# Systolic Architectures for Signal Compression and Discrimination

*by S-S. Yu*
*Advisor: J.F. JáJá*

Ph.D. 94-7

# Abstract

Title of Dissertation: Systolic Architectures for Signal Compression and Discrimination

Shu-Sun Yu, Doctor of Philosophy, 1994

Dissertation directed by: Professor Joseph F. JáJá
Department of Electrical Engineering

In this dissertation we propose systolic architectures for several classes of signal processing computations including schemes based on vector quantization and high order crossings techniques. The systolic concept is adapted to design architectures that are simple, regular, and that achieve high concurrency, local communication, and high throughput. Our tree-structured vector quantization (TSVQ) architecture is composed of a linear array of processors, each processor performing the computations required at one level of the binary tree. Encoding is performed in a pipelined fashion with each processor contributing a portion of the path decision through the tree until the final processor is reached to get the complete index. The predictive TSVQ (PTSVQ) architecture for real-time video coding applications uses pipelined arithmetic components to speed up the computation and to provide for regularity in design. This high throughput architecture is suitable for implementing a fully pipelined real-time PTSVQ system. Data and control flow in both architectures flow in a pipelined fashion and no global control signals are needed. We also present a class of architectures for performing signal discrimination and classification based on higher order crossing (HOC) methods. We also present a detailed design of a prototype HOC

PCB system using off-the-shelf components that can be used for non-destructive testing.

# Systolic Architectures for Signal Compression and Discrimination

by

Shu-Sun Yu

Dissertation submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1994

# Dedication

To My Parents

# Acknowledgements

First of all, I would like to express my sincere gratitude to my advisor, Professor Joseph F. JáJá, for his guidance, discussions and whole-hearted support throughout the course of my graduate study.

I would like to thank Professor Nariman Farvardin for his help and advice in many ways. My gratitude also extends to the other members of the dissertation committee, Dr. Ronald Greenberg, Dr. Linda Milor and Dr. Benjamin Kedem for their help and insightful comments. I would also like to thank Professors Hung C. Lin, Bernard Menezes, and Kazuo Nakajima for helping me in various ways during my study at Maryland.

I would like to thank Ravi Kolagotla for the valuable discussions we had for those SQ and VQs projects we worked together. Thanks are also to Hasan Fallaahadl for numerous assistance in the HOC project.

I would also like to thank those of my friends who accompanied me through my study here. For their inspiration and encouragements, I have the wonderful memories of the days we stayed together. I would like to thank Chaitali Chakrabarti, Shing-Chong Chang, Chieh-Yuan Chao, Shan Gao, Ying-Min Huang, Sridhar Krishnamurthy, Daw-Tung Lin, Jyh-Fong Lin, Po-Yang F. Lin, Zhiming Lin, Kwan-Woo Ryu, Sachidanandan Sambandan, Vishnu Srinivasan, Zhusheng Wang, and Sen-Jung Wei.

Finally, I am very grateful to my parents and brothers. Without their love and suppott, this dissertation would have been impossible.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

During the past decades, digital signal processing (DSP) systems have increasingly played a critical role in many application domains. Ever since the introduction of the fast Fourier transform (FFT) by Cooley and Tukey[1], the field of digital signal processing has progressed rapidly. Different DSP aspects have been studied and applied to an extensive number of real-world problems, involving acoustic waves, speech, image, and video signals.

Despite the fact that the designed systems are getting more powerful and more complex, the capability to handle large amounts of data is still problematic and hence sophisticated algorithms are necessary to provide the desired performance. While the processing power of general purpose workstations has increased tremendously, their data throughput rates have only increased modestly to about 10 Mbytes/sec, which is not sufficient for many important signal processing applications. Consider for example a typical real-time image processing system that has to handle 30 frames/sec, each frame consisting of $1024 \times 1024$ pixels. The throughput rate is 31.5 Mpixels/sec. It is hard to achieve such high rate using a general purpose computing system. Many real-time applications require throughput rates that are much higher. For example, HDTV demands a rate of about 70 Mbytes/sec of image data. Special purpose VLSI architectures, tailored for a particular application, can effectively balance the computational power with throughput to achieve real-time performance. Hence, special purpose VLSI systems are likely to be heavily used at least in the short run to handle processing requiring high data throughput.

It has been noticed that the most efficient signal processing algorithms have some common key properties that include regularity, simple I/O communications, and large amount of concurrency. Such properties seem to lead an effective mapping of the algorithm into compact and regular architectures.

## 1.1 Design Methodology

In this section we describe the design discipline commonly used to build special purpose systems. Under such discipline, the whole design process can proceed in an efficient manner. The methodology is divided into the following phases.

