A CUSTOM VLSI ARCHITECTURE FOR IMPLEMENTING LOW-DELAY ANALYSIS-BY-SYNTHESIS SPEECH CODING ALGORITHMS

by

Peter Dean Schuler
B.A.Sc., Simon Fraser University, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE
in the School
of
Engineering Science

© Peter Dean Schuler 1991
SIMON FRASER UNIVERSITY
June 1991

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

Approval

Name:	Peter Dean Schuler
Degree:	Master of Applied Science
Title of Thesis:	A Custom VLSI Architecture for Implementing Low-Delay Analysis-by-Synthesis Speech Coding Algorithms
Examining Committee:	Dr. John D. Jones Associate Professor School of Engineering Science Chair
	Dr. Vladimir Cuperman Professor School of Engineering Science
	Dr. R. H. Stephen Hardy Professor School of Engineering Science
	Dr. Paul K. M. Ho Assistant Professor School of Engineering Science
	Dr. James K. Cavers Professor School of Engineering Science
Date Approved:	June 21, 1991

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

" A Custom VLSI Architecture for Implementing Low-Delay Analysis-by-Synthesis Speech Coding Algorithms"

Author:

(signature)

Peter D. Schuler

(name)

June 25, 1991

(date)

Abstract

In the past, digital signal processing (DSP) algorithms have typically been implemented on general-purpose DSP chips. However, the high complexities of modern algorithms are pushing the capabilities of these chips to their limits. An alternative solution is to design a custom VLSI architecture which exploits the structure of a specific algorithm.

This thesis presents a custom VLSI architecture for implementing low-delay analysis-by-synthesis speech coding algorithms more efficiently than general-purpose DSP chips. The criteria used to compare the efficiency of different architectures are the performance characteristics of execution speed and memory requirements, and the physical properties of die size and power consumption. The architecture is based on a detailed analysis of two speech coding algorithms: Low-Delay Code-Excited Linear Prediction, which will be the new CCITT standard for 16 kbit/s speech coding, and Lattice Low-Delay Vector Excitation Coding. The required operations for these algorithms were determined and the architecture was designed to implement these operations efficiently.

The custom VLSI architecture consists of two types of arithmetic units: a variable number of Adaptive Arithmetic Units (AAUs) connected in parallel, and one Distortion Arithmetic Unit (DAU). Each AAU contains an adaptive datapath to perform various operations, such as filtering and inner product calculations, on one element (sample) of data. The DAU computes the distortion measure required for a vector quantization codebook search and determines the minimum distortion value. The number of AAUs is optimized for power consumption and chip area. My research shows that an efficient configuration for implementing the Lattice Low-Delay Vector Excitation Coding algorithm consists of 4 AAUs. This configuration implements the algorithm with an estimated power consumption of less than 300 mW at a clock rate of 2 MHz, and an area of 90 mm^2; a corresponding implementation on a general-purpose DSP chip such as the DSP32C would require approximately 1 W of power while operating at a clock rate of 50 MHz.

Acknowledgements

I wish to thank Dr. Cuperman and Dr. Hardy for allowing me the opportunity to work on this project, and for their guidance throughout my work on the thesis.

Many thanks also to MPR Teltech for the use of their Computer-Aided Design tools, and specifically to Mr. Tino Varelas, Mr. Graham Smith, and Mr. Greg Aasen for their assistance.

This work was supported by the Science Council of British Columbia and by the Natural Sciences and Engineering Research Council of Canada.

Table of Contents

List of Tables

List of Figures

1. An Introduction to VLSI Signal Processing

Digital signal processing (DSP) algorithms are typically implemented on general-purpose DSP chips. Custom architectures and application-specific integrated circuits (ASICs) may provide a more efficient implementation of specific algorithms. This thesis presents a new custom VLSI architecture for implementing low-delay analysis-by-synthesis speech coding algorithms more efficiently than general-purpose DSP chips. This chapter provides an introduction to past and present implementation solutions.

1.1 The Need for Custom VLSI Architectures

In the past, most digital signal processing (DSP) algorithms have been implemented on general-purpose DSP chips, such as the TMS320C25 [1]. This chip operates at a clock speed of 40 MHz. The instruction cycle time is 100 ns, allowing 10 million instructions to be carried out each second. One floating point operation (flop), such as a multiply-accumulate instruction, may be performed in one cycle if these instructions are repeated; therefore, the maximum throughput of this chip is 10 Mflops (million flops). Other chips have similar specifications. However, the complexities of modern algorithms often exceed 10 Mflops, and are pushing the capabilities of these chips to their limits. In particular, one of the speech coding algorithms analyzed in this thesis (LD-CELP) has an encoder complexity of 9 Mflops and a decoder complexity of 3.4 Mflops; therefore, it would not be possible to implement both an encoder and a decoder on a single chip.

One method of increasing computational power is to connect several DSP chips in parallel. However, most chips were not designed specifically for such connections, and interconnection problems arise. The overhead associated with sending data off-chip often makes such solutions unfeasible. This situation is beginning to change, though. Certain general-purpose chips, such as the new TMS320C40, are being designed especially for parallel processing applications.

An alternative solution is to design a custom VLSI architecture for a specific algorithm, such as speech coding. This architecture may use parallelism and pipelining to increase throughput by exploiting the structure of the algorithm. By using processing elements in parallel on a single chip rather than distributing the elements over several chips, inter-element communication is greatly simplified. Application-specific architectures have already been designed for applications such as wideband audio coding [2].

The design of application-specific VLSI architectures for digital signal processing, and the use of computer-aided design (CAD) tools for VLSI design were discussed in my Bachelor's thesis [3]. The results of that thesis showed that a custom architecture is required to implement the low-delay speech coding algorithms efficiently.

1.2 A Proposed Custom Architecture

This thesis proposes a custom VLSI architecture for implementing speech coding algorithms more efficiently than general-purpose DSP chips. To compare the efficiency of various architectures, certain criteria are needed to evaluate the performance. These criteria include performance characteristics such as speed of execution and program and data memory requirements, and physical properties such as die size and power consumption. The architecture discussed here implements the LLD-VXC algorithm with a slower clock rate and lower power consumption than general-purpose DSP chips, and requires a comparable amount of memory and die area.

For my Bachelor's thesis, I compared the implementation of a small section of the LLD-VXC speech coding algorithm on various architectures, including general-purpose DSP chips and custom architectures. Both scalar and vector processors were considered. The results show that a custom vector architecture provides an implementation which requires the fewest cycles and the least number of instructions. This thesis follows up on those results.

First, I broadened the scope of my analysis to include the entire LLD-VXC algorithm; as well, I studied other speech coding algorithms, including LD-CELP and VSELP. I determined what hardware structures were necessary to implement these algorithms efficiently. Next, I reviewed the previously designed architectures and considered the implementation of the other algorithms on these architectures. The designs were modified as required and a final architecture was developed.

Estimates of the power consumption and chip area were obtained by using the VLSI System tools available at MPR Teltech. The various blocks of the architecture were compiled to a layout. The estimates were analyzed to see if the design was feasible and the limiting factors for a practical design were determined.

1.3 A Guide to the Thesis

The remainder of the thesis is organized as follows: Chapter 2 describes the speech coding algorithms which were analyzed to develop the new architecture, including a summary of their complexities; the mapping of these algorithms onto hardware structures is discussed in Chapter 3. The complete custom architecture is presented in Chapter 4, and its control system is described in Chapter 5. Chapter 6 discusses the implementation of the LLD-VXC algorithm on the architecture, and Chapter 7 discusses the efficiency of this implementation. The suitability of the architecture for other algorithms, including LD-CELP, VSELP, and the Fast Fourier Transform, is considered in Chapter 8. Finally, Chapter 9 presents the conclusions of the thesis and Chapter 10 suggests some areas of future research.

2. The Speech Coding Algorithms

Speech coding consists of digitizing the speech signal and eliminating the redundancies from the signal so that a lower bandwidth is required to transmit the digital voice data. There are currently CCITT standards for transmission rates of 64 kbit/s (PCM) and 32 kbit/s (ADPCM); a 16 kbit/s standard is under consideration. This chapter briefly discusses several common techniques of speech coding, including analyses of the speech coding algorithms which were studied to develop the custom architecture.

2.1 Techniques of Speech Coding

The most straight-forward speech coding technique is pulse-code modulation (PCM). The voice data is sampled at 8 kHz, and each sample is quantized to 8 bits; therefore, a transmission rate of 64 kbit/s is required. A logarithmic quantizer is used instead of a linear quantizer, allowing finer quantization of low-amplitude signals. This approach results in toll-quality speech (acceptable for commercial telephony); it is desirable to maintain this quality while reducing the transmission rate.

When a predictor is added to the coder, the resulting system is known as Differential PCM (DPCM). The current sample of the speech signal is predicted, based on previous input samples, and the prediction subtracted from the actual input. The difference signal is then transmitted. Because the difference should be smaller than the input, only 4 bits are used to digitize the signal; the resulting transmission rate is 32 kbit/s. Further modifications are to adapt the coefficients of the predictor to maintain a near-optimal predictor at all times, and to adapt the quantizer; the system is then called Adaptive DPCM (ADPCM) [4]. The quality of current ADPCM systems is nearly as high as that of PCM, with half the transmission rate.

Vector quantization [5,6] may be used to further reduce the transmission rate to 16 kbit/s and below. The samples of the input signal are grouped into vectors; each vector is compared to all codevectors stored in a codebook, and the optimal codevector, based on a distortion measure such as least-square distance, is transmitted. Because there are far fewer codevectors than actual input vectors, fewer bits are required to transmit the data. Again, when the difference (residual) between the input and a predicted value is vector quantized rather than the input itself, fewer codevectors are required for accurate representation and lower bit rates are achieved.

An important class of speech coding algorithms is based on linear prediction in an Analysis-by-Synthesis configuration, which is sometimes called Code Excited Linear Prediction (CELP). The basic structure of an Analysis-by-Synthesis speech coder includes an excitation codebook and a synthesis filter, as shown in Figure 1. Input speech samples are grouped into vectors and codebook samples are grouped into codevectors. Each codevector is passed through the synthesis filter and the filtered codevector is compared to the input vector using a distortion measure such as weighted least-squared distance. The index of the codevector resulting in the smallest distortion is then transmitted.

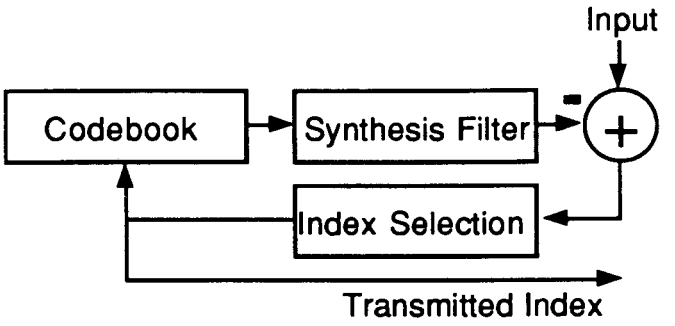

Figure 1—Basic Configuration of a Typical CELP Speech Coder

Other filters, such as a pitch predictor and a weighting filter, may also be included in the coder. A pitch predictor exploits the periodicity of voiced speech by estimating the pitch period of the input and using samples from the previous period to predict the current value; this filter is connected in series with the synthesis filter. A weighting filter reduces the amount of perceived noise by exploiting the acoustic masking properties of the human ear; using this filter in the index selection results in the weighted least-square distortion measure.

The two most common implementation structures for the filters are the direct form and the lattice structure [4]. The direct form has the advantage that the filter coefficients are linearly related to the transfer function. However, it also has some important disadvantages: adding one stage to a direct form implementation requires recalculation of all filter coefficients, the coefficients are particularly sensitive to quantization, and it is difficult to check for filter stability.

The lattice filter overcomes these disadvantages. The reflection coefficients which specify a lattice structure are less sensitive to quantization effects. When a stage is added to the filter, the coefficients of the previous stages do not change. Also, the check for stability is much simpler—the coefficients must all be less than 1. In fact, direct form coefficients

are often converted to reflection coefficients to check for stability. For these reasons, filters are often implemented using a lattice structure. On the other hand, a lattice structure is more complex than a direct structure and each stage of a lattice filter requires two multiplications rather than one.

The adaptation routines fall into one of two categories: forward adaptive or backward adaptive [4]. Forward adaptive routines adapt the predictors based on the actual input signal. Because this signal is not available at the receiver, the predictor parameters must be coded and transmitted to the receiver. Typically, block adaptation is used; that is, a block of the input signal is analyzed to determine the optimal parameters, resulting in a significant algorithmic delay.

On the other hand, backward adaptive routines adapt the predictors based on the reconstructed signal, which is available at both the transmitter and the receiver. Therefore, the entire data rate is available for transmission of the coded error signal. Typically, recursive adaptation is used; that is, the adaptation proceeds on a sample-by-sample basis, using a gradient algorithm to correct the predictor parameters after each sample. As a result, the algorithmic delay is negligible and backward adaptive routines may be used in low-delay algorithms. The higher available data rate compensates for the fact that quality is degraded by adapting on a noisy signal: For a fixed overall transmission rate, forward and backward adaptation result in nearly equivalent speech quality.

Filter coefficients are not necessarily updated every input vector, even if they are adapted that frequently. The coefficients often do not change significantly with each vector. Therefore, a significant complexity reduction is achieved by computing new estimates of the coefficients on a vector-by-vector basis but replacing the old coefficients only every few vectors.

An important point to note is that the encoder for a backward-adaptive system must include a simulated decoder because the reconstructed signal used to adapt the predictors at the decoder must also be used at the encoder. Therefore, the decoder is just a subset of the encoder; it need not be specified separately, and it has a much lower complexity than the encoder. The decoder, and hence the simulated decoder, includes the excitation codebook and the synthesis filter and its adapter.

