## When PHI1 goes high
        if (READ==ONE) {                ## When READ is high
          UNIT      = int (RUNIT)
          read (UNIT,1001),RINPUT
1001      FORMAT (E15.7)
          INPUT     = int (RINPUT * 2.0 ** 30)
          NEXTWORD  = ONE
        } else {
          NEXTWORD  = ZERO
        }
        NEXTSIG = mod (INPUT, 2)
        if (NEXTSIG<0) { NEXTSIG = 1 - NEXTSIG }
        else          { NEXTSIG = 1 + NEXTSIG }
        INPUT  = ishft (INPUT, -1)
      }
      if (PHI2==ONE) {                  ## When PHI2 is high
        WORD      = NEXTWORD
        SIGNAL    = NEXTSIG
      }
      LASTPHI1 = PHI1
      LASTPHI2 = PHI2
      LASTREAD = READ
.END

**-----
**
** COMPONENT      : Read data from file, convert to bit-serial format
**
** FUNCTION       : When a rising edge is detected at the READ input
**                  a number X is read from an external file in real
**                  format (-1 < X < +1) and converted to bit-serial
**                  format. The resulting bits are shifted out, LSB
**                  first, through the SIGNAL output. A pulse is
**                  asserted on the WORD output co-incident with the
**                  LSB of SIGNAL.
**
** MODEL STATEMENT : .model readfile lgmod : model=2 PAR1=<integer>
**
** INSTANTIATION   : <name> SIGNAL,WORD,READ,PHI1,PHI2 readfile
**
** OUTPUTS         : SIGNAL - Data word in serial format, LSB first
**                  WORD    - Pulse co-incident with LSB of SIGNAL
**
** INPUTS          : READ   - Rising edge triggers read from file
**                  PHI1    - Clock 1
**                  PHI2    - Clock 2 - outputs asserted on rising edge
**
** PARAMETERS      : PAR1   - FORTRAN unit identifier to read from
**                  N.B. units 0,5,6,7 are used by SPLICE

```

```

##
.DEFINE LGMOD2
.NUMOUT 2
.NUMIMP 3
.NUMMEM 6
.*BEGIN
##
## Outputs
##
    define (SIGNAL, LVOP1)
    define (WORD, LVOP2)
##
## Inputs
##
    define (READ, LVIN1)
    define (PHI1, LVIN2)
    define (PHI2, LVIN3)
##
## Parameters
##
    define (RUNIT, PAR1)
##
## Memories
##
    define (LASTREAD, IVAR1)
    define (NEXTSIG, IVAR2)
    define (NEXTWORD, IVAR3)
    define (INPUT, IVAR4)
    define (LASTPHI1, IVAR5)
    define (LASTPHI2, IVAR6)
##
## Temporaries
##
    real RINPUT
    integer UNIT
##
## Executable model
##
    if (PHI1==ONE & LASTPHI1!=ONE) { ## When PHI1 goes high
        if (READ==ONE) { ## When READ is high
            UNIT = int (RUNIT)
            read (UNIT,1001),RINPUT
1001    FORMAT (E15.7)
            INPUT = int (RINPUT * 2.0 ** 30)
            NEXTWORD = ONE
        } else {
            NEXTWORD = ZERO
        }
        NEXTSIG = mod (INPUT, 2)
        if (NEXTSIG<0) { NEXTSIG = 1 - NEXTSIG }
        else          { NEXTSIG = 1 + NEXTSIG }
        INPUT = ishft (INPUT, -1)
    }

```

```

    }
    if (PHI2==ONE) {
        WORD      = NEXTWORD
        SIGNAL     = NEXTSIG
    }
    LASTPHI1 = PHI1
    LASTPHI2 = PHI2
    LASTREAD = READ
.END

##-----
##
## COMPONENT      : Read data from file, convert to bit-parallel format
##
## FUNCTION       : When a rising edge is detected at the READ input
##                  a number X is read from an external file in integer
##                  format (-2048 < X < +2047) and converted to
##                  bit-parallel format. The resulting bits are
##                  asserted on the SIG0,...,SIG11 outputs.
##
## MODEL STATEMENT : .model readfile lgmod : model=5 PAR1=<integer>
##
## INSTANTIATION   : <name> SIG11,SIG10,SIG9,SIG8,SIG7,SIG6,SIG5,SIG4,$
##                  SIG3,SIG2,SIG1,SIG0,READ,PHI1,PHI2 readfile
##
## OUTPUTS         : SIG11,...,SIG0 - Data word in bit-parallel format,
##                  SIG11 is MSB, SIG0 is LSB
##
## INPUTS          : READ   - Rising edge triggers read from file
##                  PHI1    - Clock 1
##                  PHI2    - Clock 2 - outputs asserted on rising edge
##
## PARAMETERS      : PAR1   - FORTRAN unit identifier to read from
##                  N.B. units 0,5,6,7 are used by SPLICE
##
.DEFINE LGMOD5
.NUMOUT 12
.NUMINP 3
.NUMMEM 15
.*BEGIN
##
## Outputs
##
    define (SIG0, LVOP12)
    define (SIG1, LVOP11)
    define (SIG2, LVOP10)
    define (SIG3, LVOP9)
    define (SIG4, LVOP8)
    define (SIG5, LVOP7)
    define (SIG6, LVOP6)
    define (SIG7, LVOP5)
    define (SIG8, LVOP4)
    define (SIG9, LVOP3)

```

```

define (SIG10, LVOP2)
define (SIG11, LVOP1)
##
## Inputs
##
define (READ, LVIN1)
define (PHI1, LVIN2)
define (PHI2, LVIN3)
##
## Parameters
##
define (RUNIT, PAR1)
##
## Memories
##
define (LASTREAD, IVAR1)
define (LASTPHI1, IVAR2)
define (IINPUT, IVAR3)
define (NITSIG0, IMEM(MPTR+4+0))
define (NITSIG1, IMEM(MPTR+4+1))
define (NITSIG2, IMEM(MPTR+4+2))
define (NITSIG3, IMEM(MPTR+4+3))
define (NITSIG4, IMEM(MPTR+4+4))
define (NITSIG5, IMEM(MPTR+4+5))
define (NITSIG6, IMEM(MPTR+4+6))
define (NITSIG7, IMEM(MPTR+4+7))
define (NITSIG8, IMEM(MPTR+4+8))
define (NITSIG9, IMEM(MPTR+4+9))
define (NITSIG10, IMEM(MPTR+4+10))
define (NITSIG11, IMEM(MPTR+4+11))
##
## Temporaries
##
integer UNIT, SINPUT
##
## Executable model
##
if (READ==ONE & LASTREAD!=ONE) { ## When READ goes high
    UNIT = int (RUNIT)
    read (UNIT,*),IINPUT
}
if (PHI1==ONE & LASTPHI1!=ONE) { ## When PHI1 goes high
    SINPUT = IINPUT
    NITSIG0 = mod (SINPUT, 2)
    if (NITSIG0<0) {NITSIG0 = 1 - NITSIG0} ## Kluge for when IINPUT < 0
    else {NITSIG0 = 1 + NITSIG0}
    SINPUT = ishift (IINPUT, -1)
    NITSIG1 = mod (SINPUT, 2) + 1
    SINPUT = ishift (IINPUT, -2)
    NITSIG2 = mod (SINPUT, 2) + 1
    SINPUT = ishift (IINPUT, -3)
    NITSIG3 = mod (SINPUT, 2) + 1

```

```

        SINPUT  = ishft (IINPUT, -4)
        NXTSIG4 = mod (SINPUT, 2) + 1
        SINPUT  = ishft (IINPUT, -5)
        NXTSIG5 = mod (SINPUT, 2) + 1
        SINPUT  = ishft (IINPUT, -6)
        NXTSIG6 = mod (SINPUT, 2) + 1
        SINPUT  = ishft (IINPUT, -7)
        NXTSIG7 = mod (SINPUT, 2) + 1
        SINPUT  = ishft (IINPUT, -8)
        NXTSIG8 = mod (SINPUT, 2) + 1
        SINPUT  = ishft (IINPUT, -9)
        NXTSIG9 = mod (SINPUT, 2) + 1
        SINPUT  = ishft (IINPUT, -10)
        NXTSIG10 = mod (SINPUT, 2) + 1
        SINPUT  = ishft (IINPUT, -11)
        NXTSIG11 = mod (SINPUT, 2) + 1
    }
##    if (PHI2==ONE) {                                ## When PHI2 is high

        SIG0 = NXTSIG0
        SIG1 = NXTSIG1
        SIG2 = NXTSIG2
        SIG3 = NXTSIG3
        SIG4 = NXTSIG4
        SIG5 = NXTSIG5
        SIG6 = NXTSIG6
        SIG7 = NXTSIG7
        SIG8 = NXTSIG8
        SIG9 = NXTSIG9
        SIG10 = NXTSIG10
        SIG11 = NXTSIG11

    ## }
    LASTPHI1 = PHI1
    LASTREAD = READ

.END

##-----
##
## COMPONENT      : Write data (in bit-serial format) to file
##
## FUNCTION       : When the WORD input is high, that bit and the
##                  the following 31 bits are gathered together and
##                  written to an external file in exponential format.
##
## MODEL STATEMENT : .model write lgmod : model=3 PAR1=<integer>
##
## INSTANTIATION   : <name> DUMMY,SIGNAL,WORD,PHI1,PHI2 write
##
## OUTPUTS         : DUMMY - This has no function. SPLICE seems to
##                  get confused if a block has no outputs
##
## INPUTS          : SIGNAL - Data word in serial format, LSB first

```

```

##          WORD   - Pulse co-incident with LSB of SIGNAL
##          PHI1   - Clock 1
##          PHI2   - Clock 2 - outputs asserted on rising edge
##
## PARAMETERS      : PAR1 - FORTRAN unit identifier to write to
##                  N.B. units 0,5,6,7 are used by SPLICE
##                  PAR2 - *** DEBUG *** flag 0=normal, 1=debug
##
.DEFINE LGMOD3
.NUMOUT 1
.NUMINP 4
.NUMMEM 7
.*BEGIN
##
## Inputs
##
    define (SIGNAL, LVIN1)
    define (WORD, LVIN2)
    define (PHI1, LVIN3)
    define (PHI2, LVIN4)
##
## Memories
##
    define (LASTWORD, IVAR1)
    define (LASTPHI1, IVAR2)
    define (LASTPHI2, IVAR3)
    define (COUNT, IVAR4)
    define (OUTPUT, IVAR5)
    define (LSBTIME, IVAR6)    ## Time at which WORD goes high,
                                ## signifying the LSB of the input signal
    define (WRITEIT, IVAR7)    ## 1 if we should write to the file
##
## Parameters
##
    define (RUNIT, PAR1)
    define (DBGFLG, PAR2)
        define (NORMAL, 0)
        define (DEBUG, 1)
##
## Temporaries
##
    integer UNIT, ISIGNAL
    real ROUTPUT
##
## Executable model
##
    if (WORD==ONE & LASTWORD!=ONE) { ## When WORD goes high
        COUNT = 0
        LSBTIME = SIMTIM
        if (DBGFLG==DEBUG) {
            PRINT *, 'LGMOD3 : LSB at ', SIMTIM
        }
    }

```