### 1.1.1 Define the problem

For any particular application, the initial phase of the design is to describe the problem in terms of functional specifications, including I/O, performance requirement etc. Other different factors such as realistic goals, feasibility, resource requirements, and technical objectives needed to be considered meeting the requirements. A brief analysis of system complexity is necessary to partition the system into sub-systems for further detailed design. Hence high-level hardware description languages, such as VHDL, are used to describe the system and to analyze it before proceeding to implement such a system.

### 1.1.2 Algorithms

The VLSI circuit technology imposes restrictions on the type of algorithms that can be implemented efficiently in hardware. An increase in efficiency can be expected, for example, if the algorithm manages to preserve a balanced distribution of work load while observing the requirement of locality. These properties of load distribution and information flow serve as guidelines to the designer of algorithms for VLSI.

Some of the critical aspects of efficient VLSI algorithms include the following:

- Maximum parallelism.
- Maximum pipelining.
- Balance among computation, communication and memory.
- Good numerical performance and quantization effects.

### 1.1.3 Hardware Architectures

For general purpose computing, the CPU is the main engine of the processor. The capability of the conventional computer architecture is limited by the bottleneck due to the shared memory that is used for both instructions and data. Most applications in DSP require the handling of signals in real time, and most of the operations in DSP algorithms are multiplications and additions. The conventional processor seems to be incapable to meet the throughput requirements in many applications. The processing power of these general purpose computers has increased greatly during the past decade. Another trend is the fact that pipelining and parallelism, once the exclusive domain of special purpose VLSI architectures, are now routinely incorporated in general purpose ASICs and workstations [2, 3].

In addition to providing a mass computational capability, a high performance architecture is expected to execute most required primitive operations at the highest speed, in order to minimize the overhead task, and to maximize throughput. Interfacing to the processor must be easy, allowing not only the processor to fit as system element, but also allowing multiple DSPs to cooperate in solving a particularly demanding task.

As mentioned earlier, the regularity, local communication and concurrency play an important role in mapping the algorithms into efficient architectures. There are several topologies, such as the pyramid network, the binary tree network, the hypercube network, and the butterfly network, that have been proposed as general-purpose parallel architectures. For instance, the systolic array architecture, initially introduced by Kung [4], provides an effective way to solve many signal processing algorithms, such as filtering, matrix operations, sorting, convolution, template matching, etc.

### 1.1.4 Physical Design

After the hardware architecture is specified, the functional blocks need to be determined. The information concerning performance, power, and size of these blocks are provided from an existing library. Based on the requirements, we can evaluate whether a single or a multi-chip solution is necessary. In chip-level implementations, the conventional ways include random logic, PLA, standard cell, gate array, FPGA, and high-level synthesis. If multiple chips solution is suitable, then further floor-planning step can proceed and Printed Circuit Board (PCB) layout is further considered.

### Functional Block

The functional block is defined as a macro of interconnected circuits to perform a specific logic functions such as Arithmetic Logic Unit (ALU), multiplier, register files, multiplexer, Random Access Memory (RAM), and Read Only Memory (ROM). For regular array of elements, such as RAM and ROM, the best way is to design the smallest cell. Then the whole memory is expanded by duplication so that the total area is optimized. For the random logic function, or finite state machine, the semi-custom design is adapted to implement those functions where the functional blocks are hierarchically mapped into gate level description in terms of NOT, NAND, XOR, etc. According to the specified schematic diagram, these library cells are placed and routed. There are trade-offs between manual and automatic routing and placement procedures. For a large number of gates, it is time consuming to implement it manually. The design by automated Computer-Aided Design (CAD) tool would save time, while it usually uses more area.

### Verification, Simulation

As the complexity of the system increases, the testing methodology becomes more and more necessary to assure the functionality of the design. The verification and simulation are done in different areas to ensure the validity of the design. The comparisons of the simulation results from different phases give the discrepancy between any two different phases. Advances in CAD tools have assisted in many aspects of the verification and testing of VLSI circuits. Also test pattern generation is a method to set up the necessary conditions in the circuit to provoke a fault condition and subsequently propagate the effect of a fault to an observable output. Other approaches include design for testability which is built-in test circuitry incorporated in the system so that we can test components and enhance the testability of the system. Hence, the correction or improvement can be effectively executed to ensure the correctness of the whole design process.

### Technology

Finally, new technologies include the inventory of the fast device and the more advanced manufacturing process. The future trends will include the implementation of low cost, low power consumption, and high density devices to integrate the maximum system and maximize the throughput of the system. On the other hand, an advanced tool suite is necessary for software development and system simulation. The CAD tools will be developed and widely used to speedup the turnaround time of the system design. For example, the state-of-art technology can squeeze hundreds of mega bits of RAM into a small area of chips. And in the current VLSI technology, up to millions of transistors can be integrated into a single chip.