Three algorithms were studied in detail to develop the custom architecture: Lattice Low-Delay Vector Excitation Coding (LLD-VXC), Low-delay Code-Excited Linear Prediction (LD-CELP), and Vector Sum Excited Linear Prediction (VSELP). These algorithms are now discussed.

2.2 The 16 kbit/s LLD-VXC Algorithm

The 16 kbit/s Lattice Low-Delay Vector Excitation Coding (LLD-VXC) algorithm [7,8] has the CELP structure shown in Figure 1. It also contains a pitch predictor and a perceptual weighting filter. A more detailed block diagram of the encoder is shown in Figure 2. This section discusses primarily the encoder; the decoder consists of a subset of the operations required for the encoder.

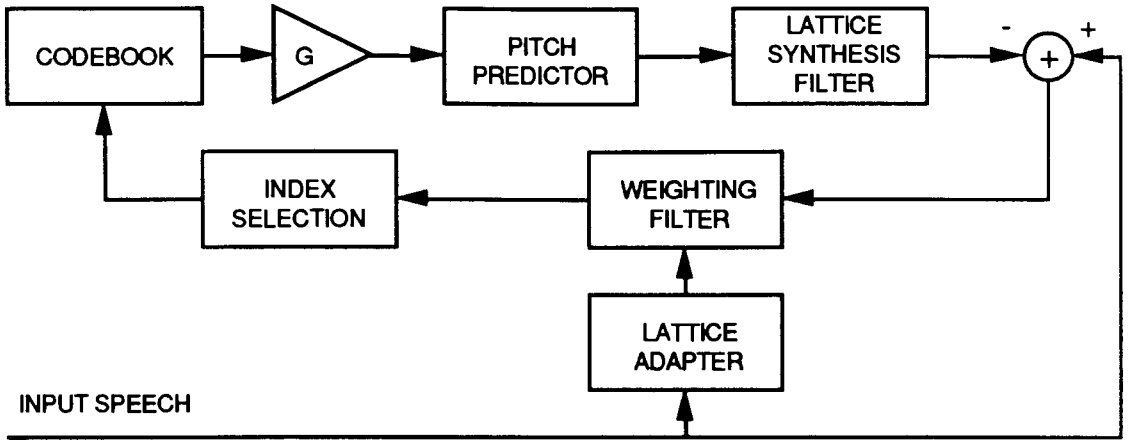

Figure 2—Block Diagram of LLD-VXC

The LLD-VXC algorithm operates as follows: For each input vector the codebook is searched; that is, each excitation codevector in turn is scaled by a gain and filtered by a long-term filter (pitch predictor) and a short-term filter. Each filtered codevector is then compared to the input vector. The distortion (difference) vector is weighted to account for perceptual properties, and the index of the codevector which results in the minimum distortion is transmitted by the coder.

The vector dimension for LLD-VXC is 4. At a sampling rate of 8 kHz and a transmission rate of 16 kbit/s, each sample must be encoded using 2 bits, or equivalently, each vector using 8 bits; therefore, the codebook contains $2^8 = 256$ codevectors. The least-squared distortion measure is used to compare filtered codevectors to input vectors:

$$d = \|\mathbf{x}-\mathbf{y}\|^2 = \sum_{i=1}^{V}(x_i-y_i)^2 \qquad (1)$$

where V is the vector dimension, $\mathbf{x}$ is the weighted input vector, and $\mathbf{y}$ is the current filtered codevector.

The synthesis filter is an adaptive 20th-order lattice filter. It is adapted using a backward recursive gradient algorithm. Because LLD-VXC is a backward-adaptive system, it can meet the CCITT low-delay criterion for 16 kbit/s speech coding: the processing delay is less than 5 ms. The pitch predictor has three taps:

$$x_{PRED}(n) = \beta_{-1}x(n-L+1) + \beta_0 x(n-L) + \beta_{+1}x(n-L-1) \quad (2)$$

where β_{-1}, β_0, and β_{+1} are the predictor coefficients and L is the pitch period. The perceptual weighting filter is a 10th-order pole-zero direct-form filter; however, it is adapted based on the input signal with a lattice adapter. Filter coefficients are updated every 12 vectors. Also, a 10-pole non-adaptive gain predictor is present. The operation of the filters is discussed in more detail in the next chapter.

An important feature of this algorithm is that lattice filters are used. As a result, the stability check has a very low complexity. As well, there are important implications for the design of a hardware structure because the architecture must be able to implement both lattice and direct-form filtering operations.

The algorithm provides several complexity reductions over the CELP structure. The most notable is that the response to the synthesis filter is separated into the zero-state response (ZSR) and the zero-input response (ZIR):

$$r(n) = r_{ZSR}(n) + r_{ZIR}(n) \quad (3)$$

The ZSR is due only to the input signal to the synthesis filter, which is the excitation codevectors. Because the codevectors never change and the synthesis filter coefficients are updated only every 12 vectors, the ZSR need be computed only when the filter coefficients change. On the other hand, the ZIR is based on the previous filter state, which is independent of the input codevector. As a result, the ZIR need be computed only once per vector rather than once per codevector.

A second complexity reduction is that the weighting filter is moved from the output of the adder to the input branches of the adder. On the path of the input speech, the input vector is weighted before being compared to the filtered codevector. On the codevector path, the weighting filter is combined with the synthesis filter, resulting in a weighted, filtered codevector.

A detailed analysis of the computational complexity of the Low-Delay VXC algorithm with a 2-pole 6-zero synthesis filter may be found in [9]. The total complexity of the algorithm is shown to be 4.6 Mflops. By changing the synthesis filter to a 20th-order lattice filter, the complexity of the algorithm is increased, as demonstrated by my analysis

as shown in Table 1. Here, a floating-point operation (flop) is defined as a multiplication and an accumulate.

Table 1—Complexity Analysis of LLD-VXC

Operation	fp ops	freq (kHz)	Comp (Mflops)
Perceptual weighting filter			
Input weighting	20	8	0.160
Adapt weighting filter coeffs	70	8	0.560
Stability check, convert to LPC coeffs	220	1/6	0.037
Gain predictor			
Predict gain	10	2	0.020
Adapt gain predictor coeffs	—	—	0
Pitch predictor			
Pitch prediction	3	8	0.024
Pitch tracking	20	8	0.160
Adapt pitch predictor coeffs	18	8	0.144
Update pitch period	19500	1/32	0.610
Computation of ZIR vector			
Synthesis and weighting filter	80	8	0.640
Adapt filter coeffs	100	8	0.800
Stability check	340	1/6	0.057
Codebook search module			
Impulse response vector computation	48	1/6	0.008
Filter codevectors	2048	1/6	0.341
Calculate energy	1024	1/6	0.171
Error calc & best index selection	1280	2	2.560
Simulated decoder			
Filter memory update	8	2	0.016
TOTAL			6.3

The total complexity of the LLD-VXC algorithm is 6.4 Mflops. The codebook search requires the most computational power (2.6 Mflops), which indicates that this section must be implemented very efficiently for an efficient implementation of the entire algorithm. Other high-complexity sections are the computation of the pitch period and the adaptation of the filter coefficients.

2.3 The 16 kbit/s LD-CELP Algorithm

The 16 kbit/s Low-Delay Code Excited Linear Prediction (LD-CELP) algorithm [10] is similar in structure to LLD-VXC and results in comparable voice quality. A block diagram of the encoder is shown in Figure 3. It is being considered for standardization at 16 kbit/s by the CCITT.

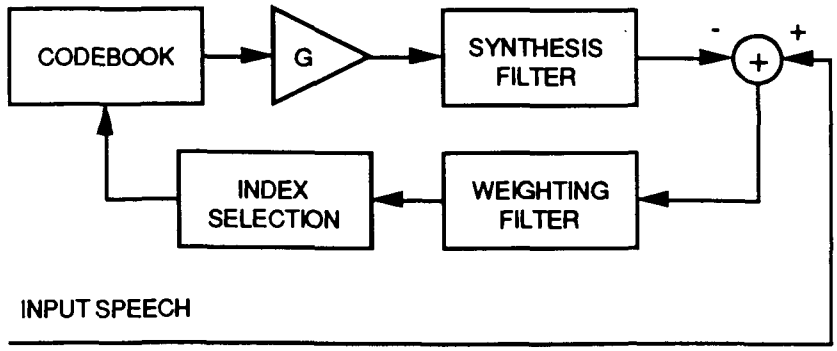

Figure 3—Block Diagram of LD-CELP

The vector dimension for LD-CELP is 5, allowing 10 bits for coding each vector. To reduce the codebook search complexity, the codebook is divided into a shape codebook containing 128 codevectors, requiring 7 bits to encode, and a gain codebook containing 8 gain values, requiring 3 bits to encode. To produce the final codevector, the chosen shape codevector is multiplied by the chosen gain value. The least-squared distortion measure is again used to compare the input to the filtered codevectors. The codebook search is the same in principle as that in LLD-VXC, with the exception of the separate gain codebook:

$$d = \|\mathbf{x}-G\mathbf{y}\|^2 = \sum_{i=1}^{V}(x_i-Gy_i)^2 \qquad (4)$$

where V is the vector dimension, $\mathbf{x}$ is the weighted input vector, $\mathbf{y}$ is the current filtered codevector, and G is the current gain value.

The synthesis filter is a 50-pole direct-form filter. There is no pitch predictor in this system; therefore, the synthesis filter must be significantly longer than when a pitch predictor is present, in order to take the periodicity of the input speech into account. As in LLD-VXC, a ZIR-ZSR decomposition provides a significant complexity reduction. The perceptual weighting filter is a 10th-order pole-zero filter. All filter coefficients are updated every 4 vectors. Backward adaptation again results in a processing delay of less than 5 ms. A 10-pole gain predictor is present; its coefficients are updated every vector.

The LD-CELP algorithm uses Levinson-Durbin recursion to adapt the coefficients of all filters. The input signal is windowed and used to estimate the autocorrelation function on a vector-by-vector basis. Then, at each coefficient update period, the optimal filter coefficients are computed recursively, based on the autocorrelation function. Because the coefficients are calculated explicitly rather than being estimated, and because of the long synthesis filter, the complexity of the adaptation routines is much higher than those of

LLD-VXC. Also, the recursive nature of the routine makes its implementation on a parallel and pipelined architecture inefficient. Problems with the implementation of LD-CELP are discussed further in Chapter 8.

I performed a detailed analysis of the computational complexity of the LD-CELP encoder. The results are shown in Tables 2 and 3, the first listing the total complexity of the major sections of the algorithm, the second giving a detailed breakdown of the analysis.

Table 2—Summary of Complexity of LD-CELP

Section of Algorithm	Complexity (Mflops)
Adapter for perceptual weighting filter	0.427
Adapter for synthesis filter	2.763
Adapter for vector gain	0.175
Perceptual weighting filter	0.160
Computation of ZIR vector	0.568
Codebook search module	4.813
Simulated decoder	0.098
TOTAL	9.00

The total complexity of the LD-CELP encoder is 9.0 Mflops, significantly higher than that of LLD-VXC. As mentioned in the previous chapter, the decoder consists of the excitation VQ codebook, the gain predictor and its adaptor, and the synthesis filter and its adapter. The complexity of the decoder is approximately 3.4 Mflops, much lower than that of the encoder.

Once again, the codebook search has the highest complexity (3.7 Mflops), and the need for an efficient codebook search architecture is seen. The codevector filtering operation also requires a large amount of computation. As well, the adapter for the synthesis filter also has a higher complexity than that of LLD-VXC. Because the Levinson-Durbin algorithm is used for the adaptation, this high complexity may present some difficulty in the overall implementation of this algorithm.

Except for the adaptation routines, implementation of LD-CELP is nearly identical to that of LLD-VXC. Therefore, most of the analysis in the following chapter applies to both algorithms.

Table 3—Detailed Complexity Analysis of LD-CELP

Operation	fp ops	freq (kHz)	Comp (Mflops)
Adapter for weighting filter			
Recursive windowing	226	1.6	0.361
Levinson-Durbin	124	0.4	0.050
Weighting filter coeff calc	40	0.4	0.016
Adapter for synthesis filter			
Recursive windowing	1046	1.6	1.673
Levinson-Durbin	2624	0.4	1.050
Bandwidth expansion	100	0.4	0.040
Adapter for vector gain			
RMS & log calc	8	1.6	0.013
Recursive windowing	50	1.6	0.080
Levinson-Durbin	124	0.4	0.050
Bandwidth expansion	20	0.4	0.008
Log-gain linear predictor	10	1.6	0.016
Miscellaneous	5	1.6	0.008
Perceptual weighting filter	100	1.6	0.160
Computation of ZIR vector			
Synthesis filter	250	1.6	0.400
Weighting filter	100	1.6	0.160
VQ target vector comp	5	1.6	0.008
Codebook search module			
Impulse response vector calc	34	0.4	0.014
Shape codevector conv & table	2688	0.4	1.075
VQ target vector norm	6	1.6	0.010
Time-reversed convolution	15	1.6	0.024
Error calc & best index selection	2306	1.6	3.690
Simulated decoder			
Excitation VQ codebook	6	1.6	0.010
Gain scaling unit	5	1.6	0.008
Filter memory update	50	1.6	0.080

2.4 The 8 kbit/s VSELP Algorithm

The 8 kbit/s Vector Sum Excited Linear Prediction (VSELP) algorithm [11] is significantly different from the other two algorithms. It was developed for mobile communications and is the TIA standard for digital cellular communication. The overall rate of transmission is 13 kbit/s, of which 8 kbit/s is used for speech coding and 5 kbit/s for error control; I analyzed only the speech coding section. This algorithm is also based on the CELP class of coders; however, it uses forward adaptation and a large vector dimension of 40, and therefore does not meet the low-delay criterion. Most importantly, it uses a structured codebook rather than a trained codebook to reduce the complexity of the vector quantization operation.