```

}
if (PHI1==ONE & LASTPHI1!=ONE) {  ## When PHI1 goes high
  if (SIGNAL==ONE) { ISIGNAL = 1 }
  else                { ISIGNAL = 0 }
  OUTPUT = ishft(OUTPUT, -1) + ishft(ISIGNAL, 31)
  COUNT = COUNT + 1
  if (COUNT==32) {
    ROUTPUT = real(OUTPUT) * 2.0**-30
    UNIT     = int(RUNIT)
    write (UNIT,2),ROUTPUT,LSBTIME
2    format(E15.7,' @ TIME',I14)
    if (DBGFLG==DEBUG) {
      print *,'LGMOD3 : wrote to the output file at',SIMTIM
    }
  }
  if (DBGFLG==DEBUG) {
    PRINT *,'LGMOD3 : ',COUNT,' bits recd, OUTPUT=',OUTPUT,' at ',SIMTIM
  }
}
if (PHI2==ONE & LASTPHI2!=ONE) {  ## When PHI2 goes high
  if (WRITEIT==1) {
    }
  }
LASTWORD = WORD
LASTPHI1 = PHI1
LASTPHI2 = PHI2
.END

```

```

##-----
##
## COMPONENT      : Write data (in bit-serial format) to file
##
## FUNCTION       : When the WORD input is high, that bit and the
##                  the following 31 bits are gathered together and
##                  written to an external file in real format.
##
## MODEL STATEMENT : .model write lgmod : model=4 PAR1=<integer>
##
## INSTANTIATION   : <name> DUMMY,SIGNAL,WORD,PHI1,PHI2 write
##
## OUTPUTS         : DUMMY - This has no function. SPLICE seems to
##                  get confused if a block has no outputs
##
## INPUTS          : SIGNAL - Data word in serial format, LSB first
##                  WORD    - Pulse co-incident with LSB of SIGNAL
##                  PHI1    - Clock 1
##                  PHI2    - Clock 2 - outputs asserted on rising edge
##
## PARAMETERS      : PAR1   - FORTRAN unit identifier to write to
##                  N.B. units 0,5,6,7 are used by SPLICE
##                  PAR2    - *** DEBUG *** flag 0=normal, 1=debug
##
.DEFINE LGMOD4

```



```

.NUMOUT 1
.NUMINP 4
.NUMMEM 7
.*BEGIN
##
## Inputs
##
    define (SIGNAL, LVIN1)
    define (WORD, LVIN2)
    define (PHI1, LVIN3)
    define (PHI2, LVIN4)
##
## Memories
##
    define (LASTWORD, IVAR1)
    define (LASTPHI1, IVAR2)
    define (LASTPHI2, IVAR3)
    define (COUNT, IVAR4)
    define (OUTPUT, IVAR5)
    define (LSBTIME, IVAR6)    ## Time at which WORD goes high,
                                ## signifying the LSB of the input signal
    define (WRITEIT, IVAR7)    ## 1 if we should write to the file
##
## Parameters
##
    define (RUNIT, PAR1)
    define (DBGFLG, PAR2)
        define (NORMAL, 0)
        define (DEBUG, 1)
##
## Temporaries
##
    integer UNIT, ISIGNAL
    real ROUTPUT
##
## Executable model
##
    if (WORD==ONE & LASTWORD!=ONE) { ## When WORD goes high
        COUNT = 0
        LSBTIME = SIMTIM
        if (DBGFLG==DEBUG) {
            PRINT *, 'LGMOD4 : LSB at ', SIMTIM
        }
    }
    if (PHI1==ONE & LASTPHI1!=ONE) { ## When PHI1 goes high
        if (SIGNAL==ONE) { ISIGNAL = 1 }
        else { ISIGNAL = 0 }
        OUTPUT = ishft(OUTPUT, -1) + ishft(ISIGNAL, 31)
        COUNT = COUNT + 1
        if (COUNT==32) {
            ROUTPUT = real(OUTPUT) * 2.0**-30
            UNIT = int(RUNIT)
        }
    }

```

```

2      write (UNIT,2),ROUTPUT,LSBTIME
      format(E15.7,' @ TIME',I14)
      if (DBGFLG==DEBUG) {
        print *, 'LGMOD4 : wrote to the output file at',SIMTIM
      }
    }
    if (DBGFLG==DEBUG) {
      PRINT *, 'LGMOD4 : ',COUNT,' bits recd, OUTPUT=',OUTPUT,' at ',SIMTIM
    }
  }
  if (PHI2==ONE & LASTPHI2!=ONE) {  ## When PHI2 goes high
    if (WRITEIT==1) {
    }
  }
  LASTWORD = WORD
  LASTPHI1 = PHI1
  LASTPHI2 = PHI2
.END

```

```

##-----
##
## COMPONENT      : Write data (in bit-parallel format) to file
##
## FUNCTION       : When the WRITE input goes high, the sixteen
##                  bit input is converted to real (-1 < s < +1)
##                  and written to an external file.
##
## MODEL STATEMENT : .model write lgmod : model=6 PAR1=<integer>
##
## INSTANTIATION   : <name> DUMMY,SIG15,SIG14,SIG13,SIG12,SIG11,SIG10,$
##                  SIG9,SIG8,SIG7,SIG6,SIG5,SIG4,SIG3,SIG2,$
##                  SIG1,SIG0,WORD,PHI1,PHI2 write
##
## OUTPUTS        : DUMMY - This has no function. SPLICE seems to
##                  get confused if a block has no outputs
##
## INPUTS         : SIG15,...,SIG0 - Data word in parallel format
##                  SIG15 is MSB, SIG0 is LSB
##                  WRITE - Write occurs on rising edge of WRITE
##                  PHI1  - Clock 1
##                  PHI2  - Clock 2 - outputs asserted on rising edge
##
## PARAMETERS     : PAR1 - FORTRAN unit identifier to write to
##                  N.B. units 0,5,6,7 are used by SPLICE
##                  PAR2 - if 0, write X to the file. If 1, write
##                  -X to the file.
##
##
.DEFINE LGMOD6
.NUMOUT 1
.NUMINP 19
.NUMMEM 5
.*BEGIN
##

```

```

## Inputs
##
  define (SIG0, LVIN16)
  define (SIG1, LVIN15)
  define (SIG2, LVIN14)
  define (SIG3, LVIN13)
  define (SIG4, LVIN12)
  define (SIG5, LVIN11)
  define (SIG6, LVIN10)
  define (SIG7, LVIN9)
  define (SIG8, LVIN8)
  define (SIG9, LVIN7)
  define (SIG10, LVIN6)
  define (SIG11, LVIN5)
  define (SIG12, LVIN4)
  define (SIG13, LVIN3)
  define (SIG14, LVIN2)
  define (SIG15, LVIN1)
  define (WORD, LVIN17)
  define (PHI1, LVIN18)
  define (PHI2, LVIN19)

##
## Memories
##
  define (LASTWORD, IVAR1)
  define (LASTPHI1, IVAR2)
  define (LASTPHI2, IVAR3)
  define (DOIT, IVAR4)
  define (OUTPUT, IVAR5)

##
## Parameters
##
  define (RUNIT, PAR1)
  define (NEGATE, PAR2)

##
## Temporaries
##
  integer UNIT, ISIGNAL
  real ROUTPUT

##
## Executable model
##
  if (WORD==ONE & LASTWORD!=ONE) { ## When WORD goes high
    DOIT = 1
  }
  if (PHI1==ONE & LASTPHI1!=ONE) { ## When PHI1 goes high
    if (SIG15==ONE) { OUTPUT = 2**17 - 1 } ## Get lots of 1s in
    else { OUTPUT = 0 } ## the word, if necc.
    OUTPUT = ishft (OUTPUT, 1) + SIG14 - 1
    OUTPUT = ishft (OUTPUT, 1) + SIG13 - 1
    OUTPUT = ishft (OUTPUT, 1) + SIG12 - 1
    OUTPUT = ishft (OUTPUT, 1) + SIG11 - 1
  }

```

```

        OUTPUT = ishft (OUTPUT, 1) + SIG10 - 1
        OUTPUT = ishft (OUTPUT, 1) + SIG9 - 1
        OUTPUT = ishft (OUTPUT, 1) + SIG8 - 1
        OUTPUT = ishft (OUTPUT, 1) + SIG7 - 1
        OUTPUT = ishft (OUTPUT, 1) + SIG6 - 1
        OUTPUT = ishft (OUTPUT, 1) + SIG5 - 1
        OUTPUT = ishft (OUTPUT, 1) + SIG4 - 1
        OUTPUT = ishft (OUTPUT, 1) + SIG3 - 1
        OUTPUT = ishft (OUTPUT, 1) + SIG2 - 1
        OUTPUT = ishft (OUTPUT, 1) + SIG1 - 1
        OUTPUT = ishft (OUTPUT, 1) + SIG0 - 1
    }
    if (PHI2==ONE & LASTPHI2!=ONE) { ## When PHI2 goes high
        if (DOIT==1) {
            ROUTPUT = real(OUTPUT) * 2.0**-15
            if (NEGATE==1) { ROUTPUT = -ROUTPUT }
            UNIT = int(RUNIT)
            write (UNIT,2),ROUTPUT,SIMTIM
2          format(F14.10,' @ TIME',I14)
            DOIT = 0
        }
    }
    LASTWORD = WORD
    LASTPHI1 = PHI1
    LASTPHI2 = PHI2
.END

##-----
##
## COMPONENT      : Fixed-coefficient bit-serial multiplier
##
## FUNCTION       : Accepts a data word in bit-serial format, and
##                  a synchronization pulse co-incident with the
##                  LSB of the data word. Produces the product of
##                  the input data word and the parameter PAR1
##                  (determined from the .model statement)
##
## MODEL STATEMENT : .model fcmult lgmod : model=11 PAR1=<coefficient>
##
## INSTANTIATION   : <name> PRODUCT,DATA0,WORD0,DATAI,WORDI,$
##                  PHI1,PHI2 fcmult
##
## OUTPUTS        : PRODUCT - DATAI * PAR1, delayed one word time
##                  DATA0   - DATAI, delayed one word time
##                  WORD0    - Pulse co-incident with LSB of PRODUCT
##
## INPUTS         : DATAI   - Input data word, LSB first
##                  WORDI    - Pulse co-incident with LSB of DATAI
##                  PHI1     - Clock 1
##                  PHI2     - Clock 2 - outputs asserted on rising edge
##
## PARAMETERS     : PAR1    - Multiplier coefficient (-1 < PAR1 < 1)
##

```