## 1.2 Data Compression

The rapidly evolving communication technology will allow the generation of vast amounts of data that will have to be transmitted, manipulated or stored. The main goal of data compression schemes is in substantially shrinking the data involved without any significant loss of information.

There are several factors used to evaluate compression schemes[5]. These factors include the following.

- Bit Rate: The bit rate of the digital system is defined as the product of the sample rate and the number of bits per sample. For audio signal, the sample rate ranges from 8 kHz for telephone speech, 16kHz for teleconference, and up to 48kHz for DAT (Digital Audio Tape). Assuming 8 bits per sample, the bit rate can vary from 64k bits/sec (kps) to 384kps. As for the video coding, for example HDTV system, the sample rate is about 1280*720*60=55.3 MHz. The bit rate will be 55.3*8=442.4 Mbps.

- Performance Measurement: In spite of the fact that the signal quality is quite subjective, the discrepancy between two signals can be measured quantitatively. A distortion measure $d$ is an assignment of a nonnegative cost $d(\mathbf{x}, \hat{\mathbf{x}})$ associated when quantizing an input vector $\mathbf{x}$ with a reproduction vector $\hat{\mathbf{x}}$. Hence, the performance of a system can be measured in terms of average distortion when the input vector satisfies a certain statistical distribution. The most widely used method is the Mean-Square-Error (MSE), defined as

$$d(\mathbf{x}, \hat{\mathbf{x}}) = ||\mathbf{x} - \hat{\mathbf{x}}||^2$$

Other distortion measures include Absolute Error and Weighted Square

Error. On the other hand, due to the human perception effect considered recently, other methods, such as the method using the Mean-Opinion-Score (MOS) scale in speech quality testing, have been widely introduced in some particular areas.

- Computation Complexity: The complexity of an algorithm can be measured in terms of the number of arithmetic operations and storage elements required to execute the algorithm. There are different issues that can play a decisive role, such as, hardware or software implementation, cost, power consumption, and portability etc. In particular, advances in digital signal processing units provide up to 100 MFLOPs of peak throughput and can support the increasing demands of complicated and sophisticated algorithms.

- Communication Delay: The communication delay is often associated with the processing rate of the hardware system, and the complexity of the algorithms. For those applications such as storage, or TV broadcasting, the delay is not particularly relevant. But in two-way communication systems such as teleconferencing, the delay time poses an important restriction on the implementation of the system. Hence, the compromise is usually made between the computational complexity of the algorithm, and hence affects the signal quality.

### 1.2.1 Basic Strategies

Data compression is a fundamental signal processing task that has been the subject of extensive studies in theory and practice. In general, a signal compression procedure operates to remove the redundancy of the signal. This has led to the development of several important lossy or lossless schemes including predictive coding, block transform coding, vector quantization, and sub-band coding[6]. We will give a brief discussion on some of the lossy compression schemes.

- **Transform coding:** The transform coding method linearly transforms a block of $N$ digitized input samples, called $\mathbf{X}$, into a set of $N$ transform coefficients, called $\mathbf{Y}$. The transform coding is to convert the statistically correlated source data into less correlated coefficients in the transform domain. The coefficients are then quantized for transmission, and thus the compression can be done more efficiently. The receiver performs the inverse transform on the quantized coefficients to obtain the reconstructed signal. For example, the Fourier transform is the most widely used transform between time and frequency domain. For picture coding, Discrete Cosine Transform (DCT), Karhunen-Loéve transform (KLT), and Walsh transform (WT) can compact the energy into only a few coefficients containing most of the original information. A crucial procedure is the bit-allocation which is to minimize the mean squared error for the reconstructed signal.

- **Subband Coding:** In Subband coding, the scheme consists of a bank of M bandpass filters to operate on the input sample and to generate a set of narrowband signals which represent a subband of the input spectrum. The narrowband signals are allowed to be subsampled and reduce the bit-rate to code each sub-band. The receiver performs the inverse stages to synthesize the original subband signals. Each of the subband signals is bandpass filtered, and then they are added to reconstruct the original signal. In this approach, the input signal is divided into a number of separate frequency components, and each of these component is encoded separately. This division in the frequency domain removes the redundancy in input and provides the set of uncorrelated signals to the channel. The main advantage is that the number of bits used to encode each frequency component can be variable.