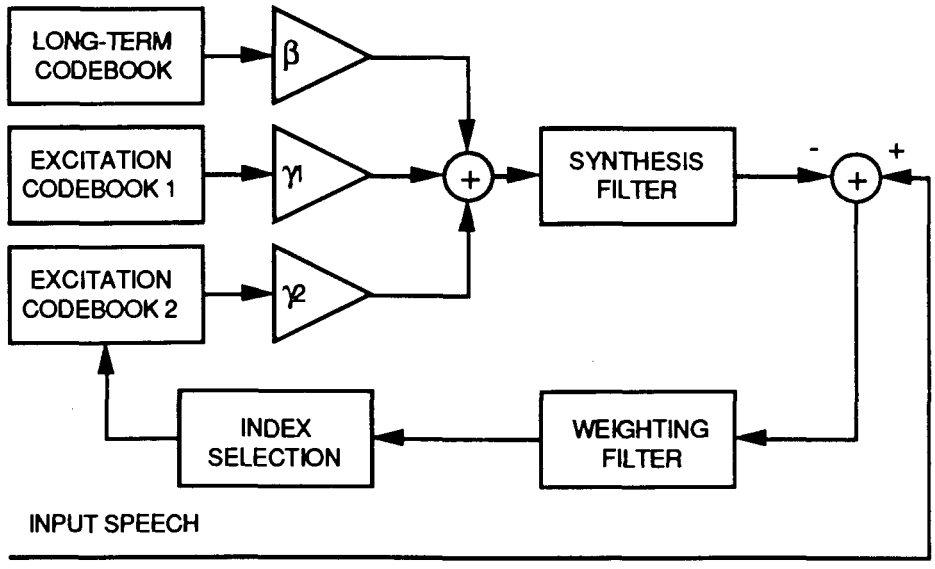

Figure 4—Block Diagram of VSELP

Whereas the approach to the LLD-VXC and LD-CELP algorithms is fairly intuitive, the approach to VSELP is much more mathematical. The encoder consists of three codebooks, one adaptive codebook for long-term prediction, which replaces the pitch prediction filter, and two excitation codebooks. The three codebooks are searched sequentially. The long-term codebook implements one-tap pitch prediction:

$$x_{PRED}(n) = \beta x(n-L) \tag{5}$$

where β is the long-term filter coefficient and L is the lag (pitch period). The codebook stores the past values of the reconstructed speech signal; for different possible lags, different vectors are retrieved. The codebook is searched to determine the optimal lag; this operation is equivalent to computing the pitch period in LLD-VXC.

The excitation codebooks, on the other hand, are highly structured to simplify the search procedure; each codevector $u_i(n)$ is constructed from M=7 basis vectors $v_m(n)$ using the following approach:

$$u_i(n) = \sum_{m=1}^{M} \theta_{im} v_m(n) \tag{6}$$

where θ_{im}=+1 if bit m of codeword i is 1, and θ_{im}=–1 if bit m of codeword i is 0. That is, each codevector is constructed as the sum of the M basis vectors where the sign of each basis vector is determined by the state of the corresponding bit in the codeword (index).

Each excitation codebook contains $2^7=128$ codevectors, resulting in a total of 2^{14} possible codevectors in all.

The high complexity of performing an exhaustive search with an unstructured codebook of this size is prohibitive; however, the use of two structured codebooks and of an efficient mathematical method for searching the codebooks makes the VSELP approach quite practical. The long-term codebook is searched first, to determine the optimal lag and select a vector consisting of past reconstructed speech samples. Then the first excitation codebook is orthogonalized from the selected past-speech vector. This operation has a low complexity because only the basis vectors need to be orthogonalized from the selected vector; also, because the codevector filtering is a linear operation, only the basis vectors need to be filtered. This codebook is searched to determine the optimal codevector. Next, the second excitation codebook is orthogonalized from both previously chosen vectors, filtered, and searched. Finally, the gains associated with each codebook are jointly optimized.

The advantages of the VSELP approach include an extremely efficient codebook search procedure, low codebook storage requirements, more robustness to channel errors than other types of codebooks, and efficient joint optimization of the codeword and the long-term predictor coefficient. A problem with using multiple codebooks is that the gains must be optimized jointly with the codevectors for optimal performance, which would result in an impractically high complexity. By optimizing the gains jointly with each other but independently of the codevectors, the results are suboptimal and the performance is degraded slightly but the complexity is reduced considerably. The gains are jointly quantized using another vector quantization operation.

The synthesis filter is a 10-pole filter, and the perceptual weighting filter is a 10th-order pole-zero filter. Additionally, the weighting filter W(z) is related to the synthesis filter A(z) in such a way that combining the two results in a simpler filter:

$$A(z) = \frac{1}{1-\sum_{i=1}^{Np}\alpha_i z^{-i}}, \quad W(z) = \frac{1-\sum_{i=1}^{Np}\alpha_i z^{-i}}{1-\sum_{i=1}^{Np}\alpha_i \lambda^i z^{-i}} \tag{7}$$

$$H(z) = A(z)W(z) = \frac{1}{1-\sum_{i=1}^{Np}\alpha_i \lambda^i z^{-i}} \tag{8}$$

The filter coefficients are updated every 4 vectors using a lattice adaptation technique. Because this algorithm is forward adaptive, not only the VQ index but also the filter parameters, long-term lag, and gain parameters must be transmitted to the receiver.

I also performed a detailed analysis of the computational complexity of the VSELP algorithm; the results are shown in Table 4. Because this algorithm has a data rate of 8 kbit/s, half that of the other algorithms discussed, one would expect that the complexity would be significantly higher; however, the results show that this is not the case. In fact, the complexity is comparable to that of LLD-VXC. The low complexity is a result of the highly specialized codebooks used.

Table 4—Complexity Analysis of VSELP

Operation	fp ops	freq (kHz)	Comp (Mflops)
Input filtering, etc.	35	8	0.28
Synthesis filter adapter	3500	0.05	0.175
Long-term predictor lag	13700	0.2	2.75
Basis vector filtering	5000	0.05	0.25
Codebook search	7400	0.2	1.48
Gain coefficient quantization	8200	0.2	1.64
TOTAL			6.6

The total complexity of the VSELP encoder is 6.6 Mflops. The decoder consists of the excitation codebooks and a synthesis filter, as for the other algorithms; it also contains a pitch prefilter and a spectral postfilter. The prefilter is used to enhance the periodicity of the excitation signal; the postfilter is used to enhance the perceptual quality of the reconstructed speech. The complexity of the decoder is only 0.4 Mflops, much lower than that of the encoder.

The complexity of this algorithm is not distributed in the same way as that of the other algorithms. The excitation codebook search complexity is much lower because a structured mathematical search is used instead of an exhaustive search. As well, the codevector filtering operation has a low complexity because only the basis vectors need to be filtered. On the other hand, the search of the long-term lag codebook has a high complexity because this codebook is not structured. The gain coefficient vector quantization also involves an unstructured search.

The custom architecture presented in this thesis is based primarily on the LLD-VXC algorithm. Chapter 8 discusses the implementation of the LD-CELP and VSELP algorithms on the new architecture and shows that certain sections of these algorithms are also implemented efficiently whereas other sections are not well suited to the architecture.

3. Mapping the Algorithms onto Hardware

The LLD-VXC and LD-CELP algorithms are very similar in structure, the only significant difference being the adaptation routines. Therefore, both of these algorithms were considered when developing the new architecture.

The most frequently used operations in the algorithms include filtering, magnitude-squared, inner product, autocorrelation, and convolution computations. Each of these may be expressed as a sum of products. Typically, these sums of products can be expressed in a vector format, so that the products can be computed in parallel. Therefore, a parallel architecture would appear to meet the requirements of these algorithms.

The complexity analysis of the algorithms shows that the codebook search has the highest computational complexity. Therefore, an efficient way of implementing this routine is also desirable. This chapter discusses the mapping of the required operations onto a VLSI architecture.

3.1 The Filtering Operations

The most commonly used operation in the algorithms is filtering, including lattice, pitch prediction, and pole and zero direct-form filtering. These operations are similar in structure and may all be vectorized; the products may be computed in parallel.

A basic inner product operation may be written in vector format as

$$z = \sum_{i=1}^{n} x_i y_i = \mathbf{x}^T \mathbf{y} \tag{9}$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors of dimension n. The direct-form filtering operation may then be written as

$$y(n) = \sum_{i=0}^{z} g_i x(n-i) + \sum_{j=1}^{p} h_j y(n-j) = \mathbf{g}^T \mathbf{x_{rev}} + \mathbf{h}^T \mathbf{y_{rev}} \tag{10}$$

where p is the number of poles, z is the number of zeros, and $\mathbf{x_{rev}}$ and $\mathbf{y_{rev}}$ are the vectors $\mathbf{x}$ and $\mathbf{y}$ with time-reversed indices. Long-term prediction, based on the pitch period, is also implemented as a direct-form filtering operation; however, the delay is larger—instead of using the most recent samples in the inner product, it uses samples from the previous pitch period:

$$y(n) = \sum_{j=L-q}^{L+q} h_j y(n-j) = \mathbf{h}^T \mathbf{y}_{K,rev} \qquad (11)$$

where L is the pitch period and q is the number of poles. Typically, q=1; that is, there are three poles in the predictor. The inner product operation can be implemented with a basic multiply-accumulate structure, as shown in Figure 5.

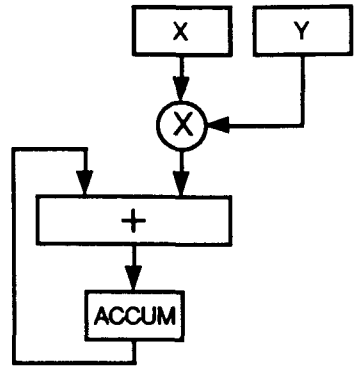

Figure 5—The Basic Multiply-Accumulate Structure

By using several of these structures in parallel, it is possible to reduce the execution time at the expense of the chip area. The power consumption can be expected to remain relatively constant. For example, by doubling the number of such structures, the required power per cycle is approximately doubled, but the number of cycles required to implement the operation is approximately halved. The exact results depend on the overheads involved. A parallel multiply-accumulate structure with four parallel multipliers is shown in Figure 6.

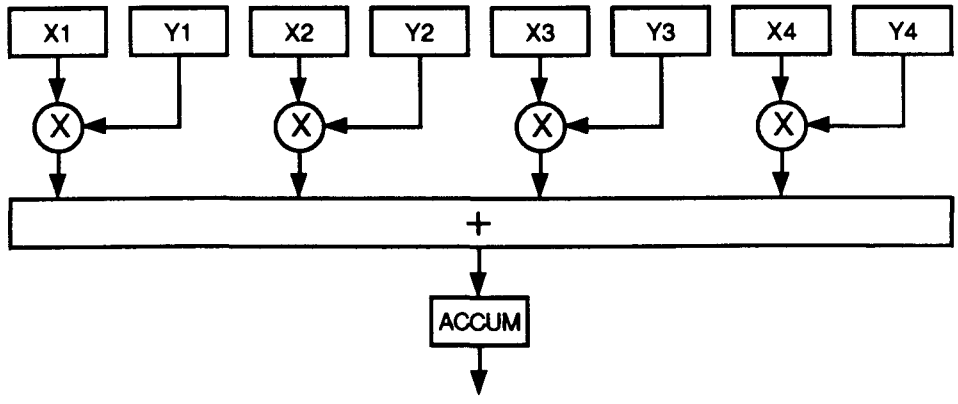

Figure 6—A Parallel Multiply-Accumulate Structure

For the fastest execution of the operation, one multiplier would be provided for each pole and zero in the operation. The multiple-input adder may be implemented as a tree of two-input adders. The penalty paid is that the propagation delay through the datapath is increased because of the chained adders. The z^{-1} delay required by the filtering operations is accomplished by chaining the data registers together; the input data is passed through the resulting delay line. This structure is the direct form implementation of the filtering operation.

There is a slight difference between filtering the poles and the zeros. Zero filtering does not require the output to be computed before the next computation can be begun because each output depends only on previous and current inputs. The same is not true for pole filtering. The computation of one output sample in this case is based on the previous output sample. As a result, each output sample must be fed back to the input. Figure 7 shows the feedback required to implement direct-form filtering with poles.

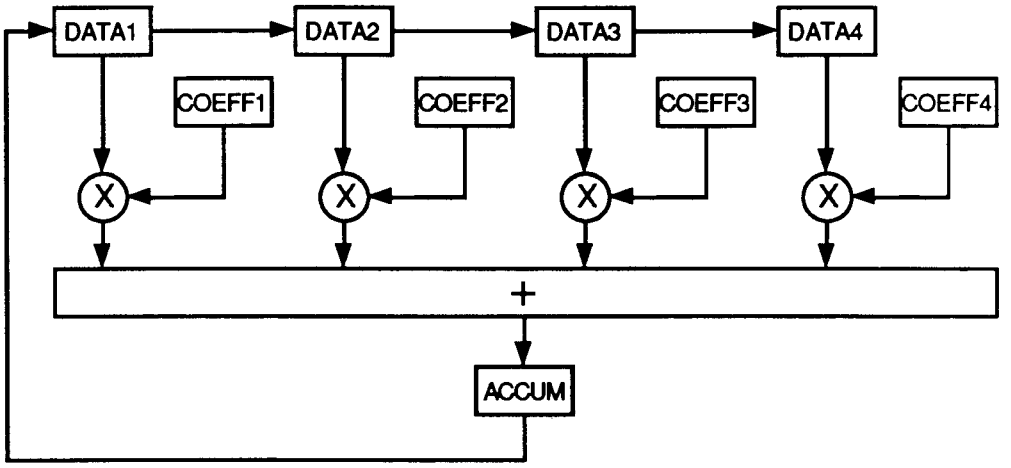

Figure 7—Direct-form Filtering with Poles

This fact has a serious impact on the pipelineability of the architecture. In a structure where there are fewer multipliers than zeros, partial sums may be computed by filtering all input data with each subset of the zeros, then summing the results. Therefore, the filter coefficients need only be replaced after all input data has been filtered by each subset. However, when there are fewer multipliers than poles, the output must be computed for each input sample, requiring the filter coefficients to be replaced for each input sample. As a result, pole filtering has a higher overhead than zero filtering and will be executed more slowly.