```

.DEFINE LGMOD11
.NUMOUT 3
.NUMINP 4
.NUMMEM 9
.*BEGIN
##
## Outputs
##
    define (PRODUCT, LVOP1)
    define (DATAO, LVOP2)
    define (WORDO, LVOP3)
##
## Inputs
##
    define (DATAI, LVIN1)
    define (WORDI, LVIN2)
    define (PHI1, LVIN3)
    define (PHI2, LVIN4)
##
## Parameters
##
    define (COEFFICIENT, PAR1)
##
## Memories
##
    define (IIN, IVAR1)
    define (RIN, IVAR2)
    define (LASTPHI1, IVAR3)
    define (XOUT, IVAR4)
    define (ROUT, IVAR5)
    define (PPOUT, IVAR6)
    define (NEXTDATAO, IVAR7)
    define (NEXTWORDO, IVAR8)
    define (NEXTPROD, IVAR9)
##
## Executable model
##
    if (PHI1==ONE & LASTPHI1!=ONE) {      ## When PHI1 goes high
## Read in the current inputs
        if (DATAI!=ONE) { DATAI = ZERO }
        if (WORDI!=ONE) { WORDI = ZERO }
        IIN = ishft (IIN, -1) + ishft ((DATAI-1), 31)
        RIN = ishft (RIN, -1) + ishft ((WORDI-1), 31)
## Generate the next set of outputs
        XOUT = ishft (XOUT, -1)
        ROUT = ishft (ROUT, -1)
        PPOUT = ishft (PPOUT, -1)
##
        NEXTDATAO = mod (XOUT, 2)
        NEXTDATAO = mod (XIN, 2)
        if (NEXTDATAO < 0) { NEXTDATAO = 1 - NEXTDATAO }
        else { NEXTDATAO = 1 + NEXTDATAO }
##
        NEXTWORDO = mod (ROUT, 2)

```

```

NEXTWORDO = mod (RIN, 2)
if (NEXTWORDO < 0) { NEXTWORDO = 1 - NEXTWORDO }
else { NEXTWORDO = 1 + NEXTWORDO }
## If we've received an entire word at the input, do the multiply
## PRINT *, '** DEBUG ** RIN=',RIN,' @',SINTIM
    if (RIN == 1) {
        XOUT = XIN
        ROUT = RIN
        PPOUT = XOUT * COEFFICIENT
    ## PRINT *, '** DEBUG **',XOUT,' * ',COEFFICIENT,' = ',PPOUT,' @ ',SINTIM
    }
    NEXTPROD = mod (PPOUT, 2)
    if (NEXTPROD < 0) { NEXTPROD = 1 - NEXTPROD }
    else { NEXTPROD = 1 + NEXTPROD }
}
if (PHI2==ONE) {
    DATAO = NEXTDATAO
    WORDO = NEXTWORDO
    PRODUCT = NEXTPROD
}
LASTPHI1 = PHI1

.END

##-----
##
## COMPONENT      : Parallel-coefficient bit-serial multiplier
##
## FUNCTION       : Accepts a data word in bit-serial format, and
##                  a synchronization pulse co-incident with the
##                  LSB of the data word. Produces the product of
##                  the input data word and the coefficient present
##                  at the inputs C11,...,C0 (in bit-parallel
##                  format).
##
## MODEL STATEMENT : .model spmult lgmod : model=12
##
## INSTANTIATION   : <name> PRODUCT,DATAO,WORDO,DATAI,WORDI,$
##                  C11,C10,C9,C8,C7,C6,C5,C4,C3,C2,C1,C0,$
##                  PHI1,PHI2 spmult
##
## OUTPUTS        : PRODUCT - DATAI * coefficient, delayed one word time
##                  DATAO   - DATAI, delayed one word time
##                  WORDO    - Pulse co-incident with LSB of PRODUCT
##
## INPUTS         : DATAI   - Input data word, LSB first
##                  WORDI    - Pulse co-incident with LSB of DATAI
##                  C11,...,C0 - Coefficient in bit-parallel format
##                          C11 is MSB, C0 is LSB
##                  PHI1     - Clock 1
##                  PHI2     - Clock 2 - outputs asserted on rising edge
##
##.DEFINE LGMOD12

```

```

.NUMOUT 3
.NUMINP 16
.NUMMEM 9
.*BEGIN
##
## Outputs
##
    define (PRODUCT, LVOP1)
    define (DATA0, LVOP2)
    define (WORD0, LVOP3)
##
## Inputs
##
    define (DATAI, LVIN1)
    define (WORDI, LVIN2)
    define (C11, LVIN3)
    define (C10, LVIN4)
    define (C9, LVIN5)
    define (C8, LVIN6)
    define (C7, LVIN7)
    define (C6, LVIN8)
    define (C5, LVIN9)
    define (C4, LVIN10)
    define (C3, LVIN11)
    define (C2, LVIN12)
    define (C1, LVIN13)
    define (C0, LVIN14)
    define (PHI1, LVIN15)
    define (PHI2, LVIN16)
##
## Memories
##
    define (XIN, IVAR1)
    define (RIN, IVAR2)
    define (LASTPHI1, IVAR3)
    define (XOUT, IVAR4)
    define (ROUT, IVAR5)
    define (PPOUT, IVAR6)
    define (NEXTDATA0, IVAR7)
    define (NEXTWORD0, IVAR8)
    define (NEXTPROD, IVAR9)
##
## Temporaries
##
    integer COEFFT
##
## Executable model
##
    if (PHI1==ONE & LASTPHI1!=ONE) {      ## When PHI1 goes high
## Read in the current inputs
        if (DATAI!=ONE) { DATAI = ZERO }
        if (WORDI!=ONE) { WORDI = ZERO }

```

```

        XIN = ishft (XIN, -1) + ishft ((DATAI-1), 31)
        RIN = ishft (RIN, -1) + ishft ((WORDI-1), 31)
    ## Generate the next set of outputs
        XOUT = ishft (XOUT, -1)
        ROUT = ishft (ROUT, -1)
        PPOUT = ishft (PPOUT, -1)
        NEXTDATAO = mod (XIN, 2)
        if (NEXTDATAO < 0) { NEXTDATAO = 1 - NEXTDATAO }
        else { NEXTDATAO = 1 + NEXTDATAO }
        NEXTWORDO = mod (RIN, 2)
        if (NEXTWORDO < 0) { NEXTWORDO = 1 - NEXTWORDO }
        else { NEXTWORDO = 1 + NEXTWORDO }
    ## If we've received an entire word at the input, convert
    ## the parallel coefficient to an integer and do the multiply.
        if (NEXTWORDO == ONE) {
            XOUT = XIN
            ROUT = RIN
            COEFFT = 0 + ishft (C11-1,31) + ishft (C10-1,30)
            COEFFT = COEFFT + ishft (C9-1, 29) + ishft (C8-1, 28)
            COEFFT = COEFFT + ishft (C7-1, 27) + ishft (C6-1, 26)
            COEFFT = COEFFT + ishft (C5-1, 25) + ishft (C4-1, 24)
            COEFFT = COEFFT + ishft (C3-1, 23) + ishft (C2-1, 22)
            COEFFT = COEFFT + ishft (C1-1, 21) + ishft (C0-1, 20)
            PPOUT = XOUT * (real(COEFFT) * 2.0 ** -31)
        }
        NEXTPROD = mod (PPOUT, 2)
        if (NEXTPROD < 0) { NEXTPROD = 1 - NEXTPROD }
        else { NEXTPROD = 1 + NEXTPROD }
    }
    if (PHI2==ONE) {
        DATAO = NEXTDATAO
        WORDO = NEXTWORDO
        PRODUCT = NEXTPROD
    }
    LASTPHI1 = PHI1

.END

##-----
##
## COMPONENT      : Word delay
##
## FUNCTION       : Accepts a data stream in bit-serial format, and
##                  delays it one word time (32 bits)
##
## MODEL STATEMENT : .model wddelay lgmod : model=13
##
## INSTANTIATION   : <name> DATAO,DATAI,PHI1,PHI2 wddelay
##
## OUTPUTS        : DATAO - DATAI, delayed one word time
##
## INPUTS         : DATAI - Input data word, LSB first
##                  PHI1 - Clock 1

```



```

##          PHI2          - Clock 2 - outputs asserted on rising edge
##
.DEFINE LGMOD13
.NUMOUT 1
.NUMINP 3
.NUMMEM 4
.*BEGIN
##
## Outputs
##
    define (DATA0, LVOP1)
##
## Inputs
##
    define (DATAI, LVIN1)
    define (PHI1, LVIN2)
    define (PHI2, LVIN3)
##
## Memories
##
    define (WORD, IVAR1)
    define (LASTPHI1, IVAR2)
    define (NEXTDATA0, IVAR3)
    define (LASTPHI2, IVAR4)
##
## Executable model
##
    if (PHI1==ONE & LASTPHI1!=ONE) {      ## When PHI1 goes high
## Read in the current input
        if (DATAI!=ONE) { DATAI = ZERO }
        WORD = ishft (WORD, -1) + ishft ((DATAI-1), 31)
## Generate the next output
        NEXTDATA0 = mod (WORD, 2)
        if (NEXTDATA0 < 0) { NEXTDATA0 = 1 - NEXTDATA0 }
        else                { NEXTDATA0 = 1 + NEXTDATA0 }
    }
    if (PHI2==ONE & LASTPHI2!=ONE) {
        DATA0 = NEXTDATA0
    }
    LASTPHI1 = PHI1
    LASTPHI2 = PHI2

.END

##-----
##
## COMPONENT          : Clock buffer
##
## FUNCTION           : Enable/disable the clock
##
## MODEL STATEMENT    : .model clkbuf lgmod : model=14
##
## INSTANTIATION      : <name> PHI10,PHI20,PHI1I,PHI2I,ENABLE clkbuf

```

```

##
## OUTPUTS      : PHI10 - Clock phase 1 output
##               PHI20 - Clock phase 2 output
##
## INPUTS       : PHI1I  - Clock phase 1 input
##               PHI2I  - Clock phase 2 input
##               ENABLE - Active high enable for clocks
##
.DEFINE LGMOD14
.NUMOUT 2
.NUMINP 3
.*BEGIN
##
## Outputs
##
    define (PHI10, LVOP1)
    define (PHI20, LVOP2)
##
## Inputs
##
    define (PHI1I, LVIN1)
    define (PHI2I, LVIN2)
    define (ENABLE, LVIN3)
##
## Executable model
##
    if (ENABLE == ONE) {
        PHI10 = PHI1I
        PHI20 = PHI2I
    } else {
        PHI10 = ZERO
        PHI20 = ZERO
    }
}

.END

##-----
##
## COMPONENT      : Bit delay
##
## FUNCTION       : Delay the input one bit time
##
## MODEL STATEMENT : .model bitdelay lgmod : model=15
##
## INSTANTIATION  : <name> BIT0,BIT1,PHI1,PHI2 bitdelay
##
## OUTPUTS       : BIT0 - Output = BIT1, delayed one bit time
##
## INPUTS        : BIT1 - Input bit (to be delayed)
##                PHI1 - Clock phase 1
##                PHI2 - Clock phase 2
##
.DEFINE LGMOD15

```

```

.NUMOUT 1
.NUMINP 3
.NUMMEM 2
.*BEGIN
**
** Outputs
**
    define (BITO, LVOP1)
**
** Inputs
**
    define (BITI, LVIN1)
    define (PHI1, LVIN2)
    define (PHI2, LVIN3)
**
** Memories
**
    define (TEMP, IVAR1)
    define (LASTPHI1, IVAR2)
**
** Executable model
**
    if (PHI1==ONE & LASTPHI1!=ONE) { ** When PHI1 goes high
        if (BITI!=ONE) { BITI = ZERO }
        TEMP = BITI
    }
    if (PHI2==ONE) { ** When PHI2 is high
        BITO = TEMP
    }
    LASTPHI1 = PHI1
.END