- **Vector Quantization:** A vector quantization (VQ) $Q$ of dimension $k$ and size $N$ is a mapping from an input vector in $k$-dimensional Euclidean space, $R^k$, into a set $C$, where $C$ contains $N$ reproduction points, called codevectors or codewords. The index is then sent over the channel instead of the reproduction codevector. The decoder, having the identical codebook, simply performs the table-lookup in the codebook to generate the reproduction codevector. The rate of the quantizer is $r = (\log N)/k$ bit/vector. And VQ provides the best performance among all block structured image coding schemes for a given blocksize and bit-rate.

- **Predictive Coding:** Prediction is a procedure to statistically estimate one or more random variables from observations of other random variables. Suppose a sequence $U_n$ is subtracted from the input sequence $X_n$, the difference $e_n = X_n - U_n$ is quantized for transmission, where $U_n$ is a prediction of $X_n$ based on some information from the past of $X_n$. At the receiver end, the same sequence $U_n$ is added into the quantized difference $\hat{e}_n = Q(e_n)$ to form a signal $\hat{X}_n = Q(e_n) + U_n$. The idea is to code the difference ( or error ) between the current value and the value predicted based on previous sampled values. The difference value contains less redundant information. The predictive coding is one of the most promising method for bit-rate reduction. For example, differential pulse-code modulation (DPCM) is one of the commonly used predictive coding schemes.

## 1.3 Vector Quantization

Vector quantization is one of the most powerful data compression techniques. This has led to its adoption in several standards such as the JPEG (Joint Photographic Expert Group) standard for still image compression, the MPEG ( Moving Pictures Experts Group ) standard for audio/video compression for storage applications. While a significant amount of work has been done in the development and implementation of data compression algorithms in software , not much has been done in terms of real-time hardware implementation of these algorithms [7, 8]. In this thesis, we consider and develop efficient VLSI architectures for the real-time implementation of Tree-Structured VQ (TSVQ) and Predictive TSVQ (PTSVQ). We give a brief introduction here and leave the details of the schemes for the rest of the thesis[7].

As defined earlier, VQ can be defined as a mapping $Q : R^k \rightarrow Y$ where $Y = \{Y_i; i = 1, 2, \cdots, N\}$ is the set of reproduction vectors, called **codebook,** $Y_i$ is called **codevector**, and $N$ is the number of codevectors in the codebook. Only the index $i$ of the resulting codevector is sent over the channel to the decoder. The decoder has an identical copy of the codebook as the encoder. The decoding process can be implemented by a simple table-lookup operation. The **rate** of the quantization is $R = \log_2 N$ bit per input vector. The **compression rate** is $R/k$ bits per sample pixel.

The performance of a VQ system depends on the composition of the codebook. There are several criteria that are used to design an optimal codebook.

One of the mostly used criteria is to minimize the mean-squared-error (MSE). The most popular method for generating a codebook was proposed by Lloyd and extended by Linde, Buzo, and Gray. The method uses the clustering approach, and is commonly referred to as the generalized Lloyd algorithm or the LBG algorithm. In the LBG algorithm, all the training vectors, representing the typical signal source to be coded, are clustered around the current candidates for being codevectors. The centroid of these clusters then become the new preset codevectors at the next iteration. The procedure continues until all the training vectors are clustered around the codevectors. However, it is likely that the method, depending upon the initial codebook, yields a local minimum of distortion.

For a codebook of $N$ codevectors, it take $O(N)$ distance comparisons to find out a nearest codevector in a full search VQ. For high rate coding, it becomes infeasible to implement such a VQ. There are other variation of VQ's which are developed to reduce the computation complexity at the expense of a reduced signal to noise ratio, such as Tree-Structured VQ, Multi-Stage VQ, etc. There is also a variation of VQ's which exploit the relationship between input vectors to reduce the transmitted bits or achieve the better performance at the same rate, such as Predictive VQ, and Finite-State VQ.

## 1.4 Higher Order Crossings

A time series is a sequence of observations or measurements ordered in time. Experience suggests that almost all observed time series are oscillatory, displaying the up and down property either locally or globally. This phenomenon exists in different fields and applications such as fluid mechanics, speech processing, biomedical engineering, optical communications, image processing, etc. The information contained in the oscillation of a time series can be extracted and represented by the sequence of zero crossings. It is of great interest to develop useful new techniques to explore this property and try to extract as much information from it as possible.

The simplest form of time series analysis is to count the number of zero-crossings. However, with digital signal processing techniques, filters can be applied to the sequence of the observed time series to change the oscillation and obtain a different set of counts. It is surprising that such a sequence can provide extensive information for signal discrimination, classification, and for frequency estimation in the presence of noise[9]. We refer to such a sequence of zero crossings count as HOC ( Higher Order Crossings ) sequence. This connection gives useful and interesting properties of zero-crossings, especially for the fast analysis of random signal.