The convolution sum may also be expressed as an inner product:

$$z(n) = \sum_{j=1}^{N} x(j)y(n-j) = \mathbf{x}^T\mathbf{y}_{\mathbf{rev}} \tag{12}$$

This operation differs from the filtering operations only in that the function y(n) is often not defined for n<0; where such indices are required, the function is assumed to be zero, and multiplications with zero need not be computed. As a result, the number of multiplications increases with the index n. This fact is advantageous when implementing the operation on a single multiplier structure because it is not necessary to compute N multiplications for each index. However, when a parallel multiply-accumulate structure is used, this advantage no longer exists. With four multipliers, for example, the same length of time is required to compute one, two, three, or four multiplies.

Lattice filtering involves a different computation:

$$e_{j+1}(n) = e_j(n) - k_j(n)r_j(n-1) \tag{13a}$$

$$r_{j+1}(n) = r_j(n-1) - k_j(n)e_j(n) \tag{13b}$$

where $e_1(n)$ is the input and $e_{M+1}(n)$ is the output of an M-stage filter. These recursive equations are not as simple to vectorize because of the interdependence among vector components; however, the products for each stage may still be computed in parallel if the flow of data is adapted from the standard inner product calculation.

The lattice filtering operation may be implemented in a number of ways. One method is to compute the forward and backward residuals separately, as shown in Figure 8. This method implements Equations 13 directly. First, the multiplications in Equation 13a are performed in parallel; the forward residuals are calculated as a sum of the products. Next, the multiplications in Equation 13b are performed in parallel; the backward residuals are calculated by adding the backward residuals from the previous iteration. This method has the advantage that the datapath is similar to that of the other filtering operations.

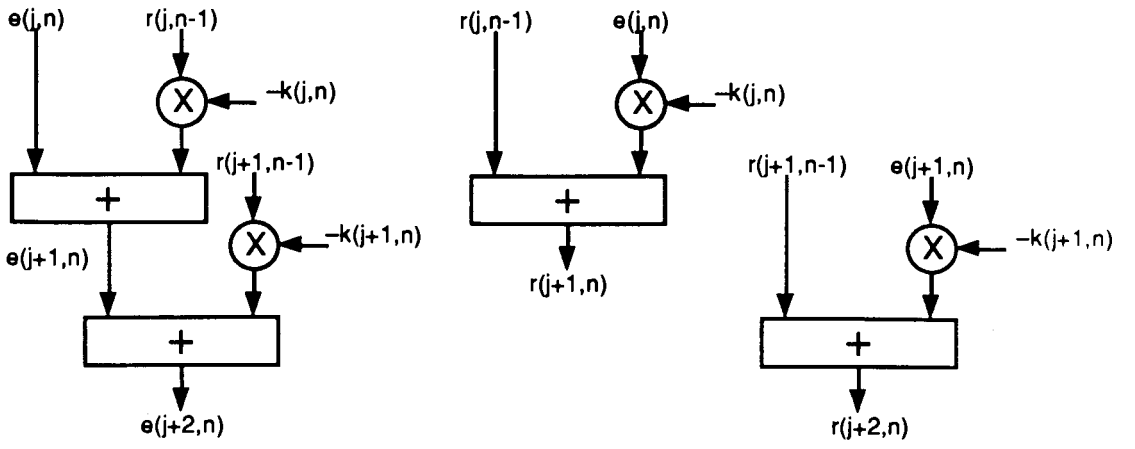

Figure 8—Lattice Filtering Based on Recursive Equations

A second method is to compute one stage of the filter, including forward and backward residuals, simultaneously. This method requires implementing the cross-connections shown in Figure 9. These connections are more natural to a lattice implementation than the previous method; however, they also have the disadvantage that the datapath differs significantly from other filter operations; in particular, only two multiplications can be performed in parallel. The first method will be used to implement the lattice filtering operations.

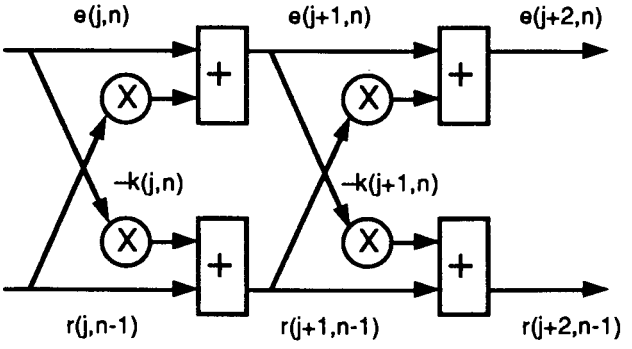

Figure 9—Lattice Filtering Based on Cross-Connection Structure

The autocorrelation function is estimated in the pitch prediction section of the LLD-VXC algorithm, where the maximum value of the autocorrelation function is determined in calculating the new pitch period, and in the adaptation routines of the LD-CELP algorithm, where the Wiener-Hopf equations are solved in calculating the optimal filter coefficients. The estimate of the autocorrelation function is computed by multiplying a signal by a

delayed version of itself, and is therefore also expressed as a sum of products. The same structure that is used for the filtering operations may be used; the filter coefficients are replaced by the input signal, and the delay line allows the function to be estimated for different delays. As a result, all of the above operations may be performed by one type of arithmetic unit, or several of these units in parallel, providing that the unit can adapt its datapath as required.

3.2 The Codebook Search Operations

Another very important section of the algorithm is the codebook search. The filtered codevectors must be compared to the input vector, and the codevector which results in the smallest distortion (distance from the input vector) must be remembered. Therefore, the codebook search involves two stages: computing the distortion measure and comparing the result to the minimum distortion value.

The distortion measure may be computed in one of several ways, each of which involves inner product calculations [12]. For example, the weighted least-square distance may by written as

$$d = \|\mathbf{x}-\mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\mathbf{x}^T\mathbf{y} \tag{14}$$

where $\mathbf{x}$ is the weighted input vector and $\mathbf{y}$ is the filtered codevector. Each of the three terms in this computation may be written as an inner product. Additionally, some algorithms, such as LD-CELP, split the codebook into shape and gain values; in this case the distortion measure must take the gain G into account:

$$d = \|\mathbf{x}-G\mathbf{y}\|^2 = \|\mathbf{x}\|^2 + G^2\|\mathbf{y}\|^2 - 2G\mathbf{x}^T\mathbf{y} \tag{15}$$

Now the inner products must be scaled by the appropriate values.

Not all three inner products must be calculated in order to compare distortion values, however. The weighted input vector $\mathbf{x}$ remains constant throughout the codebook search. As a result, the energy of this vector, $\|\mathbf{x}\|^2$, also remains constant. Only the relative magnitude of the distortion values is of interest because the object of the codebook search is to find the minimum distortion; therefore, the energy of the input vector need not be computed. The number of inner products required is thus reduced to two. Furthermore, the codevectors are constant and the synthesis filter coefficients are updated only every few vectors; therefore, the energy of the filtered codevectors can be precomputed when the filter coefficients are updated. Only one inner product need be calculated for each distortion computation.

The magnitude squared terms and the inner product are computed similarly to the filtering operations described above; however, the codebook search must also determine the minimum distortion value, which is a completely different type of operation. Each distortion value must be compared to the present minimum distortion value; if the new value is smaller, then it replaces the minimum value. This operation requires a structure like the one shown in Figure 10.

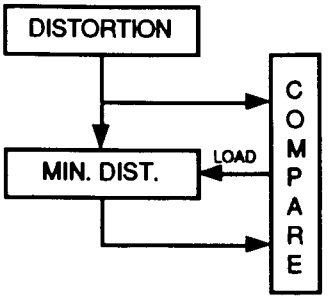

Figure 10—The Distortion Comparison Structure

This structure could further simplify the implementation of the distortion computation by including multipliers to scale the inner product and the codevector energy by the gain values as required, and an adder to compute the final distortion measure. This computation also differs from the basic filtering operations.

3.3 Implementing the Operations

A detailed analysis of several architectures, including both standard DSP chips and custom architectures, for implementing the filtering operations is discussed in [3]. The results show that a custom parallel architecture is the most promising structure. The discussion in the previous section also indicates that a parallel architecture may provide an efficient implementation of the filtering operations.

The architectures for implementing the various types of filtering operations are nearly identical, with the datapath varying slightly for different functionalities. Therefore, the same processing elements may be used and the datapath varied, to create an architecture with an adaptive datapath. The architecture shown in Figure 11 provides a great deal of functionality with a minimum of hardware.

An efficient structure for implementing the codebook search operations includes both the multipliers and adder required for the distortion computation and the comparator required for the distortion comparison.

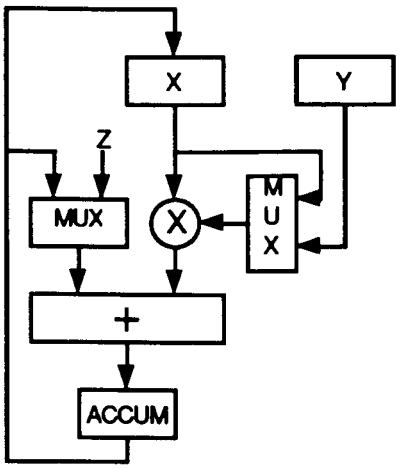

Figure 11—An Adaptive Structure for Filtering Operations

4. An Adaptive VLSI Architecture

The analysis in the previous chapter shows that two types of arithmetic units are required in the architecture. One type of unit performs all the necessary filtering operations, including lattice, pitch prediction, and pole and zero direct-form filtering. The flow of data through this unit varies for the different types of filtering; therefore, it must contain an adaptive datapath and will be called the Adaptive Arithmetic Unit (AAU). The second type of unit implements the codebook search, which is not performed efficiently by the AAUs. Its main task is to compute and compare distortion values; therefore, it will be called the Distortion Arithmetic Unit (DAU). This chapter describes these units in detail.

4.1 The Adaptive Arithmetic Unit

The AAU performs all the necessary filtering operations. These operations are similar in structure and may be written as a vectorized sum of products. Pole and zero direct-form filtering computations require a slightly different flow of data. The zero coefficients are multiplied by the previous input samples, whereas the pole coefficients are multiplied by the previous output samples; therefore, there must be a means of routing the output sample back to the input. Lattice filtering involves a different computation, but it still has a similar structure. These recursive equations are not as simple to vectorize because of the interdependence among vector components; however, the products for each stage may still be computed in parallel if the flow of data is adapted from the standard inner product calculation.

An AAU is shown in Figure 12. It consists of three pipelined stages: load, execute, and store. The processing elements in the execute stage, a multiplier and an adder, implement the multiply-accumulate required for all filtering operations. The rest of the unit implements an adaptive datapath which allows different flows of data through these elements. Although all standard DSP chips include multiplexers to load registers from different sources, the datapath in this arithmetic unit actually adapts to the requirements of the various filtering operations; the adaptation is discussed further in Chapter 6. This configuration allows varied functionality with a minimum of hardware and allows a faster execution of the algorithm than does a general-purpose DSP chip.

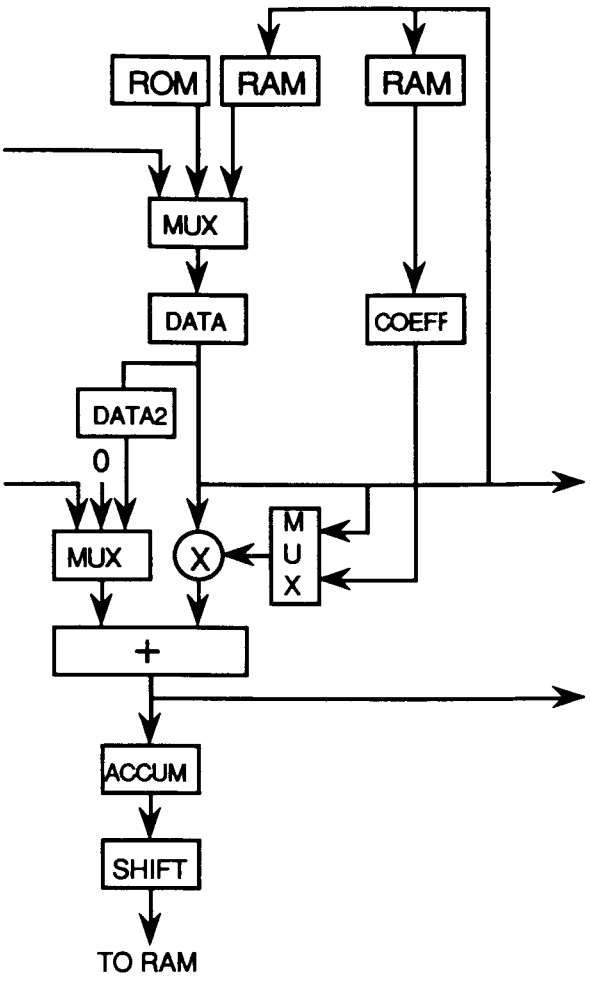

Figure 12 – Adaptive Arithmetic Unit (AAU)

The load stage contains two registers which provide the input to the multiplier; they are called the data register and the coefficient register. The names refer to the standard filtering operation which is most common—an AAU implements a filter tap, multiplying a data value by a filter coefficient. The data register may be loaded either from local memory or from a previous AAU in a chain, thereby implementing a delay line. Furthermore, the memory location may be in ROM, to access a codevector component, or in RAM, to access a temporary vector component. The coefficient register is always loaded from local memory. Separate data and coefficient memories allow these two load operations to be performed simultaneously; memory is discussed further in the following section.