**-----
**
** COMPONENT      : adder
**
** FUNCTION       : Serial adder/subtractor
**
** MODEL STATEMENT : .model adder lgmod : model=16
**
** INSTANTIATION  : <name> SUM,WORD,X,Y,WDIN,MODE,PHI1,PHI2 adder
**
** OUTPUTS       : SUM   - X+Y or X-Y in serial format, LSB first
**                 WORD   - Pulse co-incident with LSB of SUM
**
** INPUTS        : X     - Input #1 in serial format, LSB first
**                 Y     - Input #2 in serial format, LSB first
**                 WDIN  - Pulse co-incident with LSB of X and Y
**                 MODE  - Select ADD (MODE=0) or SUBTRACT (MODE=1)
**                 PHI1  - Clock 1 - inputs latched on rising edge
**                 PHI2  - Clock 2 - outputs asserted on rising edge
**
** PARAMETERS    : none

```

```

##
.DEFINE LGMOD16
.NUMOUT 2
.NUMINP 6
.NUMMEM 4
.*BEGIN
##
## Outputs
##
    define (SUM, LVOP1)
    define (WORD, LVOP2)
##
## Inputs
##
    define (X, LVIN1)
    define (Y, LVIN2)
    define (WDIN, LVIN3)
    define (MODE, LVIN4)
        define (ADD, ZERO)      ## A MODE of ZERO means perform addition
        define (SUBTRACT, ONE)  ## A MODE of ONE means perform subtraction
    define (PHI1, LVIN5)
    define (PHI2, LVIN6)
##
## Memories
##
    define (CARRY, IVAR1)
    define (LASTPHI1, IVAR2)
    define (CYOUT, IVAR3)
    define (LASTPHI2, IVAR4)
##
## Temporaries
##
    integer YBIT, TMP1, TMP2, TMP3, TMP4, TMP5
##
## Executable model
##
    if (PHI1==ONE & LASTPHI1!=ONE) {    ## when PHI1 goes high
        if (X!=ONE) { X = ZERO }
        if (Y!=ONE) { Y = ZERO }
        if (WDIN!=ONE) { WDIN = ZERO }
        if (MODE!=ONE) { MODE = ZERO }
        if (WDIN==ONE) {
            if (MODE==ADD) { CYOUT = ZERO }
            else          { CYOUT = ONE }
        }
    }
    if (PHI2==ONE & LASTPHI2!=ONE) {    ## when PHI2 goes high
        CARRY = CYOUT
    }
## Combinational adder
YBIT      = lxor (Y, MODE)
TMP1      = lxor (X, YBIT)

```

```

SUM      = lxor (CARRY, TMP1)
TMP2     = land (X, YBIT)
TMP3     = land (X, CARRY)
TMP4     = land (YBIT, CARRY)
TMP5     = lor (TMP2, TMP3)
CYOUT    = lor (TMP4, TMP5)
WORD     = WDIN

```

```

LASTPHI1 = PHI1
LASTPHI2 = PHI2

```

```

.END

```

```

##-----
##
## COMPONENT      : latch
##
## FUNCTION       : Bit latch
##
## MODEL STATEMENT : .model latch lgmod : model=17
##
## INSTANTIATION  : <name> OUT,IN,TRIGGER latch
##
## OUTPUTS        : OUT      - Bit output
##
## INPUTS         : IN        - Bit input
##                  TRIGGER - Latch is transparent when high
##
## PARAMETERS     : none
##
.DEFINE LGMOD17
.NUMOUT 1
.NUMINP 2
.NUMMEM 1
.*BEGIN
##
## Outputs
##
    define (OUT, LVOP1)
##
## Inputs
##
    define (IN, LVIN1)
    define (TRIGGER, LVIN2)
##
## Memories
##
    define (MEM, IVAR1)
##
## Executable model
##
    if (TRIGGER==ONE) {
        if (IN==ONE) { OUT = ONE }

```

```

    else          { OUT = ZERO }
  }

.END

##-----
##
## COMPONENT      : Serial coefficient, serial data multiplier
##
## FUNCTION       : Accepts a data word in bit-serial format,
##                  a coefficient also in bit-serial format, and
##                  a synchronization pulse co-incident with the
##                  LSB of the words. Produces the product of
##                  the input data word and the coefficient.
##
## MODEL STATEMENT : .model smult lgmod : model=19
##
## INSTANTIATION   : <name> PRODUCT,DATA0,COEFO,WORD0,DATAI,$
##                  COEFI,WORDI,PHI1,PHI2 smult
##
## OUTPUTS        : PRODUCT - DATAI * PAR1, delayed one word time
##                  DATA0  - DATAI, delayed one word time
##                  COEFO   - COEFI, delayed one word time
##                  WORD0   - Pulse co-incident with LSB of PRODUCT
##
## INPUTS         : DATAI  - Input data word, LSB first
##                  COEFI   - Input coefficient word, LSB first
##                  WORDI   - Pulse co-incident with LSB of DATAI
##                          and COEFI
##                  PHI1    - Clock 1
##                  PHI2    - Clock 2 - outputs asserted on rising edge
##
##.DEFINE LGMOD19
##.NUMOUT 4
##.NUMINP 5
##.NUMMEM 9
##.*BEGIN
##
## Outputs
##
##    define (PRODUCT, LVOP1)
##    define (DATA0, LVOP2)
##    define (COEFO, LVOP3)
##    define (WORD0, LVOP4)
##
## Inputs
##
##    define (DATAI, LVIN1)
##    define (COEFI, LVIN2)
##    define (WORDI, LVIN3)
##    define (PHI1, LVIN4)
##    define (PHI2, LVIN5)
##

```

```

## Memories
##
  define (XIN, IVAR1)
  define (RIN, IVAR2)
  define (CIN, IVAR3)
  define (LASTPHI1, IVAR4)
  define (PPOUT, IVAR5)
  define (NEXTDATAO, IVAR6)
  define (NEXTCOEFO, IVAR7)
  define (NEXTWORDO, IVAR8)
  define (NEXTPROD, IVAR9)
##
## Executable model
##
  if (PHI1==ONE & LASTPHI1!=ONE) {      ## When PHI1 goes high
## Read in the current inputs
    if (DATAI!=ONE) { DATAI = ZERO }
    if (COEFI!=ONE) { COEFI = ZERO }
    if (WORDI!=ONE) { WORDI = ZERO }
    XIN = ishft (XIN, -1) + ishft ((DATAI-1), 31)
    CIN = ishft (CIN, -1) + ishft ((COEFI-1), 31)
    RIN = ishft (RIN, -1) + ishft ((WORDI-1), 31)
## Generate the next set of outputs
    NEXTDATAO = mod (XIN, 2)
    if (NEXTDATAO < 0) { NEXTDATAO = 1 - NEXTDATAO }
    else { NEXTDATAO = 1 + NEXTDATAO }
    NEXTCOEFO = mod (CIN, 2)
    if (NEXTCOEFO < 0) { NEXTCOEFO = 1 - NEXTCOEFO }
    else { NEXTCOEFO = 1 + NEXTCOEFO }
    NEXTWORDO = mod (RIN, 2)
    if (NEXTWORDO < 0) { NEXTWORDO = 1 - NEXTWORDO }
    else { NEXTWORDO = 1 + NEXTWORDO }
## If we've received an entire word at the input, do the multiply
    if (NEXTWORDO == ONE) {
      PPOUT = XIN * (real(CIN) * 2.0 ** -30)
    }
    NEXTPROD = mod (PPOUT, 2)
    PPOUT = ishft (PPOUT, -1)
    if (NEXTPROD < 0) { NEXTPROD = 1 - NEXTPROD }
    else { NEXTPROD = 1 + NEXTPROD }
  }
  if (PHI2==ONE) {                      ## When PHI2 is high
    DATAO = NEXTDATAO
    COEFO = NEXTCOEFO
    WORDO = NEXTWORDO
    PRODUCT = NEXTPROD
  }
  LASTPHI1 = PHI1

```

```

.END

```

```

##-----
##

```

```

## COMPONENT      : Serial divisor, serial dividend divider
##
## FUNCTION       : Accepts a dividend in bit-serial format,
##                 a divisor also in bit-serial format, and
##                 a synchronization pulse co-incident with the
##                 LSB of the words. Produces the quotient of
##                 the dividend and the divisor (i.e.
##                 quotient = dividend/divisor) in bit-parallel
##                 format
##
## MODEL STATEMENT : .model divider lgmod : model=20
##
## INSTANTIATION   : <name> Q15,Q14,Q13,Q12,Q11,Q10,Q9,Q8,Q7,Q6,$
##                 Q5,Q4,Q3,Q2,Q1,Q0,DIVIDEND,DIVISOR,$
##                 WORDI,PHI1,PHI2 divider
##
## OUTPUTS         : Q15, ..., Q0 - DIVIDEND/DIVISOR
##
## INPUTS          : DIVIDEND - Dividend, LSB first
##                 DIVISOR  - Divisor, LSB first
##                 WORDI    - Pulse co-incident with LSB of
##                           DIVIDEND and DIVISOR
##                 PHI1     - Clock 1
##                 PHI2     - Clock 2 - outputs asserted on rising edge
##
## PARAMETERS      : PAR1    - 1=DEBUG, 0=NORMAL
##
.DEFINE LGMOD20
.NUMOUT 16
.NUMINP 5
.NUMMEM 20
.*BEGIN
##
## Outputs
##
define (Q15, LVOP1)
define (Q14, LVOP2)
define (Q13, LVOP3)
define (Q12, LVOP4)
define (Q11, LVOP5)
define (Q10, LVOP6)
define (Q9, LVOP7)
define (Q8, LVOP8)
define (Q7, LVOP9)
define (Q6, LVOP10)
define (Q5, LVOP11)
define (Q4, LVOP12)
define (Q3, LVOP13)
define (Q2, LVOP14)
define (Q1, LVOP15)
define (Q0, LVOP16)
##

```



```

## Inputs
##
    define (DIVIDEND, LVIN1)
    define (DIVISOR, LVIN2)
    define (WORDI, LVIN3)
    define (PHI1, LVIN4)
    define (PHI2, LVIN5)
##
## Memories
##
    define (LASTPHI1, IVAR1)
    define (TOP, IVAR2)
    define (BOTTOM, IVAR3)
    define (WORD, IVAR4)
    define (NEXTQ0, IVAR5)
    define (NEXTQ1, IVAR6)
    define (NEXTQ2, IVAR7)
    define (NEXTQ3, IVAR8)
    define (NEXTQ4, IVAR9)
    define (NEXTQ5, IVAR10)
    define (NEXTQ6, IVAR11)
    define (NEXTQ7, IVAR12)
    define (NEXTQ8, IVAR13)
    define (NEXTQ9, IVAR14)
    define (NEXTQ10, IVAR15)
    define (NEXTQ11, IVAR16)
    define (NEXTQ12, IVAR17)
    define (NEXTQ13, IVAR18)
    define (NEXTQ14, IVAR19)
    define (NEXTQ15, IVAR20)
##
## Parameters
##
    define (DEBUG20, PAR1)
##
## Temporaries
##
    integer QUOTIENT, TEMP20
##
## Executable model
##
    if (PHI1==ONE & LASTPHI1!=ONE) {      ## When PHI1 goes high
## Read in the current inputs
        if (DIVIDEND!=ONE) { DIVIDEND = ZERO }
        if (DIVISOR!=ONE) { DIVISOR = ZERO }
        if (WORDI!=ONE) { WORDI = ZERO }
        TOP = ishft (TOP, -1) + ishft ((DIVIDEND-1), 31)
        BOTTOM = ishft (BOTTOM, -1) + ishft ((DIVISOR-1), 31)
        WORD = ishft (WORD, -1) + ishft ((WORDI-1), 31)
## If we've received an entire word at the input, do the divide
        if (mod (WORD,2) != 0) {
            *      TEMP20 = BOTTOM / (2 ** 15)

```