Frequency estimation is an important problem in time series that has received a lot of attention in the literature[10]. There are different approaches to tackle this problem. They can be roughly classified into two categories, Fourier Transform and periodogram analysis. Though periodogram analysis can provide reasonable results in many cases, it requires a large number of iterations and a certain exhaustive search to obtain high accuracy estimate. The Fourier Transform is the traditional approach to tackle this problem. The computational complexity can be greatly reduced at the cost of resolution. However, in [10, 9], a method is suggested based on HOC sequences which provides very good results in many cases. The method, called Contraction Mapping (CM) method, exhibits a simple form that can be easily implemented. It has been shown that the method can surpass the precision of FFT results.

HOC can also be used to perform signal discrimination and classification. For example, one can apply the HOC technique to ultrasonic classification of adhesive joints in Nondestructive Evaluation (NDE)[11, 12]. The signature between perfect sample and false one can be measured using the so called $\psi^2$ statistic to measure discrepancy, where the $\psi^2$ test is related to the statistical test $\chi^2$ and will be introduced in a later chapter. The idea has been exploited in several applications such as the discrimination between white noise and stationary autoregressive moving average (ARMA) processes, and tracking the vocal sound of a whale in ambient sea noise[9].

To acquire more useful information, we can use filters, especially linear filters, to isolate specific parameters. Typically, a family of filters is applied to a time series to generate sequences of the zero-crossing counts.

In this thesis, we design a flexible special-purpose architecture for implementing a programmable HOC that can be adapted to different applications.

## 1.5 Main Contributions

In this thesis, we develop efficient VLSI architectures for implementing Tree-Structured Vector Quantizers (TSVQ) and Predictive Tree-Structured Vector Quantizers (PTSVQ) for real-time applications. These architectures can be used in any speech or image compression application based on VQ. We also develop the architectures for a system that can perform different HOC analysis schemes efficiently. In this section, we describe the TSVQ, PTSVQ and HOC algorithms, and give brief presentation of our architectures and their properties.

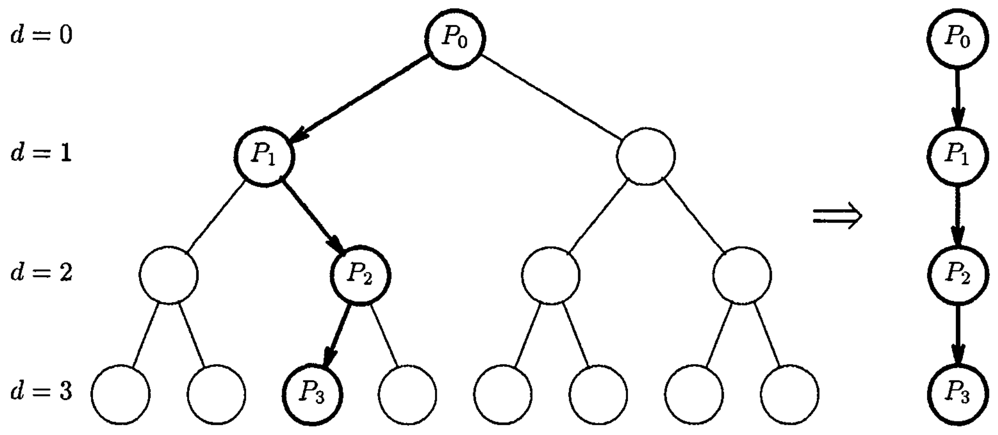


Figure 1.1: Traversal of a binary tree of depth 4, and its mapping onto a linear array of processors.

## 1.5.1 Tree-Structured Vector Quantizers

In Tree-Structured VQ (TSVQ), the codebooks are typically structured as trees to reduce the codebook search complexity and simplify hardware implementation. A TSVQ has an $O(\log N)$ codebook search complexity compared to the $O(N)$ complexity of Full Search VQ. While Full Search VQ results in a better signal to noise ratio performance than TSVQ, researchers have found that variable rate pruned TSVQs outperform Full Search VQs of the same rate [13].

The computations performed by a TSVQ can be viewed as finding a path from the root to a leaf in a binary tree. While traversing a binary tree, only one node is encountered at each level. Hence, the computations at each level can be performed by a single processor. Our architecture for the TSVQ encoder consists of a linear array of processors [14]. A tree of depth $d$ can be mapped onto a linear array of $d$ processors as shown in Fig. 1.1. Codebook values associated with each processor are stored in off-chip memories.

The number of processors required for real-time implementation equals the depth of the tree and does not depend on the input vector dimension. In our architecture all processors are identical and data flow between processors is regular and simple. Each processor performs the computations at one level of the binary tree. It then adds its result to the partial index register and transmits it to the the next processor in the array. The complete path through the tree is available from the last processor in the array. There is no global communication between the processors. Variable rate TSVQs can easily be implemented using this architecture by simply selecting the correct number of index bits at the output of the last processor.