The execute stage consists of a multiplier and an adder. One input to the multiplier always comes from the data register. The other input may come either from the coefficient register, to implement a filter tap or similar operation, or from the data register as well, to

implement a magnitude-squared operation. The product is one input to the adder. If only the product is required, then the second input is zero; if a sum of products is to be calculated, then the second input is the partial sum from a previous AAU in a chain. A third possibility is required to implement the backward-residual computation for lattice filtering. In this case, a data value must be added to the product; therefore, another register, DATA2, is connected between the data register and the adder.

The store stage stores the resulting sum in local memory if required. If a sum of products is being computed, only the final sum in the chain must be stored. A shifter allows the data value to be stored with any required precision.

Several of these units may be connected in parallel. Vector operations are then implemented by performing the calculations on each component simultaneously. Both the input data values and the output sums are chained together, providing a unidirectional data transfer. The data value chain implements a delay line; the sum chain implements a sum of products. When several AAUs are chained together, computing the sum of products requires time for one multiply, because each multiply is computed in parallel, and for one addition per AAU, because the additions are computed sequentially. Therefore, the required clock rate plays an important role in determining how many AAUs can be connected in the architecture. The performance results in Chapter 7 show that at the required clock rates the propagation delay through the sum of products structure is sufficiently short. Intuitively it would seem that an efficient solution would be to use one AAU for each vector component; the performance results show that this is indeed the case.

4.2 The Distortion Arithmetic Unit

The DAU computes and compares the distortion values required in the codebook search. The magnitude-squared terms (codevector energy) and the inner product of the distortion computation may all be computed by AAUs; however, the codebook search must also determine the minimum distortion value, which the AAU cannot do efficiently. The DAU computes the least-square distortion between the codevector and the input vector based on the result of the inner product calculations and compares the distortion values to determine the minimum.

A DAU is shown in Figure 13. It consists of two pipelined stages: the distortion calculation and the distortion comparison. The structure of the DAU is more rigid than that of the AAU, allowing the datapath to vary only slightly. Also, only one DAU is required.

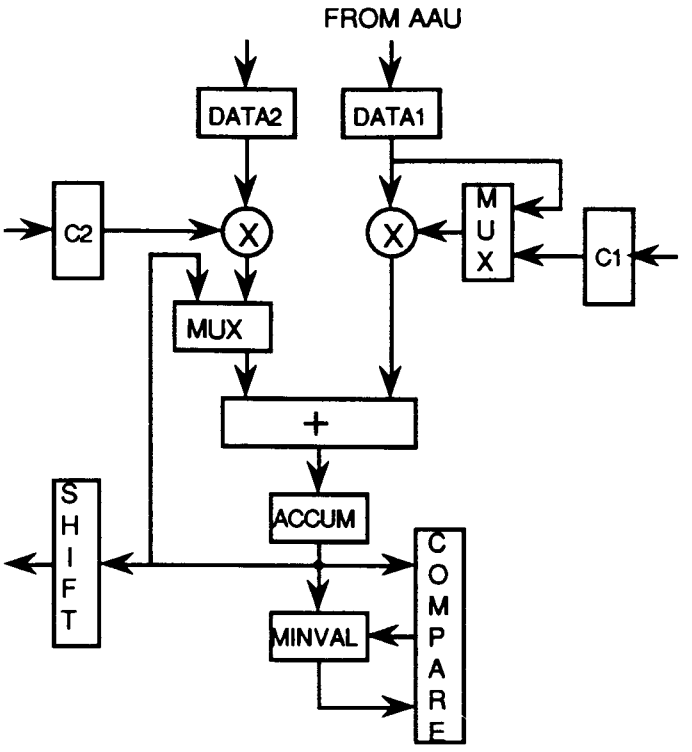

Figure 13 – Distortion Arithmetic Unit (DAU)

The first stage includes two multipliers and an adder; this stage computes the least-squared distortion, using as inputs the inner product between the codevector and the input vector, and the codevector energy. One input to the multiplier connected to the AAUs is the inner product between the codevector and the input vector, as computed by the AAUs. The second input scales this value by –2G, as required by the distortion equation, Equation 15. One input to the other multiplier is the previously-calculated codevector energy for the current codevector. The second input scales this value by G^2. In the case where the gain values are not independent of the shape codevectors, these scale factors reduce to –2 and 1, respectively. The energy of the input vector remains constant throughout the codebook search and therefore need not be considered. The two products are then summed. The result need not be stored because it is only required by the following stage.

The second stage includes a comparator; this stage determines the minimum distortion value. Each value calculated by the first stage is compared to the minimum value stored in a special register. If the new value is smaller, then the new value replaces the previous minimum value and the index to the current value is remembered.

This unit is also used in the codevector energy calculation. In the algorithms, each codevector is filtered, then its energy is calculated. Although both steps may be performed by the AAUs, the intermediate results (the filtered codevectors) must then be stored. Instead, the energy calculation is pipelined with the filtering operation using the DAU, saving memory and time. Therefore, while the AAUs filter the codevectors, the DAU acts as a simple multiply-accumulate unit. The multiplier which is connected to the AAU computes the square of each component by routing the value to both inputs of the multiplier. The product is then connected to the adder. The sum is fed back to the other input of the adder, resulting in a multiply-accumulate structure. The second multiplier and the comparator are not required for this operation. In this case, the sum needs to be stored; as with the AAU, a shifter allows the result to be stored with any desired precision.

Only one DAU is required for the codebook search. If the number of AAUs in the configuration exceeds twice the vector dimension, then multiple codebook searches could be executed in parallel, with multiple DAUs. However, such a configuration would require a large number of AAUs. The architecture is considered to contain only one DAU.

4.3 Connection of the Arithmetic Units

The high-level block diagram in Figure 14 shows how these arithmetic units are connected to form the new custom architecture. Several AAUs are chained together, the number of units being a parameter of the configuration; 4 AAUs provide an optimal implementation of LLD-VXC, as discussed in Chapter 7. The output of each data register is chained to the input of the following one, implementing a delay line. Similarly, the output of each adder is chained to one input of the following one, implementing a sum of products. The final AAU is pipelined with the DAU—the accumulator of the AAU is connected to the input of one multiplier of the DAU, allowing a sum of products computed by the chain of AAUs to be the input to the DAU. Local memory is associated with each AAU, and global memory is accessible to several of the blocks. The control system controls the operations of the blocks.

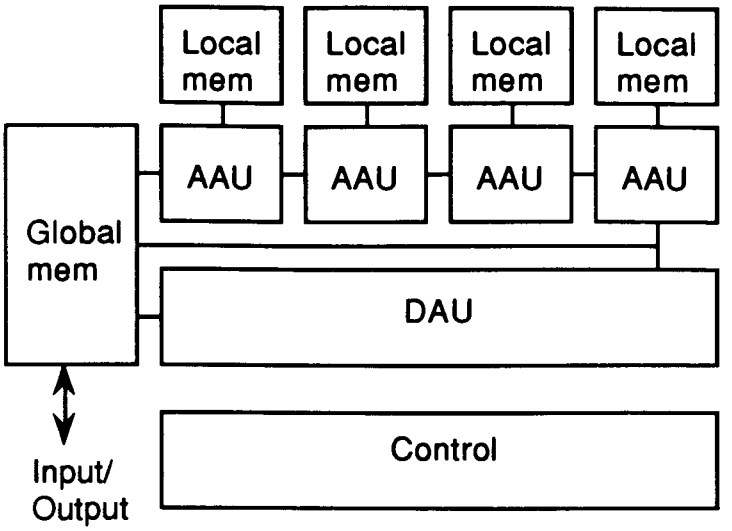

Figure 14 – Custom Architecture for Implementing Speech Coding Algorithms

The word width of the datapath and of the memory is 16 bits. The output of each multiplier is a double word; double-length accumulators follow the multiply-add structure to maintain full precision of intermediate results. A shifter scales the final result before it is truncated to 16 bits and stored in memory.

4.4 Memory

The proposed architecture includes both local and global memory. Codevector ROM provides each AAU with the appropriate codebook vector components. Local RAM stores data associated with a single AAU, such as the appropriate components of all vectors and the appropriate data and coefficients for filtering operations. Global RAM contains variables which are not specific to any AAU. The global ROM contains the program code. Thus, all required memory is located on-chip.

To store the codebooks, ROM is used instead of RAM because the original codebook does not change throughout the algorithm and because ROM requires smaller chip area and lower power. The same is true for the program ROM.

Each AAU has its own memory associated with it. The first and last AAUs also have access to global memory. These units need to read the input and store the output of the chain, which is often not distributed. The DAU has no local memory; whatever data is not passed directly from the AAUs is accessed through global memory.

Local memory can be even further distributed. When all values are stored in one RAM, multiple accesses are often required for one instruction. For example, computing the product of one filter tap requires loading the data value and the coefficient value; if these are stored in the same memory, two accesses are needed. Storing the data and coefficients in separate RAMs can be expected to increase the execution speed without significantly increasing the power consumption or chip area; the results of Chapter 7 show that this is indeed the case. Two RAMs are therefore associated with each AAU.

5. The Control System

The control unit is responsible for controlling the operation of the datapath, which has been discussed up to this point. A block diagram of the control unit is shown in Figure 15. It consists of the Program ROM, instruction decoding hardware, and flow control hardware.

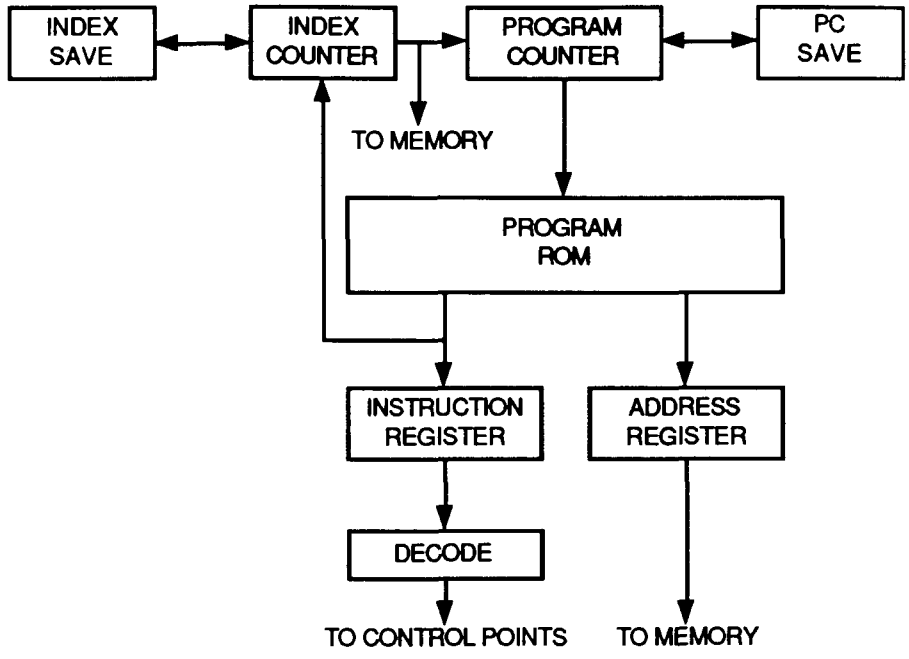

Figure 15—The Control Unit

The address of the current instruction is indicated by the program counter. The instruction is then read from the program ROM. The instruction field is decoded, and the decoded signals are routed to the required control points; the address field is routed to the various memories. Flow control hardware simplifies such operations as looping and interrupts. This chapter describes the control unit in detail.

5.1 The Program ROM

The Program ROM is 16 bits wide. A control word is divided into an instruction field (7 bits) and an address field (9 bits); the address bits are capable of accessing a memory space of 512 words. The next chapter shows that a 512-word global RAM and four 512-word local RAMs meet the memory requirements of LLD-VXC. Each address

refers either to global memory or to all local memories in parallel, depending on the instruction; therefore, this address space is sufficient.

5.2 The Instruction Set

The ASIC chip has a highly specialized instruction set. There are approximately 20 control points, such as register load and multiplexer select, in the datapath, so using a hardwired control system would require a wide control word. Instead, each the required control points for each instruction are encoded in 7 bits. This scheme allows $2^7=128$ different instructions, far more than are actually required. There are seven common instructions in the instruction set, and many of these affect the control points in the same way. In particular, corresponding control points in parallel AAUs are connected together, with the exception of the first AAU in the chain, which is often controlled differently from the others. An instruction controls several parallel operations and one instruction is always completed in one cycle. The most common instructions are listed in Table 5.

Table 5 – The Most Common Instructions

INSTRUCTION	DESCRIPTION
LOAD DATA & COEFF	Load the data and coefficient registers of each AAU in parallel, from the same address in each local RAM.
STORE DATA	Write the values in the data registers back to local RAM.
SUM OF PRODUCTS	Chain the adders together, with the input to the first being zero, and multiply; store the previous result in global RAM.
FILTER I	Chain the data registers together, with the input to the first coming from global memory; chain the adders together, with the input to the first being zero, and multiply.
FILTER II	Chain the data registers together, with the input to the first coming from global memory; chain the adders together, with the input to the first being a previous partial sum, and multiply.
SEARCH CODEBOOK	Load data registers of each AAU from codebook ROM; compute sum of products (inner product between input and codevector); compute distortion value using DAU; compare value to minimum distortion and store if less. NOTE: These four stages are pipelined, so they affect different codevectors.
FILTER CODEBOOK (includes energy calc.)	As FILTER I; also, compute magnitude-squared of result using DAU.