```

#         if (TEMP20==0) {
#             if (BOTTOM<0) {
#                 TEMP20 = -(2**15)
#             } else {
#                 TEMP20 = 2**15
#             }
#         }
#         QUOTIENT = TOP/TEMP20
#         QUOTIENT = int(real(TOP)/real(BOTTOM) * 2.0**15)
#         if (DEBUG20==1) {
#             print 201,real(QUOTIENT)*2.0**-15,real(TOP)*2.0**-30,
real(TEMP20)*2.0**-15,SIMTIM
201         format (E15.7,' = ',E15.7,' / ',E15.7,' @ ',I14)
#         }
#         if (QUOTIENT==2**15) { QUOTIENT = QUOTIENT - 1 }
#         NEXTQ0 = mod (QUOTIENT, 2)
#         if (NEXTQ0 < 0) { NEXTQ0 = 1 - NEXTQ0 }
#         else { NEXTQ0 = 1 + NEXTQ0 }
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ1 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ2 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ3 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ4 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ5 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ6 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ7 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ8 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ9 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ10 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ11 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ12 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ13 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ14 = mod (QUOTIENT, 2) + 1
#         QUOTIENT = ishft (QUOTIENT, -1)
#         NEXTQ15 = mod (QUOTIENT, 2) + 1
#     }
# }
if (PHI2==ONE) {
    Q0 = NEXTQ0
    ## When PHI2 is high

```

```

      Q1 = NEXTQ1
      Q2 = NEXTQ2
      Q3 = NEXTQ3
      Q4 = NEXTQ4
      Q5 = NEXTQ5
      Q6 = NEXTQ6
      Q7 = NEXTQ7
      Q8 = NEXTQ8
      Q9 = NEXTQ9
      Q10 = NEXTQ10
      Q11 = NEXTQ11
      Q12 = NEXTQ12
      Q13 = NEXTQ13
      Q14 = NEXTQ14
      Q15 = NEXTQ15
    }
    LASTPHI1 = PHI1

.END

##-----
##
## COMPONENT      : Constant generator (bit-serial ROM)
##
## FUNCTION       : Generates a series of 16 32-bit words, with
##                  constant values. The constant values are loaded
##                  from a disk file as 16 numbers in real format
##                  (-1 < X < +1).
##
## MODEL STATEMENT : .model constgen lgmod : model=21 PAR1=<integer>
##
## INSTANTIATION   : <name> DATA0,TRIGGER,PHI1,PHI2 constgen
##
## OUTPUTS         : DATA0 - Constant output
##
## INPUTS          : TRIGGER - Resets the generator
##                  PHI1 - Clock 1
##                  PHI2 - Clock 2 - outputs asserted on rising edge
##
## PARAMETERS      : PAR1 - FORTRAN unit identifier to read from.
##                  N.B. units 0,5,6,7 are used by SPLICE
##
.DDEFINE LGMOD21
.NUMOUT 1
.NUMINP 3
.NUMMEM 4
.*BEGIN
##
## Outputs
##
      define (DATA0, LVOP1)
##
## Inputs

```

```

**
    define (TRIGGER, LVIN1)
    define (PHI1, LVIN2)
    define (PHI2, LVIN3)
**
** Memories
**
    define (LASTPHI1, IVAR1)
    define (INDEX, IVAR2)
    define (NEXTDATA0, IVAR3)
    define (LOADED, IVAR4)
**
** Parameters
**
    define (RUNIT, PAR1)
**
** Temporaries
**
    integer I,J,UNIT,ISAMPLE,IBIT
    real RSAMPLE
**
** Statically defined memory
**
    integer MEMORY (512)
    save MEMORY
**
** Executable model
**
    if (SIMTIM==0 & LOADED==0) {
        LOADED = 1
        UNIT = int (RUNIT)
        print *, 'LGMOD21 : Loading ROM from unit',UNIT
        for (I=0; I<16; I=I+1) {
            read (UNIT, 211),RSAMPLE
211      format (E15.7)
            ISAMPLE = int (RSAMPLE * 2.0**30)
            for (J=1; J<=32; J=J+1) {
                IBIT = mod (ISAMPLE, 2)
                if (IBIT < 0) { IBIT = 1 - IBIT }
                else          { IBIT = 1 + IBIT }
                MEMORY (32*I + J) = IBIT
                ISAMPLE = ishft (ISAMPLE, -1)
            }
        }
        INDEX = 1
    }
    if (PHI1==ONE & LASTPHI1!=ONE) {
        if (TRIGGER==ONE) { INDEX = 1 }
        NEXTDATA0 = MEMORY (INDEX)
        if (INDEX==512) { INDEX = 1 }
        else          { INDEX = INDEX + 1 }
    }

```

```

    if (PHI2==ONE) {
        DATA0 = NEXTDATA0
    }
    LASTPHI1 = PHI1
.END

##-----
##
## COMPONENT      : Constant generator (bit-serial ROM)
##
## FUNCTION       : Generates a series of 16 32-bit words, with
##                  constant values. The constant values are loaded
##                  from a disk file as 16 numbers in real format
##                  (-1 < X < +1).
##
## MODEL STATEMENT : .model constgen lgmod : model=22 PAR1=<integer>
##
## INSTANTIATION   : <name> DATA0,TRIGGER,PHI1,PHI2 constgen
##
## OUTPUTS         : DATA0   - Constant output
##
## INPUTS          : TRIGGER - Resets the generator
##                  PHI1     - Clock 1
##                  PHI2     - Clock 2 - outputs asserted on rising edge
##
## PARAMETERS      : PAR1     - FORTRAN unit identifier to read from.
##                  N.B. units 0,5,6,7 are used by SPLICE
##
.DEFINE LGMOD22
.NUMOUT 1
.NUMINP 3
.NUMMEM 4
.*BEGIN
##
## Outputs
##
    define (DATA0, LVOP1)
##
## Inputs
##
    define (TRIGGER, LVIN1)
    define (PHI1, LVIN2)
    define (PHI2, LVIN3)
##
## Memories
##
    define (LASTPHI1, IVAR1)
    define (INDEX, IVAR2)
    define (NEXTDATA0, IVAR3)
    define (LOADED, IVAR4)
##
## Parameters
##

```

```

        define (RUNIT, PAR1)
##
## Temporaries
##
        integer I,J,UNIT,ISAMPLE,IBIT
        real RSAMPLE
##
## Statically defined memory
##
        integer MEMORY (512)
        save MEMORY
##
## Executable model
##
        if (SINTIM==0 & LOADED==0) {
            LOADED = 1
            UNIT = int (RUNIT)
            print *, 'LGMOD22 : Loading ROM from unit',UNIT
            for (I=0; I<16; I=I+1) {
                read (UNIT, 211),RSAMPLE
211         format (E15.7)
                ISAMPLE = int (RSAMPLE * 2.0**30)
                for (J=1; J<=32; J=J+1) {
                    IBIT = mod (ISAMPLE, 2)
                    if (IBIT < 0) { IBIT = 1 - IBIT }
                    else          { IBIT = 1 + IBIT }
                    MEMORY (32*I + J) = IBIT
                    ISAMPLE = ishft (ISAMPLE, -1)
                }
            }
            INDEX = 1
        }
        if (PHI1==ONE & LASTPHI1!=ONE) {
            if (TRIGGER==ONE) { INDEX = 1 }
            NEXTDATA0 = MEMORY (INDEX)
            if (INDEX==512) { INDEX = 1 }
            else          { INDEX = INDEX + 1 }
        }
        if (PHI2==ONE) {
            DATA0 = NEXTDATA0
        }
        LASTPHI1 = PHI1
    .END

##-----
##
## COMPONENT          : Parallel-coefficient bit-serial multiplier
##
## FUNCTION           : Accepts a data word in bit-serial format, and
##                     a synchronization pulse co-incident with the
##                     LSB of the data word. Produces the product of
##                     the input data word and the coefficient present
##                     at the inputs C15,...,C0 (in bit-parallel

```

```

##                               format).
##
## MODEL STATEMENT : .model spmult lgmod : model=23
##
## INSTANTIATION      : <name> PRODUCT,DATA0,WORD0,DATAI,WORDI,$
##                      C15,C14,C13,C12,C11,C10,C9,C8,C7,C6,C5,$
##                      C4,C3,C2,C1,C0,PHI1,PHI2 spmult
##
## OUTPUTS            : PRODUCT - DATAI * coefficient, delayed one word time
##                      DATA0  - DATAI, delayed one word time
##                      WORD0   - Pulse co-incident with LSB of PRODUCT
##
## INPUTS              : DATAI   - Input data word, LSB first
##                      WORDI     - Pulse co-incident with LSB of DATAI
##                      C15,...,C0 - Coefficient in bit-parallel format
##                               C11 is MSB, C0 is LSB
##                      PHI1      - Clock 1
##                      PHI2      - Clock 2 - outputs asserted on rising edge
##
.DEFINE LGMOD23
.NUMOUT 3
.NUMINP 20
.NUMMEM 9
.*BEGIN
##
## Outputs
##
    define (PRODUCT, LVOP1)
    define (DATA0, LVOP2)
    define (WORD0, LVOP3)
##
## Inputs
##
    define (DATAI, LVIN1)
    define (WORDI, LVIN2)
    define (C15, LVIN3)
    define (C14, LVIN4)
    define (C13, LVIN5)
    define (C12, LVIN6)
    define (C11, LVIN7)
    define (C10, LVIN8)
    define (C9, LVIN9)
    define (C8, LVIN10)
    define (C7, LVIN11)
    define (C6, LVIN12)
    define (C5, LVIN13)
    define (C4, LVIN14)
    define (C3, LVIN15)
    define (C2, LVIN16)
    define (C1, LVIN17)
    define (C0, LVIN18)
    define (PHI1, LVIN19)

```