Several researchers have implemented TSVQs in hardware concurrently with our work. TSVQ architectures were first proposed by Lookabaugh [15]. The scheme to exploit binary Hyperplane testing [16] for generating efficient TSVQ architectures was first proposed by Lookabaugh. Hardware implementations were proposed by Yan and McCanny [17] and Wai-Chi Fang *et.al.* [18] which are similar to our schemes. Yan and McCanny do not implement their architecture. Wai-Chi Fang *et.al.* use parallel multipliers to implement a tree of depth 10 on one chip. In their scheme, memory is on chip for the first eight stages. Additional memory for the last stages are off-chip. Their implementation uses a comparator in each processor which is not necessary in our scheme. Pipelined parallel multipliers have fewer logic elements between adjacent latches and are thus faster than full parallel multipliers. More recently Markas *et.al.* [19] and Madisetti *et.al.* [13, 20] have proposed TSVQ architectures.

### 1.5.2 Predictive Tree-Structured Vector Quantizers

A Predictive VQ is a VQ with memory. Predictive VQ (PVQ) makes use of interblock correlation to predict the current input vector based on past outputs; it then vector quantizes the difference between the actual input and its predicted value. As mentioned earlier, TSVQ is a sub-optimal VQ which trades off computational complexity for performance. It is easily seen that PTSVQ has a much smaller complexity than PVQ with only a minor performance degradation [7, 21].

Recently, some researchers have implemented DPCM coding with noise shaping in hardware[22]. Their scheme uses either $\frac{1}{2}$ or -$\frac{1}{4}$ as prediction coefficients which leads to a special case of prediction. The Predictive TSVQ architecture we presented combines the DPCM and Vector Quantization scheme is suitable for real-time video coding applications[23]. Pipelined arithmetic components are used to speed up the computation and to provide for regularity in design. This high throughput architecture is suitable for implementing a fully pipelined real-time PTSVQ system. Identical processors are used for both the encoding and decoding components. Spice simulations indicate correct operation at 40 MHz using $1.2\mu m$ CMOS technology. For a typical real-time image processing system with 30 frames/sec and $1024 \times 1024$ pixels/frame, the input pixel rate is 31.5 Mpixels/sec. This architecture is capable of processing 40Mpixels/sec and can handle the above case in real-time. We fabricated prototype versions of these chips using $2\mu m$ CMOS technology. These prototype chips work at 20 MHz.

### 1.5.3 Higher Order Crossings

The Higher Order Crossings (HOC) sequence is a set of parameters generated by first applying a family of filters, and then determining the corresponding zero crossing counts from the filtered signals. The HOC sequences can be used to classify signals or to discriminate between two signals. A useful statistic, the so called $\psi^2$ statistic, can be used to simply quantify the similarity between two different signals.

Though a simple schematic HOC processor is mentioned for performing difference operators and counting operation[24]. Due to the wide applicability of HOC analysis in different areas, we designed a flexible special-purposed architecture that can adapt differeent filter operations for different applications and performs the HOC analysis efficiently. We also present a detailed design of a prototype HOC-$\psi^2$ PCB system using off-the-shelf components. The designed board provides flexibility and programmability for various applications. As the prototype system uses off-the-shelf components, the performance is restricted by the microprocessor-based controller and the elements used. The microprocessor-based controller was set to run at the clock rate 11MHz. We believe that the performance will improve substantially if the system is implemented in a VLSI circuit.

## 1.6 Thesis Organization

The rest of this thesis is organized as follows: In Chapter 2, we describe the design and VLSI implementation of a systolic architecture for TSVQ. In Chapter 3, we develop an efficient architecture for implementing PTSVQ in real time for image compression, whereas in Chapter 4, we develop a systolic architecture for implementing HOC in real time for signal classification. We summarize our results in Chapter 5 and give the conclusion.

# Chapter 2

# Tree-Structured Vector Quantizers

A Tree-Structured Vector Quantizer (TSVQ) is a VQ with a structure imposed on its codebook. This structure reduces the complexity of the encoding operation, or for the same complexity, achieves a significantly better signal to noise ratio performance.

## 2.1 Definition

At each stage of a binary TSVQ, the input vector is compared with two codevectors. Based on this comparison, one of the two branches is chosen and the codebook search space is reduced in half. This process is repeated until a leaf node is reached.