The memory storage is organized in such a way that the same address accesses the required values in both data and coefficient RAM in all AAUs in parallel. For example, for a filtering operation, the data and coefficient values are stored at the same addresses in the data and coefficient memories respectively.

To illustrate how the control unit and the datapath interact, the operation of the FILTER I instruction is as follows: The multiplexer in the load stage of each AAU selects to load the data register from the previous AAU in the chain; in the first AAU, this connection is made to global memory. The data register is loaded. In the execute stage, the multiplexer at the input of the multiplier selects the coefficient register as its source. The multiplexer at the input of the adder also selects the path from the previous AAU in the chain in all AAUs but the first; in the first AAU, this multiplexer selects 0. In the final AAU of the chain, the load control point of the accumulator in the store stage is also set. As a result of setting these control points, the data register is loaded from the previous AAU, the data value from the previous cycle is multiplied by the filter coefficient and the product added to the partial sum from the previous AAU, and the final sum from the previous cycle is loaded into the accumulator of the final AAU. Because of the pipelining of the AAU, the three stages affect three different data values. Figure 16 shows the datapath selected by the control unit for the FILTER I instruction.

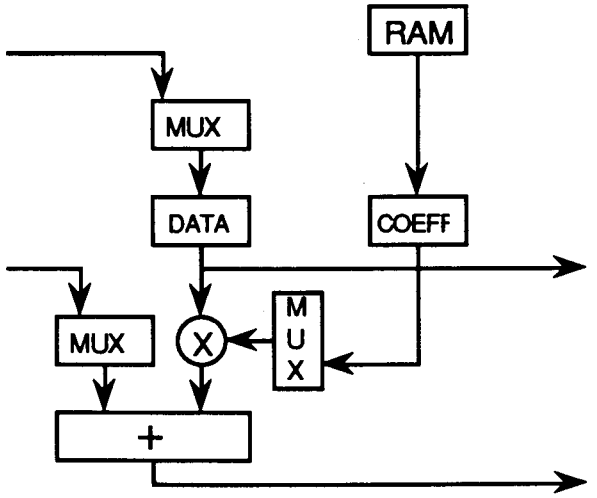

Figure 16—Datapath for the FILTER I Instruction

5.3 Hardware Support for Flow Control

The control system contains hardware support for looping and interrupts. A REPEAT N instruction allows the following instruction to be executed N times. A loop counter is initialized to N and the program counter is not advanced until the counter reaches zero; the counter is also used to address the required memory. For example, during the codebook search the SEARCH CODEBOOK instruction is repeated for each codevector in the codebook; the counter is used as an index into the local ROM to load the appropriate codevectors into the datapath. The use of a hardware counter to control the indexing allows comparisons and branching in software to be kept to a minimum.

Except for the lengthy codebook filtering and codebook search loops, however, the algorithm is implemented using straight-line code; repeated instructions are explicitly entered in the code rather than being looped. This tradeoff results in a larger program ROM but also in faster execution and simpler control hardware because little looping hardware is required. The program ROM will still be kept to a reasonable size. When loops are required, a hardware counter is used to control the indexing; therefore, comparisons and branching in software are kept to a minimum.

During the codebook search, the index of the codevector which results in the minimum distortion must be saved. Therefore, an index-save register is present in the control unit. This register is loaded from the counter by the same signal which saves the minimum distortion value in the DAU. It is the index in this register at the end of the codebook search which is actually transmitted.

Certain sections of the algorithm are not executed for each vector. For example, the filter coefficients are only adapted once every 12 vectors in LLD-VXC. Executing these routines within the time allowed for each vector would require a high clock rate. An alternative is to allow the adaptation routine to be interrupted by the vector-oriented routines. The program counter and accumulators are saved during the interrupt; the control unit provides a program-counter-save register and the accumulators are saved in local memory. The time required for the adaptation routines is thereby averaged over 12 vectors. However, this approach slightly degrades the quality of the algorithm because the adaptation routine will not necessarily be completed before the next vector must be processed.

5.4 Clock Rate Constraints

The minimum clock rate required to implement the algorithm in real time is determined by the number of instructions which must be executed for each input vector. For example, in LLD-VXC one vector consists of 4 samples taken at 8 kHz; therefore, the algorithm must process each vector in 0.5 ms. The clock rate must be high enough to allow all required instructions to be executed in this time.

On the other hand, the maximum possible clock rate is determined by how many AAUs are connected in parallel. Computing a sum of products involves a propagation through one multiplier and as many adders as AAUs; chaining many AAUs results in a large propagation delay. Because a sum of products must be computed in one clock cycle, the cycle time must be greater than this propagation delay. Conversely, a fixed clock rate determines how many AAUs may be chained together.

6. Implementation of the Algorithm

To estimate the performance results of the custom architecture, in particular the power requirements and chip area, the implementation of the LLD-VXC algorithm was studied. This algorithm is very similar to LD-CELP; however, the adaptation routines of LD-CELP make its implementation on the architecture more difficult, as discussed in Chapter 8. All of the data below refers to the LLD-VXC algorithm, unless otherwise specified. This chapter discusses the practical concerns of implementing the LLD-VXC algorithm on the custom architecture.

6.1 Memory Requirements for the Algorithm

The amount of memory required on the chip is determined by the amount of data needed by the algorithm and how the data is distributed amongst the AAUs. Table 6 shows the memory requirements of the algorithm, including whether each element of data must be stored in local or global memory.

Table 6—Memory Requirements of the LLD-VXC Algorithm

Data	Memory required (samples)	Local/Global ROM/RAM
Codevectors	256 x 4 = 1024	Local ROM
Filter coefficients	53	Local RAM
Filter memory	53	Local RAM
Temporary vectors	3 x 4 = 12	Local RAM
Codevector energy	256	Global RAM
Temporary vectors	3 x 4 = 12	Global RAM
Adaptation & tent coeffs	99	Local RAM
Past values of input	256	Local RAM
Autocorrelation values	106	Local RAM

The size of the local memory associated with each AAU depends on the number of AAUs in the configuration. The algorithm requires a fixed amount of memory; if more AAUs are present, then each has a smaller amount of memory associated with it. However, this distribution is not necessarily even. For example, if three AAUs are used to store a vector of dimension 4, one AAU must always store two components of the vector, and its memory must be twice as large as the others. (Clearly, such a configuration is not efficient!) Local ROM is used only to store the codevectors of the Vector Quantization codebook. Its size must be 1024 words for a configuration with 1 AAU, 512 words for 2 or 3 AAUs, and 256 words for 4 or more AAUs. (Recall that multiple codebook searches

in parallel are not used.) Local RAM is the most-used memory in the architecture; it stores filter memory and coefficients as well as various temporary vectors. The minimum total storage required is 512 words; however, larger RAMs will typically be used to ensure that there is enough memory for saving accumulators during interrupts and so forth.

Global RAM is used primarily to store the codevector energy. However, some temporary vectors are necessarily also stored in global memory. Note that the filtered codevectors are not stored because the energy calculation is pipelined with the filtering operation by using the DAU. Unlike with local memory, the size of the global RAM does not change as the number of AAUs varies; its size will be fixed at 512 words.

The Global ROM is the program memory; its size is determined by the number of instructions required to execute the algorithm. The program memory will be discussed further in the following section.

6.2 Program Requirements

The size of the program ROM and the required clock rate are determined by the number of instructions required to implement the algorithm with a given number of Adaptive Arithmetic Units. Because all instructions are implemented in one clock cycle, the number of instructions is nearly identical to the number of cycles required, with minor differences resulting from the looping of the codebook search and filtering.

An analysis of the number of cycles required to process one vector with the LLD-VXC algorithm is shown in Table 7. The number of cycles required is listed for configurations with one to five AAUs. The actual analysis was performed for up to ten AAUs. Note that the routines in the update section are performed only once every 12 vectors; therefore, the cycle count for these sections is divided by 12 before being added to the total. In the actual implementation, this averaging is accomplished by using interrupts, as discussed below.

For configurations with two or more AAUs, the number of cycles required to implement the algorithm, and hence the number of instructions required, is well under 2000. Therefore, a program ROM of 2K words will be used in the design.

Table 7—Number of Cycles Required to Implement LLD-VXC

Section of algorithm	1 AAU	2 AAUs	3 AAUs	4 AAUs	5 AAUs
Lattice filter (10)	200	100	80	60	40
Direct-form filter (10)	220	110	88	66	44
Long-term filter (3)	24	12	6	6	6
Lattice filter (20)	400	200	140	100	80
Direct-form filter (10)	220	110	88	66	44
Gain prediction (10)	22	12	10	8	6
Convolution	15	9	8	6	6
Codebook search (256)	1024	512	512	256	256
Lattice filter (20)	400	200	140	100	80
Direct-form filter (10)	220	110	88	66	44
ADAPTATION:					
Lattice adapter (20,10)	210	105	77	56	42
Long-term adapter	45	30	15	15	15
Pitch tracking	75	50	25	25	25
Autocorrelation calculation	400	200	134	100	80
UPDATE: (every 12 vects)					
Lattice filter (20)	400	200	140	100	80
Direct-form filter (10)	220	110	88	66	44
Codebook filtering	2560	1536	1280	1024	1024
Lattice to direct conversion	710	360	264	196	150
TOTAL	3800	1944	1559	1046	877

6.3 Timing Considerations

Because the sampling rate of the input speech signal is 8 kHz and one vector consists of 4 samples, the algorithm has 0.5 ms to process each vector. The minimum clock rate required to implement the algorithm in real time is easily calculated from the number of cycles required to process the vector: this number of cycles must require no more than 0.5 ms. From the results of Table 7, the minimum clock rates for configurations with one to five AAUs are calculated to be 7.6 MHz, 3.9 MHz, 3.2 MHz, 2.1 MHz, and 1.8 MHz respectively. All of these values are significantly lower than the clock rates of general-purpose DSP chips, which may be as high as 50 MHz.

The timing through the chain of adders is critical. The propagation delay of each adder is approximately 15 ns, and of the multiplier is 66 ns. Therefore, when n AAUs are chained together, the total propagation delay is $(66 + 15n)$ ns. For example, with 4 AAUs the minimum cycle time can be 126 ns; therefore, the maximum clock rate can be 8 MHz. The clock rates just calculated show that these constraints are not so tight as to be a problem.

The routines to update the filter coefficients are performed only once every twelve vectors. Because these routines have a significant complexity, it would be inefficient to make the clock rate high enough to handle these routines during every twelfth vector; this high rate would not be required during the other eleven vectors. Instead, the update routines are distributed over the entire vector time. They are interrupted by the vector routines in the main algorithm loop, but continue when these routines have been completed. As long as the update routines are completed within one update period (12 vectors), the algorithm will still execute correctly. There will be a slight degradation in the quality of the reconstructed speech because the filter coefficients will be updated at the end of the update period rather than at the beginning; however, the lower clock rate, and hence lower power consumption, make this solution attractive.

Computing the autocorrelation function requires a significant amount of computation. Up to 256 multiplications are required for each possible lag, and the lag varies from 20 to 105 samples, resulting in a total of almost 19000 flops. Again, the problem of implementing a high-complexity routine only during an update period appears. In this case, however, the autocorrelation computation can be distributed over each vector without the use of interrupts. As each new vector is processed, only the correlations between this and previous vectors are computed.

7. The Performance of the Architecture

Now that the custom VLSI architecture and its implementation of the Lattice Low-Delay Vector Excitation Coding algorithm have been specified, it is necessary to analyze the efficiency of the architecture in implementing the algorithm. Memory requirements have already been dealt with in the previous chapter, and execution speed has been touched upon. This chapter discusses primarily the physical characteristics of die size and power consumption.

Several factors determine an optimal number of AAUs, the most important being the vector dimension and the length of the filters. Clearly the fastest solution would be to have one AAU for each vector component and for each filter tap; however, an unrealistically large chip area would be required for long filters. The next best solution would then seem to be having a number of AAUs which evenly divides the number of components or filter taps. The performance of configurations with various numbers of AAUs was analyzed, and optimal numbers of AAUs were determined, considering power consumption and chip area. These results show that these intuitive ideas are indeed valid; 4 AAUs provide an optimal implementation of LLD-VXC.

7.1 Layout of the Architecture

To estimate the die size and chip area of the custom VLSI architecture, the architecture was analyzed using the VLSI Systems Tools [13]. The arithmetic units were entered as datapath schematics with a 16-bit word width and complied to a layout.

The technology chosen for designing the architecture was "1 μm CMOS"; this technology has a minimum feature size of 1.2 μm. An alternate technology was "1.5 μm CMOS", with a minimum feature size of 1.6 μm; however, the smaller size results in a smaller area, a lower power consumption, and a faster switching time, and therefore seemed to be the obvious choice. For example, the power of an adder/subtractor at 1.6 μm is 104 μW/MHz, while at 1.2 μm it is 71 μW/MHz.

The datapath schematic of an Adaptive Arithmetic Unit consists of several datapath elements, including flip-flops, 2- and 3-input multiplexers, an adder/subtractor, a shifter, and a multiplier. Each of these elements appears in the schematic as a 1-bit element on a 1-bit datapath; a datapath compiler then expands the schematic to the desired word width, which is set as a parameter. For these estimates, the word width is 16 bits.