```

    define (PHI2, LVIN20)
##
## Memories
##
    define (XIN, IVAR1)
    define (RIN, IVAR2)
    define (LASTPHI1, IVAR3)
    define (XOUT, IVAR4)
    define (ROUT, IVAR5)
    define (PPOUT, IVAR6)
    define (NEXTDATA0, IVAR7)
    define (NEXTWORD0, IVAR8)
    define (NEXTPROD, IVAR9)
##
## Temporaries
##
    integer COEFFT
##
## Executable model
##
    if (PHI1==ONE & LASTPHI1!=ONE) {      ## When PHI1 goes high
## Read in the current inputs
        if (DATAI!=ONE) { DATAI = ZERO }
        if (WORDI!=ONE) { WORDI = ZERO }
        XIN = ishft (XIN, -1) + ishft ((DATAI-1), 31)
        RIN = ishft (RIN, -1) + ishft ((WORDI-1), 31)
## Generate the next set of outputs
        XOUT = ishft (XOUT, -1)
        ROUT = ishft (ROUT, -1)
        PPOUT = ishft (PPOUT, -1)
        NEXTDATA0 = mod (XIN, 2)
        if (NEXTDATA0 < 0) { NEXTDATA0 = 1 - NEXTDATA0 }
        else                { NEXTDATA0 = 1 + NEXTDATA0 }
        NEXTWORD0 = mod (RIN, 2)
        if (NEXTWORD0 < 0) { NEXTWORD0 = 1 - NEXTWORD0 }
        else                { NEXTWORD0 = 1 + NEXTWORD0 }
## If we've received an entire word at the input, convert
## the parallel coefficient to an integer and do the multiply.
        if (NEXTWORD0 == ONE) {
            XOUT = XIN
            ROUT = RIN
            COEFFT = 0      + ishft (C15-1,31) + ishft (C14-1,30)
            COEFFT = COEFFT + ishft (C13-1, 29) + ishft (C12-1, 28)
            COEFFT = COEFFT + ishft (C11-1, 27) + ishft (C10-1, 26)
            COEFFT = COEFFT + ishft (C9-1, 25) + ishft (C8-1, 24)
            COEFFT = COEFFT + ishft (C7-1, 23) + ishft (C6-1, 22)
            COEFFT = COEFFT + ishft (C5-1, 21) + ishft (C4-1, 20)
            COEFFT = COEFFT + ishft (C3-1, 19) + ishft (C2-1, 18)
            COEFFT = COEFFT + ishft (C1-1, 17) + ishft (C0-1, 16)
            PPOUT = XOUT * (real(COEFFT) * 2.0 ** -31)
        }
        NEXTPROD = mod (PPOUT, 2)

```



```
      if (NEXTPROD < 0) { NEXTPROD = 1 - NEXTPROD }  
      else { NEXTPROD = 1 + NEXTPROD }  
    }  
    if (PHI2==ONE) {                                     ## When PHI2 is high  
      DATA0 = NEXTDATA0  
      WORD0 = NEXTWORD0  
      PRODUCT = NEXTPROD  
    }  
    LASTPHI1 = PHI1  
  
  .END  
  
  .FINISH
```

Appendix B

SPLICE Netlist for Simulation of Recursive Autocorrelator

SPLICE
AUTOCORRELATOR SIMULATION

```

-----
;
; System-level simulation of the recursive autocorrelator
;
; Copyright 1987, 1988, John W. O'Leary
;
; This file contains:
;   - circuit netlist of the recursive autocorrelator
;   - simulation directives
;   - subcircuit definitions for input register, input correlator,
;     scaling block, recursive filtering block, nonrecursive
;     filtering block
;   - model statements for the models defined in SERIAL-LIB.MOD
;
-----

; Circuit netlist of recursive autocorrelator block
-----

;-- Logic Sources

.model vdd dc : 5
.MODEL WORD LSRC : 0 1 5NS 320NS 0NS 1NS 2NS 9NS 10NS 320NS
.MODEL READ LSRC : 0 1 0NS 4000NS 0NS 1NS 2NS 9NS 10NS 4000NS
.MODEL CLK1 LSRC : 0 1 0NS 10NS 0NS 2NS 3NS 4NS 5NS 10NS
.MODEL CLK2 LSRC : 0 1 0NS 10NS 0NS 6NS 7NS 8NS 9NS 10NS

READ READ READ      ;-- The SAMPLE clock
CLK1 CLK1 CLK1
CLK2 CLK2 CLK2
VDD VDD VDD
CWI CWI WORD

;-- Netlist

;-- read from FORTRAN logical unit 22

pread  DIN11,DIN10,DIN9,DIN8,DIN7,DIN6,DIN5,DIN4,DIN3,DIN2,DIN1,DIN0,0

```

```

      READ,CLK1,CLK2 pread22

fsm DL7CH,C&I,CW1,READ,CLK1,CLK2 fsm1

par2ser CD.,CDI11,CDI10,CDI9,CDI8,CDI7,CDI6,CDI5,CDI4,CDI3,CDI2,CDI1,CDI0,$
      DIN11,DIN10,DIN9,DIN8,DIN7,DIN6,DIN5,DIN4,DIN3,DIN2,DIN1,DIN0,$
      DL7CH',CLK1,CLK2 par2ser
corr CDO,CWO,CEO,CDI,CWI,CEI,CDI0,CDI1,CDI2,CDI3,CDI4,CDI5,CDI6,$
      CLI7,CDI8,CDI9,CDI10,CDI11,CLK1,CLK2 corr

scale1 S1DO,S1WO,S1EO,CDO,CWO,CEO,CLK1,CLK2 scale
recur1 R1DO,S1DO,S1EO,S1WO,CLK1,CLK2 recur
scale2 S2DO,S2WO,S2EO,R1DO,S1WO,S1EO,CLK1,CLK2 scale
recur2 R2DO,S2DO,S2EO,S2WO,CLK1,CLK2 recur
scale3 S3DO,S3WO,S3EO,R2DO,S2WO,S2EO,CLK1,CLK2 scale
recur3 R3DO,S3DO,S3EO,S3WO,CLK1,CLK2 recur

nonrec NDO,NEO,NWO,R3DO,S3EO,S3WO,CLK1,CLK2 nonrecur

;-- write to FORTRAN logical unit 23

$write DUM1,NDO,NWO,CLK1,CLK2 $write23

;-----
; Simulation Directives
;-----

;-- 1ns in simulation = 10ns in real time.

.time 1ns 1920000ns

;-----
; Speech analysis system subcircuits
;-----

;-- Input register

.model par2ser subckt : (SOUT $
      POUT11 POUT10 POUT9 POUT8 POUT7 POUT6 POUT5 POUT4 POUT3 POUT2 POUT1 POUT0 $
      PIN11 PIN10 PIN9 PIN8 PIN7 PIN6 PIN5 PIN4 PIN3 PIN2 PIN1 PIN0 $
      LOAD SCLK1 SCLK2)

;-- one cell of the input register

.model p2scell subckt : (PO SO PI SI LD PH1 PH2)
ltch0 PO,PI,LD latch
ltch1 IX,SI,PH1 latch
mux0 YY,PI,IX,LD mux
ltch2 SO,YY,PH2 latch
.ends

```

```

p2sc00 IX00,S000,PIN11,GND,LOAD,SCLK1,SCLK2 p2scell
p2sc01 POUT11,S001,PIN11,S000,LOAD,SCLK1,SCLK2 p2scell
p2sc02 POUT10,S002,PIN10,S001,LOAD,SCLK1,SCLK2 p2scell
p2sc03 POUT9,S003,PIN9,S002,LOAD,SCLK1,SCLK2 p2scell
p2sc04 POUT8,S004,PIN8,S003,LOAD,SCLK1,SCLK2 p2scell
p2sc05 POUT7,S005,PIN7,S004,LOAD,SCLK1,SCLK2 p2scell
p2sc06 POUT6,S006,PIN6,S005,LOAD,SCLK1,SCLK2 p2scell
p2sc07 POUT5,S007,PIN5,S006,LOAD,SCLK1,SCLK2 p2scell
p2sc08 POUT4,S008,PIN4,S007,LOAD,SCLK1,SCLK2 p2scell
p2sc09 POUT3,S009,PIN3,S008,LOAD,SCLK1,SCLK2 p2scell
p2sc10 POUT2,S010,PIN2,S009,LOAD,SCLK1,SCLK2 p2scell
p2sc11 POUT1,S011,PIN1,S010,LOAD,SCLK1,SCLK2 p2scell
p2sc12 POUT0,S012,PIN0,S011,LOAD,SCLK1,SCLK2 p2scell
p2sc13 IX13,S013,GND,S012,LOAD,SCLK1,SCLK2 p2scell
p2sc14 IX14,S014,GND,S013,LOAD,SCLK1,SCLK2 p2scell
p2sc15 IX15,S015,GND,S014,LOAD,SCLK1,SCLK2 p2scell
p2sc16 IX16,S016,GND,S015,LOAD,SCLK1,SCLK2 p2scell
p2sc17 IX17,S017,GND,S016,LOAD,SCLK1,SCLK2 p2scell
p2sc18 IX18,S018,GND,S017,LOAD,SCLK1,SCLK2 p2scell
p2sc19 IX19,S019,GND,S018,LOAD,SCLK1,SCLK2 p2scell
p2sc20 IX20,S020,GND,S019,LOAD,SCLK1,SCLK2 p2scell
p2sc21 IX21,S021,GND,S020,LOAD,SCLK1,SCLK2 p2scell
p2sc22 IX22,S022,GND,S021,LOAD,SCLK1,SCLK2 p2scell
p2sc23 IX23,S023,GND,S022,LOAD,SCLK1,SCLK2 p2scell
p2sc24 IX24,S024,GND,S023,LOAD,SCLK1,SCLK2 p2scell
p2sc25 IX25,S025,GND,S024,LOAD,SCLK1,SCLK2 p2scell
p2sc26 IX26,S026,GND,S025,LOAD,SCLK1,SCLK2 p2scell
p2sc27 IX27,S027,GND,S026,LOAD,SCLK1,SCLK2 p2scell
p2sc28 IX28,S028,GND,S027,LOAD,SCLK1,SCLK2 p2scell
p2sc29 IX29,S029,GND,S028,LOAD,SCLK1,SCLK2 p2scell
p2sc30 IX30,S030,GND,S029,LOAD,SCLK1,SCLK2 p2scell
p2sc31 IX31,S031,GND,S030,LOAD,SCLK1,SCLK2 p2scell

```

```

.ends

```

```

;-----

```

```

;-- Input Correlator

```

```

.model corr subckt : (CORRO WORDO EXECO $
DATAI WORDI EXECI CDINO CDIN1 CDIN2 CDIN3 CDIN4 CDIN5 $
CDIN6 CDIN7 CDIN8 CDIN9 CDIN10 CDIN11 PHI1 PHI2)

```

```

del1 DELO1,DATAO,SCLK1,SCLK2 wddelay
del2 DELO2,DELO1,SCLK1,SCLK2 wddelay
del3 DELO3,DELO2,SCLK1,SCLK2 wddelay
del4 DELO4,DELO3,SCLK1,SCLK2 wddelay
del5 DELO5,DELO4,SCLK1,SCLK2 wddelay
del6 DELO6,DELO5,SCLK1,SCLK2 wddelay
del7 DELO7,DELO6,SCLK1,SCLK2 wddelay
del8 DELO8,DELO7,SCLK1,SCLK2 wddelay
del9 DELO9,DELO8,SCLK1,SCLK2 wddelay

```

```

mult      CCR0,DATA0,WORD0,MDATAI,WORDI,$
          CDIN11,CDIN10,CDIN9,CDIN8,CDIN7,CDIN6,CDIN5,CDIN4,CDIN3,$
          CDIN2,CDIN1,CDIN0,MCLK1,MCLK2 spmult

mux0      MDATAI,DATAI,DELO9,SELECT mux
and0      SELECT,EXECI,EXEC0B and
and1      SHFTEN,EXEC0,EXECI and
or0       MULTEN,EXEC0,EXECI or
inv0      EXEC0B,EXEC0 inverter
ctrldel   EXEC0,EXECI,PHI1,PHI2 wddelay
mclkb     MCLK2,MCLK1,PHI2,PHI1,MULTEN clkbuf
sclkb     SCLK2,SCLK1,PHI2,PHI1,SHFTEN clkbuf