Let $\mathbf{x} = (x_1, \ldots, x_L)^T$ represent the $L$-dimensional input vector, and $\mathbf{c_1} = (c_{1,1}, \ldots, c_{1,L})^T$, and $\mathbf{c_2} = (c_{2,1}, \ldots, c_{2,L})^T$ represent the two vectors in the codebook of a given node. The processing performed at each node is reduced to testing the condition:

$$d(\mathbf{x}, \mathbf{c_1}) \geq d(\mathbf{x}, \mathbf{c_2}), \tag{2.1}$$

where $d(\mathbf{x}, \mathbf{c_1})$, and $d(\mathbf{x}, \mathbf{c_2})$ are the distortion measures. For the general case of the weighted mean-squared error distortion,

$$d(\mathbf{x}, \mathbf{c_i}) = (\mathbf{x} - \mathbf{c_i})^T \mathbf{W} (\mathbf{x} - \mathbf{c_i}), \quad \mathbf{i} = 1, 2,$$

where $\mathbf{W}$ is the weighting matrix. Equation (2.1) can be expressed as:

$$(\mathbf{x} - \mathbf{c_1})^T \mathbf{W} (\mathbf{x} - \mathbf{c_1}) - (\mathbf{x} - \mathbf{c_2})^T \mathbf{W} (\mathbf{x} - \mathbf{c_2}) \quad \geq 0 \tag{2.2}$$

If equation (2.2) is satisfied, the input vector $\mathbf{x}$ is closer to codeword $\mathbf{c_2}$. Otherwise $\mathbf{x}$ is closer to $\mathbf{c_1}$. We expand equation (2.2) to obtain [25]:

$$\sum_{j=1}^{L} \{\alpha_j x_j\} + \beta \quad \geq 0 \tag{2.3}$$

where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_L) = 2(\mathbf{c_2} - \mathbf{c_1})^T \mathbf{W}$, and $\beta = \mathbf{c_1}^T \mathbf{W} \mathbf{c_1} - \mathbf{c_2}^T \mathbf{W} \mathbf{c_2}$. For the special case of the mean-squared error distortion measure, $\mathbf{W} = \mathbf{I}$, and hence $\alpha_j = 2(c_{2,j} - c_{1,j})$, and $\beta = \sum_{j=1}^{L} (c_{1,j}^2 - c_{2,j}^2)$.

Instead of using the raw codebook online, we can determine these $\alpha$ and $\beta$ coefficients off-line and store them in memory chips. Some applications use a weighting matrix $\mathbf{W}(\mathbf{x})$ that depends on the input vector $\mathbf{x}$. Equation (2.3) is still valid in this case, but a preprocessor is needed to compute the $\alpha$ and $\beta$ coefficients in real-time. The same simplification can be derived for the case of the Itakura-Saito distortion measure as well [17].

This algorithm is based on Binary Hyperplane Testing [16]. Directly implementing equation (2.1) requires $2(L^2 + L)$ multiplications, $2(L^2 - 1)$ additions and $L^2 + L$ words of memory storage, while implementing equation (2.3) requires only $L$ multiplications, $L$ additions, and $L + 1$ words of memory storage.

## 2.2 Single Node Processor

The Single Node Processor (SNP) performs the computations stated in equation (2.3). Its output is a '0' if equation (2.3) is satisfied and a '1' otherwise. The SNP contains of a parallel multiplier [26] pipelined at the bit-level, a pipelined accumulator, an index register and a counter. We do not need a comparator unit in the processor. The most significant bit (MSB) of the accumulated products directly represents the processor's output.

Fig. 2.1 shows a block diagram of the SNP. Input data is skewed and all internal operations are performed in a bit-skewed word-parallel mode. The multiplier takes two $b$-bit numbers $\alpha_j$ and $x_j$, and a $2b$-bit number $\beta'$, and returns a $2b$-bit number $p_j = \alpha_j x_j + \beta'$. We define $\beta' = \beta / L$ and add it during each of the $L$ multiplication steps. This can be done without any additional hardware and eliminates the need for a comparator unit to compare the accumulated sums with $\beta$. The bits of $p_j = p_{j,2b}, p_{j,2b-1}, \ldots, p_{j,1}$ are available in a skewed fashion, least significant bit (LSB) first. The latency of the multiplier depends on the bit position; it is $b$ for the LSB bit $p_{j,1}$, and $3b$ for the MSB bit $p_{j,2b}$. The accumulator must have a precision of

$$n = 2b + \lceil \log L \rceil$$

bits, to prevent overflow when $L$ $2b$-bit numbers are added together. The output of the multiplier is sign extended by $\lceil \log L \rceil$ bits and is directly applied to the accumulator.