The multiplier merits some discussion on its own. A signed, variable-pipelined multiplier was used as the datapath element. Both inputs to the multiplier had a word width of 16 bits. The number of stages in the pipeline may be specified; to determine if any pipelining was required at all, the datapath was first compiled with one stage. The estimated cycle length for the multiplier was determined to be 66 ns. Because the clock rate is expected to be low (2 MHz for a 4-AAU configuration), the multiplier need not be pipelined at all, and the entire multiply-accumulate operation can be performed in one clock cycle.

The output of the multiplier consists of two words because the product of two 16-bit numbers may be a 32-bit result. This output is divided into a high- and a low-order word. The accumulator following the adder is double-width to preserve precision; a shifter at the output of the adder allows the result to be scaled to the desired precision before being stored in 16-bit memory.

The next step was to compile the datapath schematic to a layout. This layout is then used to estimate the area of the block. The datapath elements are placed side by side along the width of the layout, while the bit-parallel word is built up along the height of the layout. The height of the layout is given by 100λ per bit, where λ is a function of the technology size used; for the 1.2 μm technology, $\lambda = 0.6$ μm. Clock buffers are placed above each clocked element. A large portion of the area is consumed by the interconnections among the datapath elements.

Each input and output of the block may be routed either to both sides of the block or to only one side of the block (not all to the same side). Because of the regular structure and interconnections of the blocks in the architecture, the inputs and outputs were routed to one side, resulting in a significant decrease in the size of the block. For example, the height of the AAU layout decreased from 1.5 mm for double-sided routing to 1.2 mm for one-sided routing; the width remained constant at 2.1 mm. Figure 17 shows the actual floorplan of an AAU.

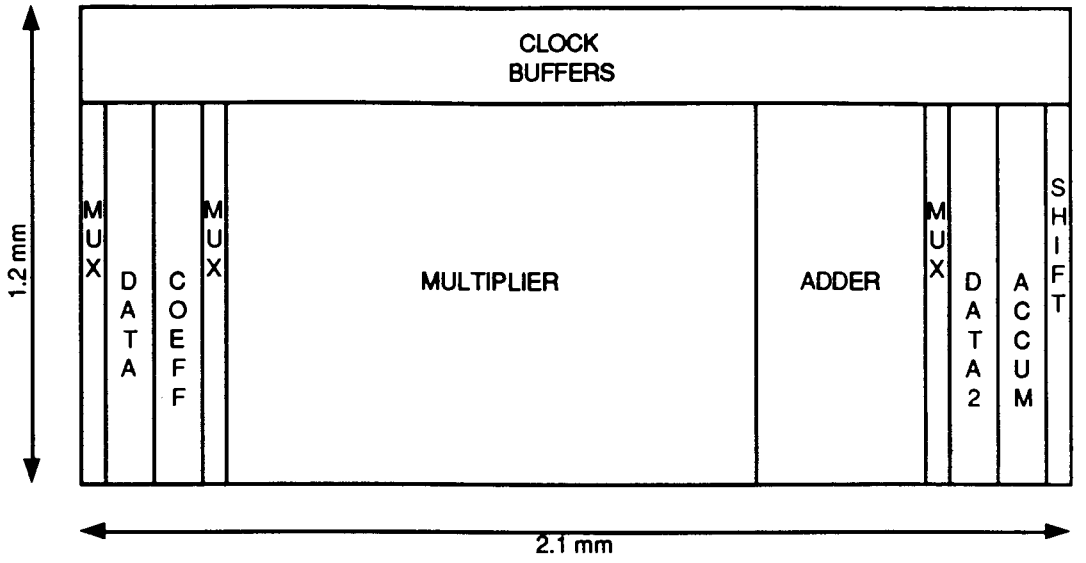

Figure 17—Floorplan of an Adaptive Arithmetic Unit

The datapath schematic of the Distortion Arithmetic Unit was developed and compiled in the same way as that of the AAU. The only element in the DAU which is not present in the AAU is the comparator. The total size of this block with one-sided routing is 3.8 mm x 1.4 mm.

The sizes for typical RAM and ROM blocks were also determined. In the previous chapter, the memory requirements for various architecture configurations were determined. The VLSI System Tools manuals provided the layout sizes of the various memory blocks.

Two types of RAMs were considered in the implementation: CRAM1 and CRAM3. The first type, CRAM1, is a clocked RAM with differential amplifiers on each output. Because there is a bias current when the clock is held low, CRAM1 has a static power dissipation. As a result, adding more RAMs is not efficient because the reduction in dynamic power due to the reduction in clock rate is typically smaller than the increase in static power. For this reason, using separate data and coefficient RAMs is not efficient.

The second type, CRAM3 is a fully static RAM; it has no static power dissipation. As well, its dynamic power dissipation is much lower than that of CRAM1. The tradeoff is that CRAM3 has a significantly larger area than CRAM1. For example, a 4-AAU configuration of the architecture consumes 530 mW of power and occupies 43 mm^2 when implemented with CRAM1, and consumes only 280 mW but occupies 56 mm^2 when implemented with CRAM3. However, power is a more important factor in evaluating this architecture than area because low power is the main advantage over standard DSP

solutions. The results using CRAM1 show that the area is quite reasonable but the power is very high. Also, using separate data and coefficient RAMs becomes feasible with CRAM3. Therefore, CRAM3 was used in the final design.

The control block requires very little hardware when compared to the arithmetic units and the memory. It consists basically of six registers and a decoder. As a result, its area and power dissipation are negligible in comparison to the other blocks.

Once the size of each block was determined, the blocks were placed manually to estimate the overall area of the architecture. An additional area of approximately 10% was allowed for routing interconnections. Pads were added with a dimension of 0.2 x 0.5 mm. An overall layout of the architecture with 4 AAUs is shown in Figure 18.

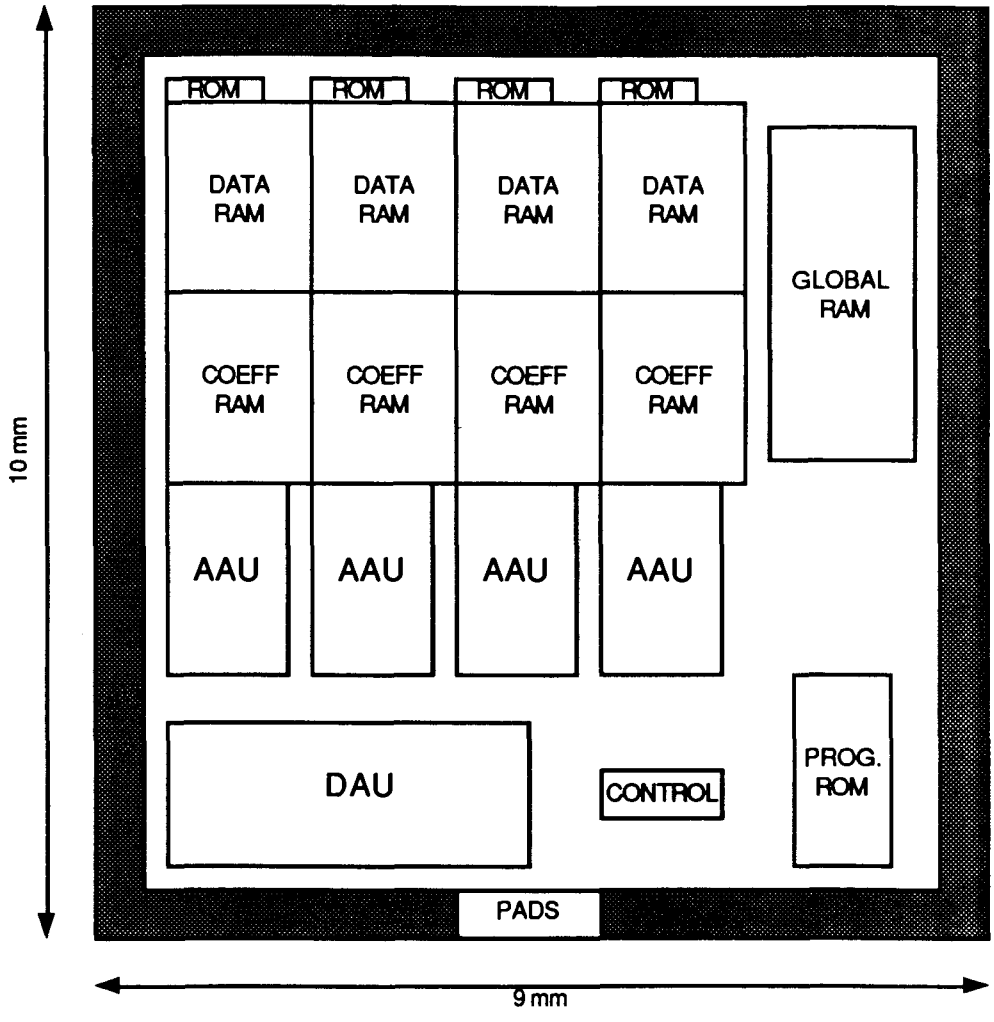

Figure 18—Floorplan of the Custom Architecture

Power figures for the arithmetic units were obtained by estimating the number of gates in an element and using the properties of CMOS technology to estimate the power consumption based on the number of gates and the clock rate, according to the formula

$$P = (0.012 \text{ mW/MHz})GSf \tag{16}$$

where G is the number of gates, S is the switching factor, and f is the clock frequency. The power estimates (0.012)G are presented in the VLSI System Tools manuals. The power depends on a switching factor—what percentage of the time the gates are actually switching. For the estimates presented here, this factor was set to 1, giving an upper limit on the power consumption. Because the architecture will be used at full capacity, the data will be changing with nearly every clock cycle and the estimates will be quite accurate.

The power consumption for the memory units was also specified in the VLSI System Tools manuals. CRAM1 has both a static and a dynamic power dissipation, whereas CRAM3 has only a dynamic power dissipation. The power figures for the ROM blocks are more complex, depending on the configuration of the ROM. Formulas for determining the total capacitances of the various lines are given, and the power consumption may be estimated from

$$P_{ROM} = IV = C_{TOT}V^2f \tag{17}$$

where V is the supply voltage of 5 V and f is the clock frequency.

The final estimates of the power consumption and chip area of the various blocks are listed in Table 8, and were first presented in [14] and [15].

Table 8 — Power and Area Figures

Block	Area (mm^2)	Power (mW/MHz)
AAU	2.1 x 1.2	20.0
DAU	3.8 x 1.4	38.4
ROM (2Kx16)	1.6 x 1.0	2.5
ROM (512x16)	1.6 x 0.5	2.4
ROM (256x16)	1.6 x 0.4	2.3
RAM (512x16)	3.5 x 1.4	1.2
RAM (256x16)	2.1 x 1.4	1.1

Using separate data and coefficient RAMs results in a decrease in power consumption and an increase in chip area. For example, for a configuration with 2 AAUs, using a single 512-word RAM results in a power dissipation of 350 mW and an area of 38 mm^2, whereas using two 256-word RAMs results in a power of 295 mW and an area

of 43 mm^2. Once again, because achieving a low power consumption is a primary goal of this architecture, separate data and coefficient RAMs are used.

7.2 Analysis of the Results

Although the architecture with one AAU implements the algorithm more efficiently than a general-purpose DSP chip, better performance is achieved by combining several AAUs in parallel. As well as increasing the throughput, this approach reduces the amount of temporary storage needed and simplifies the control requirements. By varying the number of AAUs in the configuration, tradeoffs between the power consumption and the area of the chip can be achieved. Adding AAUs creates a larger chip, but decreases the number of cycles required to implement the algorithm, thereby reducing the clock rate. To simplify the estimates, memory size was fixed at two 256-word RAMs for each AAU, regardless of how many were used; practically, smaller memories are required when more AAUs are used. A 1-AAU configuration has an approximate area of 50 mm^2, and adding one AAU increases the area by about 10 mm^2. Figure 19 shows a graph of the estimated power as a function of the number of AAUs.

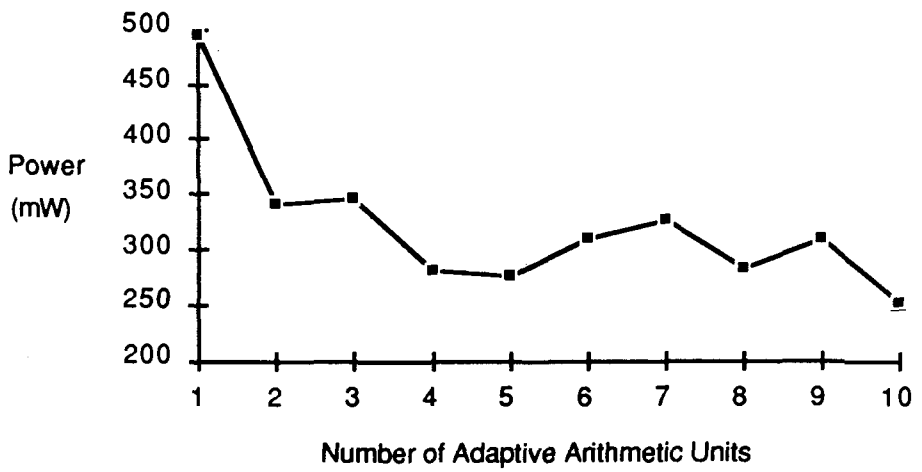

Figure 19—Power vs. Number of Adaptive Arithmetic Units

These results show that using 10 AAUs requires the lowest power consumption—approximately 250 mW; however, this configuration also has a large chip area of 180 mm^2. By reducing the number of AAUs to 4, the area is halved while the power is only increased by 10%. The clock rate required for a 4-AAU configuration is approximately 2 MHz. These results compare favorably with an implementation on a general-purpose chip such as the DSP32C, which has a power dissipation of up to 1.25 W.