.ends

;-----

;-- Scaling Block

.model scale subckt : (DATA0 WORD0 EXEC0 DATAI WORDI EXECI PHI1 PHI2)

latch DATA0 DDATAI SGNEXT latch
del DDATAI DATAI delay
or REGEIT WORDI DWI1 DWI2 DWI3 DWI4 or
nand ENDEIT EXECI DEI1 DEI2 DEI3 DEI4 nand
nor SGNEXT ENDEIT REGEIT nor
del0 DWI1 WORDI PHI1 PHI2 bitdelay
del1 DWI2 DWI1 PHI1 PHI2 bitdelay
del2 DWI3 DWI2 PHI1 PHI2 bitdelay
del3 DWI4 DWI3 PHI1 PHI2 bitdelay
del4 WORD0 DWI4 PHI1 PHI2 bitdelay
cdel0 DEI1 EXECI PHI1 PHI2 bitdelay
cdel1 DEI2 DEI1 PHI1 PHI2 bitdelay
cdel2 DEI3 DEI2 PHI1 PHI2 bitdelay
cdel3 DEI4 DEI3 PHI1 PHI2 bitdelay
cdel4 EXEC0 DEI4 PHI1 PHI2 bitdelay

.ends

;-----

;-- Recursive Filtering Block

.model recur subckt : (DATA0 DATAI EXECI WORDI PHI1 PHI2)

adder DATA0,DUMMY3,DATAI,DELOUT,WORDI,GND,LCLK1,LCLK2 cseradd
mult  DELOUT,DUMMY1,DUMMY2,DELO9,WORDI,LCLK1,LCLK2 fcmult

del1  DELO1,DATA0,LCLK1,LCLK2 wddelay
del2  DELO2,DELO1,LCLK1,LCLK2 wddelay
del3  DELO3,DELO2,LCLK1,LCLK2 wddelay

```

```

del4 DELO4,DELO3,LCLK1,LCLK2 wddelay
del5 DELO5,DELO4,LCLK1,LCLK2 wddelay
del6 DELO6,DELO5,LCLK1,LCLK2 wddelay
del7 DELO7,DELO6,LCLK1,LCLK2 wddelay
del8 DELO8,DELO7,LCLK1,LCLK2 wddelay
del9 DELO9,DELO8,LCLK1,LCLK2 wddelay

cbuf LCLK1,LCLK2,PHI1,PHI2,EXECI clkbuf
.ends

;-----

;-- Nonrecursive Filtering Block

.model nonrecur subckt : (DATA0 EXEC0 WORD0 DATAI EXECI WORDI PHI1 PHI2)

bdel1 DWORDI,WORDI,PHI1,PHI2 bitdelay
bdel2 DEXECI,EXECI,PHI1,PHI2 bitdelay
bdel3 DDATAI,DATAI,PHI1,PHI2 bitdelay
not1 DEXBAR,DEXECI inverter
and1 ROMTRIG,WORDI,DEXBAR and
mult1 DPRODUCT,DUM1,DUM2,DUM3,DDATA9,COEFF1,DWORDI,PHI1,PHI2 smult
mult2 PRODUCT,DUM4,DUM5,WORD0,DDATAI,COEFF2,DWORDI,PHI1,PHI2 smult
rom1 COEFF1,ROMTRIG,PHI1,PHI2 rom30 ; read from FORTRAN unit 30
rom2 COEFF2,ROMTRIG,PHI1,PHI2 rom31 ; read from FORTRAN unit 31
clkbuf SCLK1,SCLK2,PHI1,PHI2,DEXECI clkbuf
cdelay EXEC0,DEXECI,PHI1,PHI2 wddelay
adder DATA0,DUM6,DPRODUCT,PRODUCT,WORD0,GND,PHI1,PHI2 cseradd
del0 DDATA0,DDATAI,SCLK1,SCLK2 wddelay
del1 DDATA1,DDATA0,SCLK1,SCLK2 wddelay
del2 DDATA2,DDATA1,SCLK1,SCLK2 wddelay
del3 DDATA3,DDATA2,SCLK1,SCLK2 wddelay
del4 DDATA4,DDATA3,SCLK1,SCLK2 wddelay
del5 DDATA5,DDATA4,SCLK1,SCLK2 wddelay
del6 DDATA6,DDATA5,SCLK1,SCLK2 wddelay
del7 DDATA7,DDATA6,SCLK1,SCLK2 wddelay
del8 DDATA8,DDATA7,SCLK1,SCLK2 wddelay
del9 DDATA9,DDATA8,SCLK1,SCLK2 wddelay

.ends

;-----
; Model statements
;-----

;-- Models from SERIAL-LIB.MOD

.model sread20 lgmod : model=1 par1=20
.model sread21 lgmod : model=2 par1=21
.model pread22 lgmod : model=5 par1=22
.model swrite23 lgmod : model=3 par1=23 par2=0 ;-- DEBUG OFF
.model swrite24 lgmod : model=4 par1=24 par2=0

```

```

.model pwrite25 lgmod : model=6 par1=25 par2=1 ;-- Negate the output
.model fcmult lgmod : model=11 par1=0.9604
.model spmult lgmod : model=12
.model wddelay lgmod : model=13
.model clkbuf lgmod : model=14
.model bitdelay lgmod : model=15
.model cseradd lgmod : model=16
.model latch lgmod : model=17
.model ssmult lgmod : model=19
.model divider lgmod : model=20
.model rom30 lgmod : model=21 par1=30
.model rom31 lgmod : model=22 par1=31
.model spmult16 lgmod : model=23
.model fsm1 lgmod : model=31
.model fsm2 lgmod : model=32

```

```

;-- Some other useful models

```

```

.model or or
.model delay delay : delay=1ns
.model and and
.model nor nor
.model nand nand
.model inverter inverter
.model buffer buffer : tr=1ns tf=1ns

```

```

.model mux subckt : (OUT IN1 IN0 SEL)
or0 OUT XX0 XX1 or
and0 XX0 IN0 SELB and
and1 XX1 IN1 SEL and
inv1 SELB SEL inverter
.ends

```

```

go
end

```

Appendix C

SPICE Netlist for Simulation of LeRoux-Gueguen Block

```

SPICE
LEROUX GUEGUEN SIMULATION

;-----
;
; System-level simulation of the Leroux-Gueguen block
;
; Copyright 1987, 1988, John W. O'Leary
;
; This file contains:
;   - circuit netlist of the LeRoux-Gueguen block
;   - simulation directives
;   - subcircuit definitions for coefficient calculation block
;     and lattice block
;   - model statements for the models defined in SERIAL-LIB.MOD
;-----

;-----
; Circuit netlist of LeRoux-Gueguen block
;-----

;-- Logic Sources

.MODEL VDD DC : 5
.MODEL WORD LSRC : 0 1 5NS 320NS 0NS 1NS 2NS 9NS 10NS 320NS
.MODEL EXEC LSRC : 0 1 5NS 32960NS 0NS 1NS 2NS 3199NS 3200NS 32960NS
.MODEL CLK1 LSRC : 0 1 0NS 10NS 0NS 2NS 3NS 4NS 5NS 10NS
.MODEL CLK2 LSRC : 0 1 0NS 10NS 0NS 6NS 7NS 8NS 9NS 10NS

WORD WORD WORD
EXEC EXEC EXEC ;-- The FRAME clock
CLK1 CLK1 CLK1
CLK2 CLK2 CLK2
VDD VDD VDD

;-- Netlist

;-- read from FORTRAN logical unit 20

and DOIT,EXEC,WORD and

```



```

reader DATAI,WORDI,DOIT,CLK1,CLK2 sread20

fsm DEXEC,DWORDI,EXEC,WORDI,CLK1,CLK2 fsm2

test INTD1,INTD2,INTWD,INTEX,RESULT,$
    Q15,Q14,Q13,Q12,Q11,Q10,Q9,Q8,Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0,$
    DATAI,FBD1,FBD2,DWORDI,FBWD,DEXEC,FBEX,CLK1,CLK2 lg1

test1 FBD1,FBD2,FBWD,FBEX,$
    Q15,Q14,Q13,Q12,Q11,Q10,Q9,Q8,Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0,$
    INTD1,INTD2,INTWD,INTEX,CLK1,CLK2 lg2

;-- write on FORTRAN logical unit 25

writer DUM1,Q15,Q14,Q13,Q12,Q11,Q10,Q9,Q8,Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0,$
    RESULT,CLK1,CLK2 pwrite25

;-----
; Simulation Directives
;-----

;-- 1ns in simulation = 10ns in real time

.time 1ns 1648000ns

;-----
; Speech analysis system subcircuits
;-----

;-- Coefficient calculation block

.model LG1 subckt : (DAT10 DAT20 WORD0 EXE0 RESULT $
    Q15 Q14 Q13 Q12 Q11 Q10 Q9 Q8 Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0 $
    DATAI DAT1I DAT2I WORDI WORD1I EXE1 EXEC1I PHI1 PHI2)

mux1 DDAT10,DATAI,DAT1I,EXE1 mux
mux2 DDAT20,DATAI,DAT2I,EXE1 mux
mux3 DWDO,WORDI,WORD1I,EXE1 mux

div Q15,Q14,Q13,Q12,Q11,Q10,Q9,Q8,Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0, $
    DDAT10,DDAT21,DIVTRIG,PHI1,PHI2 divider

d1del0 DDAT11,DDAT10,PHI1,PHI2 wddelay
d1del1 DDAT12,DDAT11,PHI1,PHI2 wddelay
d1del2 DDAT13,DDAT12,PHI1,PHI2 wddelay
d1del3 DDAT14,DDAT13,PHI1,PHI2 wddelay
d1del4 DDAT15,DDAT14,PHI1,PHI2 wddelay
d1del5 DDAT16,DDAT15,PHI1,PHI2 wddelay
d1del6 DDAT17,DDAT16,PHI1,PHI2 wddelay
d1del7 DDAT18,DDAT17,PHI1,PHI2 wddelay
d1del8 DDAT19,DDAT18,PHI1,PHI2 wddelay

```

```

d2del0 DDAT21,DDAT20,PHI1,PHI2 wddelay
d2del1 DDAT22,DDAT21,PHI1,PHI2 wddelay
d2del2 DDAT23,DDAT22,PHI1,PHI2 wddelay
d2del3 DDAT24,DDAT23,PHI1,PHI2 wddelay
d2del4 DDAT25,DDAT24,PHI1,PHI2 wddelay
d2del5 DDAT26,DDAT25,PHI1,PHI2 wddelay
d2del6 DDAT27,DDAT26,PHI1,PHI2 wddelay
d2del7 DDAT28,DDAT27,PHI1,PHI2 wddelay
d2del8 DDAT29,DDAT28,PHI1,PHI2 wddelay
d2del9 DDAT2A,DDAT29,PHI1,PHI2 wddelay

```

```

and1 NORMTRIG,DWD1,IDEI2 and
and2 RESULT,WORD0,DEX1,IDEI0 and
and3 WORD0,DWD10,IEIECI and
and4 EXECO,DEX10,IEIECI and
and5 DAT10,DDAT19,IEIECI and
and6 DAT20,DDAT2A,IEIECI and
inv1 IDEI2,DEX2 inverter
inv2 IDEI0,DEX0 inverter
inv3 IEIECI,EXECI inverter
or1 DEX0,EXECI,EXEC1I or
and7 INITTRIG,EXECI,DEIC1,IDEIC2,DWD1 and
idel1 DEIC1,EXECI,PHI1,PHI2 wddelay
idel2 DEIC2,DEIC1,PHI1,PHI2 wddelay
inv4 IDEIC2,DEIC2 inverter
or2 DIVTRIG,INITTRIG,NORMTRIG or

```