At the beginning of the thesis, four criteria were set for evaluating the efficiency of an architecture: execution speed, memory requirements, die size, and power consumption. The efficiency of the new custom architecture presented here is equal to or better than that of general-purpose DSP chips based on these criteria. Using a configuration with 4 AAUs, this architecture implements the LLD-VXC algorithm in real time with a clock rate of approximately 2 MHz; a standard chip typically requires a clock rate of at least 20 MHz to achieve the same goal. The memory required by the algorithm is approximately 1K words of data RAM, 1K words of data ROM, and 2K words of program ROM; this amount of storage is comparable to what is found on standard chips. The die size of the architecture is estimated at 90 mm^2; this is a large area, but still quite reasonable. The main advantage of the new architecture is its low power consumption: less than 300 mW, compared to power consumptions of over 1 W for standard chips. These results show that the new custom architecture does provide an efficient implementation of the LLD-VXC speech coding algorithm.

8. The Suitability of the Architecture for Other Algorithms

The architecture presented in this thesis was optimized for the implementation of the LLD-VXC speech coding algorithm; however, the implementation of other speech coding algorithms and of non-speech coding algorithms was also studied briefly. This chapter discusses the suitability of the architecture for these other algorithms.

8.1 Implementation of LD-CELP

The implementation of the LD-CELP algorithm was studied. This algorithm has been implemented on a DSP32C [16], requiring approximately 75% of the capability of the chip with an 80 ns instruction cycle. Therefore, its power consumption can be estimated at slightly less than 1 W. The memory requirements were 1100 words of program memory, 900 words of data RAM, and 800 words of data ROM.

On the custom architecture its implementation is not as efficient as that of LLD-VXC. The LD-CELP algorithm uses Levinson-Durbin recursion to adapt the filters. This routine converts the autocorrelation coefficients to linear predictor coefficients. The approach is recursive: The predictor coefficients at each stage of the routine are based on those at the previous stage. As a result, the parallelism of the architecture does not improve the implementation performance because successive stages cannot be computed in parallel. Also, the pipelining of the architecture cannot be exploited because the computations of each stage must be completed before those of the next stage can be started.

Except for the adaptation routines, the LD-CELP algorithm is implemented similarly to LLD-VXC, with the result that the power of the custom implementation (estimated at 800 mW) is still lower than that of the DSP32C implementation.

8.2 Implementation of VSELP

The implementation of VSELP was also studied. However, the architecture is geared towards low-delay speech coding algorithms, not towards a specific, highly specialized algorithm which does not even meet the low-delay criterion, such as VSELP. Therefore, it is expected that certain sections of this algorithm will not be implemented efficiently.

The adaptation of the synthesis filter parameters is implemented as a fixed point covariance lattice technique (FLAT). This routine involves the computation of the

autocorrelation function of the input speech and of the reflection coefficients for each stage of a lattice filter used to adapt the filter. The autocorrelation is efficiently implemented by the custom architecture; however, the method of computing the reflection coefficients from the autocorrelation that is used by FLAT involves several calculations which cannot exploit the parallelism of the custom architecture.

The optimization of the long-term lag is based primarily on operations which can be expressed as inner products. The codebook search involves the filtering of basis vectors and the computation of cross-correlations and energies, both of which are performed in all three algorithms analyzed in the thesis. Therefore, these operations may be implemented efficiently on the custom architecture.

The final section of the VSELP algorithm is the quantization of the gain and long-term predictor coefficients. These gains are transformed to the energy domain before being vector quantized. This procedure requires the computation of reciprocals and square roots, as well as other operations which cannot be expressed as inner products. Therefore, the custom architecture is poorly suited for this section of the algorithm.

Overall, the custom architecture presented here will not implement the VSELP algorithm efficiently, as expected. This algorithm uses many mathematical techniques to reduce the complexity of the standard filtering and vector quantization operations. The efficient implementation of VSELP would require a different specialized architecture.

8.3 Implementation of the Fast Fourier Transform

Many of the important operations in digital signal processing are part of the low-delay speech coding algorithms discussed above. For example, filtering operations of all types are the most common operations in the algorithms. Clearly, these are implemented efficiently on the custom VLSI architecture because it was for these operations that the architecture was designed. However, there are also other operations which are important to DSP which are not relevant to these speech coding algorithms.

One such operation is the Fourier transform. The implementation of this operation on standard DSP chips such as the TMS32020 is well documented [17]. This reference provides the code for the general radix-2 FFT butterfly shown in Figure 20, as well as for 128, 256, and 1024-point FFTs. Some are implemented with looped code and some with straight-line code. The butterfly operation requires 22 instructions.

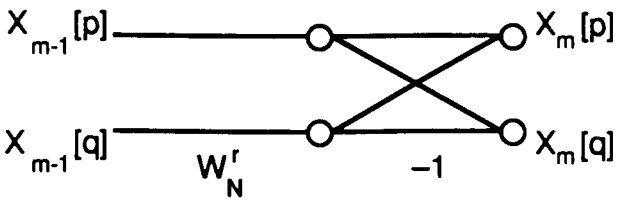

Figure 20—The General Radix-2 Decimation-In-Time FFT Butterfly

The equations for the butterfly operation are the following:

$$X_m[p] = X_{m-1}[p] + W_N^r X_{m-1}[q] \ ,$$

$$X_m[q] = X_{m-1}[p] - W_N^r X_{m-1}[q] \qquad (18)$$

The values for W_N^r are known for FFTs of a given size and may therefore be precomputed and stored. In order to perform real computations rather than complex computations, these equations may be separated into their real and imaginary components. By writing

$$W_N^r = \cos W + j \sin W \qquad (19)$$

the equations become

$$RE(X_m[p]) = RE(X_{m-1}[p]) + RE(X_{m-1}[q])\cos W - IM(X_{m-1}[q])\sin W \ ,$$

$$IM(X_m[p]) = IM(X_{m-1}[p]) + IM(X_{m-1}[q])\cos W + RE(X_{m-1}[q])\sin W \ ,$$

$$RE(X_m[q]) = RE(X_{m-1}[p]) - RE(X_{m-1}[q])\cos W + IM(X_{m-1}[q])\sin W \ ,$$

$$IM(X_m[p]) = IM(X_{m-1}[p]) - IM(X_{m-1}[q])\cos W - RE(X_{m-1}[q])\sin W \qquad (20)$$

The custom architecture can compute the required multiplications with cosW and sinW in parallel. If the architecture contains three or more AAUs in parallel, then each of the above equations can be computed in one cycle by multiplying the appropriate data values by the coefficients 1, ±cosW, and ±sinW. However, the real and imaginary parts of $X_{m-1}[p]$ and $X_{m-1}[q]$ must be stored in more than one location in the local memories of the appropriate AAUs. The FFT can then be computed in 4 cycles.

Clearly, the custom architecture is not optimal for implementing the Fast Fourier Transform. However, the parallelism of the architecture can be exploited to implement the FFT with a shorter execution time than a general-purpose DSP chip.

9. Conclusions

The new custom architecture presented in this thesis provides an efficient implementation of low-delay analysis-by-synthesis speech coding algorithms. It was designed by analyzing the requirements of two low-delay 16 kbit/s speech coding algorithms: LLD-VXC and LD-CELP. The required operations were mapped onto hardware structures and two types of arithmetic units were developed.

The most common operations in the speech coding algorithms are filtering operations of various types. All of these operations may be implemented on one type of arithmetic unit, provided it has an adaptive datapath; this unit is called an Adaptive Arithmetic Unit (AAU). Several AAUs may be connected in parallel to increase the throughput of the architecture.

The section of the speech coding algorithms with the highest complexity is the vector quantization codebook search, which involves computing a distortion measure and determining the minimum distortion value. This operation is not performed efficiently by the AAUs; therefore, a second type of unit, called a Distortion Arithmetic Unit (DAU), is required. The DAU is pipelined with the AAUs.

The custom architecture consists of several AAUs connected in parallel and pipelined with one DAU. Local memory is associated with each AAU and global memory is available to several of the units. By varying the number of AAUs in the architecture, tradeoffs between power consumption and chip area are achieved. With a configuration of 4 AAUs, this architecture implements the Lattice Low-Delay Vector Excitation Coding algorithm with an estimated power consumption of under 300 mW and area of 90 mm^2. This solution provides significant power savings over implementations on general-purpose DSP chips, which may consume approximately 1 W.

10. Future Directions

The results of this thesis show that the new custom architecture presented here provides an efficient implementation of low-delay analysis-by-synthesis speech coding algorithms. The estimates of memory requirements, clock speed, die size, and power consumption provide a good idea of the capabilities of this architecture. However, before fabricating a chip based on this architecture, there is still some work to be done. This chapter discusses possible future work on the architecture.

The primary weakness of this architecture is that it does not implement the recursive Levinson-Durbin routines efficiently. The rest of the LD-CELP algorithm is well suited to the architecture because it is very similar to the LLD-VXC algorithm. However, if the architecture could be modified to implement Levinson-Durbin recursion more efficiently, then a complete implementation of LD-CELP (soon to be the CCITT standard at 16 kbit/s) would be practical.

The implementation of other speech coding algorithms on this architecture could be investigated. Algorithms to consider include forward adaptive approaches and algorithms with different bit rates.

To determine more accurate timing and power estimates for the architecture, an extensive simulation may be performed. The VLSI System Tools provide the capability of simulating a design which was entered as a datapath schematic. The results would clarify the importance of the timing constraints discussed in the thesis and would provide more practical estimates of the power consumption than the upper limits presented here.

Additional simulation is required to analyze the performance of the algorithm on the fixed-point architecture. The issues of precision and scaling affect the overall performance of a speech coder implemented on a fixed-point processor.

Rather than implementing the algorithm with a low clock rate, both an encoder and a decoder could be implemented on one chip. Furthermore, the implementation of multiple coders is conceivable because of the low clock rate required for a single coder. However, there are several problems with this approach. First, the amount of memory increases because separate memory must be maintained for each copy of the algorithm. The data could be swapped between larger external memory and the internal memory but this technique would require a large overhead of time. Second, independent of the previous consideration, the time required to run multiple copies of the algorithm does not increase linearly because of the overhead associated with swapping operating parameters. Third,

the timing constraints due to the chain of adders place critical and severe limits on the clock rate of the chip. Attempting to implement multiple coders on a single chip would present an interesting challenge.

References

[1] Texas Instruments Incorporated, "Second-Generation TMS320 User's Guide," 1987.

[2] P. T. Whitcomb, H. M. Ahmed, "A Custom VLSI Architecture for the CCITT Wideband Coding Standard," *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 8, pp. 1492–1499, Oct. 1990.

[3] P. D. Schuler, "Implementation Study of a 16 kbit/s Speech Coder," B.A.Sc. Thesis, Simon Fraser University, Dec. 1989.

[4] J. D. Gibson, "Adaptive Prediction for Speech Encoding," *IEEE ASSP Magazine*, Vol. 1, No. 3, pp. 12–26, July 1984.

[5] R. M. Gray, "Vector Quantization," *IEEE ASSP Magazine*, Vol. 1, No. 2, pp. 4–29, April 1984.

[6] A. Gersho, V. Cuperman, "Vector Quantization: A Pattern-matching Technique for Speech Coding," *IEEE Communications Magazine*, Dec. 1983.

[7] V. Cuperman, A. Gersho, R. Pettigrew, J. J. Shynk, J-H. Yao, "Backward Adaptive Configurations for Low-Delay Vector Excitation Coding," in *Advances in Speech Coding*, B. S. Atal, V. Cuperman, A. Gersho, Editors; Kluwer Academic Publishers, Boston, MA, 1990.

[8] R. Peng, V. Cuperman, "Low-Delay Analysis-by-Synthesis Speech Coding Using Lattice Predictors," *Proc. IEEE Global Telecommunications Conference*, Vol. 2, pp. 951–956, Dec. 1990.

[9] R. G. Pettigrew, "Low-Delay Vector Excitation Coding of Speech at 16 kbit/s," M.A.Sc. Thesis, Simon Fraser University, Jan. 1990.

[10] J-H. Chen, "A Robust Low-Delay CELP Speech Coder at 16 kb/s," in *Advances in Speech Coding*, B. S. Atal, V. Cuperman, A. Gersho, Editors; Kluwer Academic Publishers, Boston, MA, 1990.

[11] I. A. Gerson, M. A. Jasiuk, "Vector Sum Excited Linear Prediction (VSELP)," in *Advances in Speech Coding*, B. S. Atal, V. Cuperman, A. Gersho, Editors; Kluwer Academic Publishers, Boston, MA, 1990.

[12] W. B. Kleijn, D. J. Krasinski, R. H. Ketchum, "Fast Methods for the CELP Speech Coding Algorithm," *IEEE Transactions on Acoustics, Speech, Signal Processing*, vol. 38, no. 8, pp. 1330–1342, Aug. 1990.

[13] VLSI Technology Inc., *VLSI Silicon Compilation System Design Manuals*, San Jose, CA.

[14] P. D. Schuler, R. H. S. Hardy, V. Cuperman, "A Custom VLSI Architecture for Low-Delay Speech Coding," *Proc. ICASSP '91 Conference*, May 1991.

[15] P. D. Schuler, R. H. S. Hardy, V. Cuperman, "Custom versus Standard VLSI Architectures for Implementing Speech Coding Algorithms," *Proc. IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, Vol. 2, pp. 639–642, May 1991.

[16] J-H. Chen, M. J. Melchner, R. V. Cox, D. O. Bowker, "Real-Time Implementation and Performance of 16 Kbit/s Low-Delay CELP Speech Coder," *Proc. ICASSP '90 Conference*, pp. 181–184, 1990.

[17] Texas Instruments Incorporated, "Digital Signal Processing Applications with the TMS320 Family," pp. 69–168, 1986.