```

wdel0 DWD1,DWD0,PHI1,PHI2 wddelay
wdel1 DWD2,DWD1,PHI1,PHI2 wddelay
wdel2 DWD3,DWD2,PHI1,PHI2 wddelay
wdel3 DWD4,DWD3,PHI1,PHI2 wddelay
wdel4 DWD5,DWD4,PHI1,PHI2 wddelay
wdel5 DWD6,DWD5,PHI1,PHI2 wddelay
wdel6 DWD7,DWD6,PHI1,PHI2 wddelay
wdel7 DWD8,DWD7,PHI1,PHI2 wddelay
wdel8 DWD9,DWD8,PHI1,PHI2 wddelay
wdel9 DWD10,DWD9,PHI1,PHI2 wddelay

```

```

edel0 DEX1,DEX0,PHI1,PHI2 wddelay
edel1 DEX2,DEX1,PHI1,PHI2 wddelay
edel2 DEX3,DEX2,PHI1,PHI2 wddelay
edel3 DEX4,DEX3,PHI1,PHI2 wddelay
edel4 DEX5,DEX4,PHI1,PHI2 wddelay
edel5 DEX6,DEX5,PHI1,PHI2 wddelay
edel6 DEX7,DEX6,PHI1,PHI2 wddelay
edel7 DEX8,DEX7,PHI1,PHI2 wddelay
edel8 DEX9,DEX8,PHI1,PHI2 wddelay
edel9 DEX10,DEX9,PHI1,PHI2 wddelay

```

```

.ends

```

```

;-----

```

```
-- Lattice block
```

```
.model LG2 subckt : (DAT10 DAT20 WORD0 EXECO $
                    C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1 C0 $
                    DAT1I DAT2I WORDI EXECI PHI1 PHI2)
```

```
inv LATCH,EXECO inverter
del1 EXECO,EXECI,PHI1,PHI2 wddelay
ltch0 LC0,C0,LATCH latch
ltch1 LC1,C1,LATCH latch
ltch2 LC2,C2,LATCH latch
ltch3 LC3,C3,LATCH latch
ltch4 LC4,C4,LATCH latch
ltch5 LC5,C5,LATCH latch
ltch6 LC6,C6,LATCH latch
ltch7 LC7,C7,LATCH latch
ltch8 LC8,C8,LATCH latch
ltch9 LC9,C9,LATCH latch
ltch10 LC10,C10,LATCH latch
ltch11 LC11,C11,LATCH latch
ltch12 LC12,C12,LATCH latch
ltch13 LC13,C13,LATCH latch
ltch14 LC14,C14,LATCH latch
ltch15 LC15,C15,LATCH latch
mult1 DATA1M,DATA1,WORD1,DAT1I,WORDI,$
      LC15,LC14,LC13,LC12,LC11,LC10,LC9,LC8,LC7,LC6,LC5,LC4,LC3,LC2,LC1,LC0,$
      PHI1,PHI2 spmult16
mult2 DATA2M,DATA2,WORD0,DAT2I,WORDI,$
      LC15,LC14,LC13,LC12,LC11,LC10,LC9,LC8,LC7,LC6,LC5,LC4,LC3,LC2,LC1,LC0,$
      PHI1,PHI2 spmult16
add1 DAT10,DUMMY,DATA1,DATA2M,WORD1,VDD,PHI1,PHI2 cseradd
add2 DAT20,DUMMY1,DATA2,DATA1M,WORD0,VDD,PHI1,PHI2 cseradd

.ends
```

```
-- Model statements
```

```
-- Models from SERIAL-LIB.MOD
```

```
.model sread20 lgmod : model=1 par1=20
.model sread21 lgmod : model=2 par1=21
.model pread22 lgmod : model=5 par1=22
.model swrite23 lgmod : model=3 par1=23 par2=0 ;-- DEBUG OFF
.model swrite24 lgmod : model=4 par1=24 par2=0
.model pwrite25 lgmod : model=6 par1=25 par2=1 ;-- Negate the output
.model fcmult lgmod : model=11 par1=0.9604
.model spmult lgmod : model=12
.model wddelay lgmod : model=13
.model clkbuf lgmod : model=14
```

```
.model bitdelay lgmod : model=15
.model cseradd lgmod : model=16
.model latch lgmod : model=17
.model ssmult lgmod : model=19
.model divider lgmod : model=20
.model rom30 lgmod : model=21 par1=30
.model rom31 lgmod : model=22 par1=31
.model spmult16 lgmod : model=23
.model fsm1 lgmod : model=31
.model fsm2 lgmod : model=32
```

```
;-- Some other useful models
```

```
.model or or
.model delay delay : delay=1ns
.model and and
.model nor nor
.model nand nand
.model inverter inverter
.model buffer buffer : tr=1ns tf=1ns

.model mux subckt : (OUT IN1 IN0 SEL)
or0 OUT XX0 XX1 or
and0 XX0 IN0 SELB and
and1 XX1 IN1 SEL and
inv1 SELB SEL inverter
.ends
```

```
go
end
```

References

- [1] Andreas Antoniou, *Digital Filters: Analysis and Design*, McGraw-Hill Book Company, New York, 1979.
- [2] Thomas P. Barnwell, III, "Recursive windowing for generating autocorrelation coefficients for LPC analysis," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-29, pp. 1062-1066, October 1981.
- [3] Peter Denyer and David Renshaw, *VLSI Signal Processing: A Bit-Serial Approach*, Addison-Wesley Publishing Company, Reading, Mass., 1985.
- [4] Donald L. Dietmayer, *Logic Design of Digital Systems, Second Edition*, Allyn and Bacon, Inc., Toronto, 1978.
- [5] Benjamin Friedlander, "Lattice filters for adaptive processing," *Proceedings of the IEEE*, vol. 70, pp. 829-867, August 1982.
- [6] Robert A. Frohwerk, "Signature analysis : A new digital field service method," *Hewlett-Packard Journal*, pp. 2-8, May 1977.
- [7] A. H. Gray, Jr. and J. D. Markel, "Linear Prediction Analysis Programs (AUTO-COVAR)," *Programs for Digital Signal Processing*, IEEE Press, John Wiley & Sons, Inc., New York, 1979.
- [8] J. R. Griffith, G. P. Bell, T. A. Kwasniewski and C. H. Chan, "User defined circuit elements for VLSI simulation," *Canadian Conference on Very Large Scale Integration Technical Digest*, Winnipeg, 1987.
- [9] Robert Grossman, "System design and verification for digital signal processing", *VLSI Systems Design*, pp. 15-21, March 1988.
- [10] *Guide to the Integrated Circuit Implementation Services of the Canadian Microelectronics Corporation*, Version 3:0, January 1987.
- [11] Robert D. Hoyle, *Digital Speech Coding for Land Mobile Radio*, M.Eng Thesis, Department of Systems and Computer Engineering, Carleton University, April 1986.
- [12] Paul J. Hurst and Robert W. Brodersen, "A MOS-LSI autocorrelator for linear pre-

- diction of speech," *IEEE Journal of Solid-State Circuits*, vol. SC-19, pp. 1022-1029, December 1984.
- [13] Kai Hwang, *Computer Arithmetic : Principles, Architecture and Design*, John Wiley & Sons, Toronto, 1979.
 - [14] L. B. Jackson, "On the interaction of roundoff noise and dynamic range in digital filters," *Bell System Technical Journal*, vol. 49, pp. 159-184, February 1970.
 - [15] Brian Kernighan, "RATFOR, a preprocessor for a Rational FORTRAN," *Unix Programmers' Manual*, 7th Edition, Bell Laboratories.
 - [16] P. Kroon, E. F. Deprettere and R. J. Sluyter, "Regular-pulse excitation — A novel approach to effective and efficient multipulse coding of speech," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol ASSP-34, pp. 1054-1063, October 1986.
 - [17] J. LeRoux and C. Gueguen, "A fixed point computation of partial correlation coefficients," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol ASSP-25, pp. 257-259, June 1977.
 - [18] R. F. Lyon, "Two's complement pipeline multipliers," *IEEE Transactions on Communications*, pp. 418-425, April 1976.
 - [19] John Makhoul, "Linear prediction: A tutorial review," *Proceedings of the IEEE*, vol. 63, pp. 561-580, April 1975.
 - [20] John Makhoul and Lynn Cosell, "Adaptive lattice analysis of speech," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-29, pp. 654-658, June 1981.
 - [21] J. D. Markel and A. H. Gray, jr., *Linear Prediction of Speech*, Springer-Verlag, New York, 1976.
 - [22] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, Reading, Mass., 1980.
 - [23] J. E. Narvig, "Evaluation of six medium bit-rate coders for the pan-European digital mobile radio system," *IEEE Journal on Selected Areas in Communications*, vol. 6,

pp. 324–331, February 1988.

- [24] P. K. Rajasekaran and J. C. Hansen, "Finite word length effects of the LeRoux–Gueguen algorithm in computing reflection coefficients," *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 1286–1290, 1982.
- [25] Shuzo Saito and Kazuo Nakata, *Fundamentals of Speech Signal Processing*, Academic Press, New York, 1985.
- [26] R. A. Saleh, "Iterated Timing Analysis and SPLICE1," *International Conference on Computer Aided Design Digest*, Santa Clara, 1983.
- [27] J. Savir and P. H. Bardell, "On random pattern test length," *Proceedings of the International Test Conference*, pp. 95–106, 1983.
- [28] J. Savir, G. Ditlow and P. Bardell, "Random pattern testability," *Proceedings of the 13th International Symposium on Fault-Tolerant Computing*, pp. 80–89, 1983.
- [29] R. W. Schafer and L. R. Rabiner, "Digital representations of speech signals," *Proceedings of the IEEE*, vol. 63, pp. 662–677, April 1975.
- [30] R. J. Sluyter, G. J. Bosscha and H. M. P. T. Schmitz, "A 9.6 Kbit/s speech coder for mobile radio applications," *Proceedings of IEEE International Conference on Communications*, May 1984, pp. 1159–1162.
- [31] Joseph Tierney, "A study of LPC analysis of speech in additive noise," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-28, pp. 389–397, August 1980.
- [32] C. K. Un and D. T. Magill, "The residually-excited linear prediction vocoder with transmission below 9.6 kbps," *IEEE Transactions on Communications*, vol. COM-23, pp. 1466–1473, 1975.
- [33] V. R. Viswanathan, A. L. Higgins and W. H. Russell, "Design of a robust baseband LPC coder for speech transmission over 9.6 Kbit/s noisy channels," *IEEE Transactions on Communications*, vol. COM-30, pp. 663–673, April 1982.
- [34] *VIVID System CMOS Design Style Reference*, Version 1.0, The Microelectronics Center of North Carolina, 1984.

- [35] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design : A Systems Perspective*
Addison-Wesley Publishing Company, Reading, Mass., 1985.

END

18-02-90

FIN