The accumulator consists of a linear array of cells, and operates on skewed input data as shown in Fig. 2.2. Each cell consists of a full adder and three latches. Carry is propagated to the neighboring cell and sum is stored within

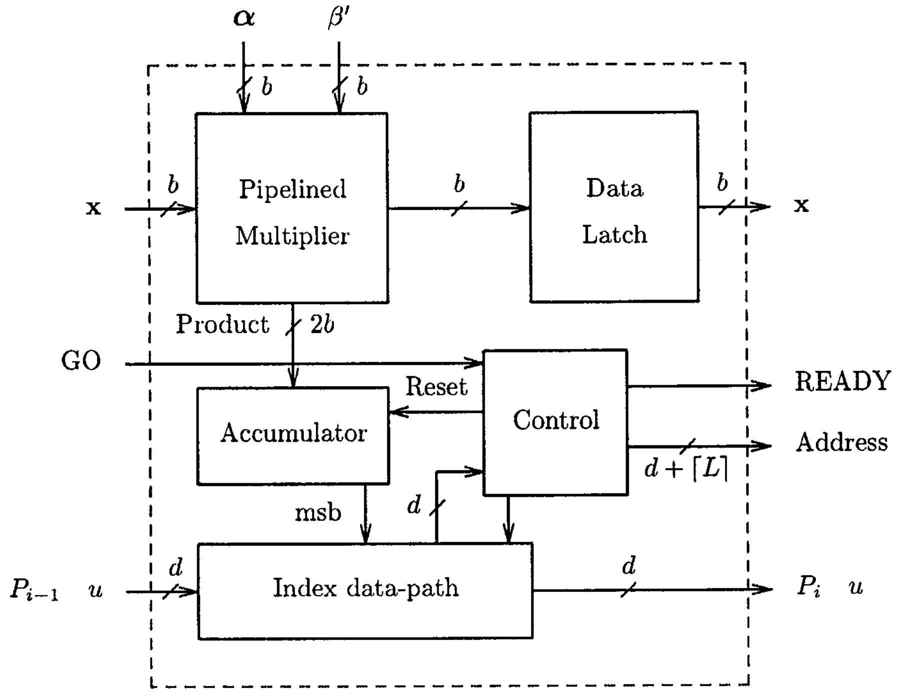
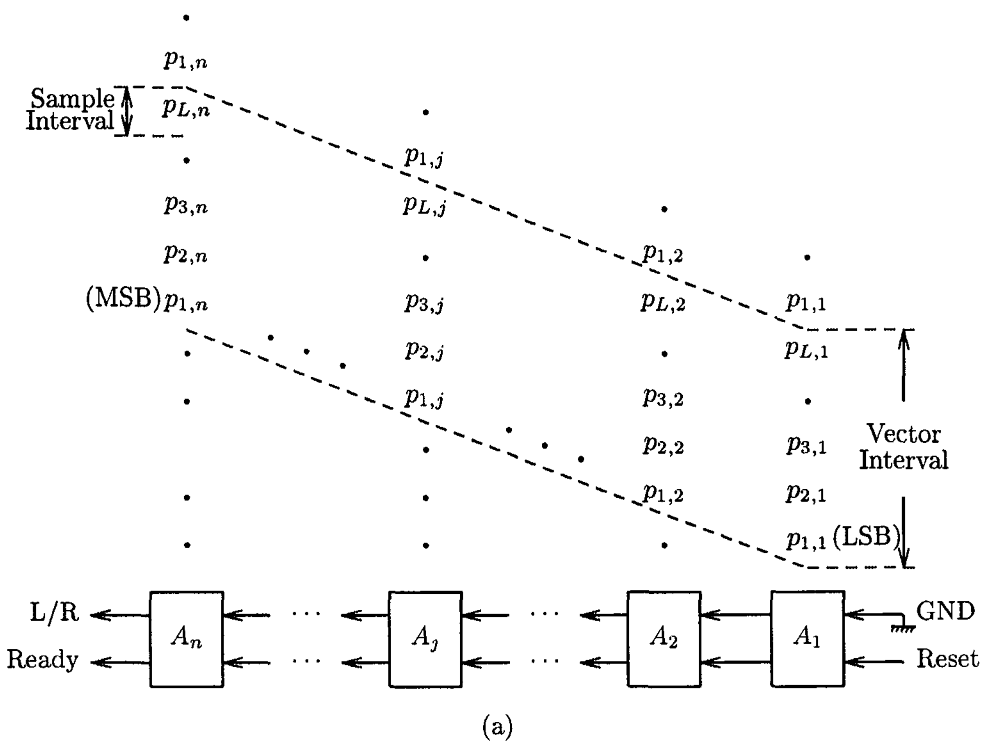


Figure 2.1: Detailed block diagram of each processor. Each processor's READY output must be connected to the GO input of its neighbor. Only the most significant $b$ bits of $\beta'$ are applied to the processor. The least significant $b$ bits are set to zero internally.



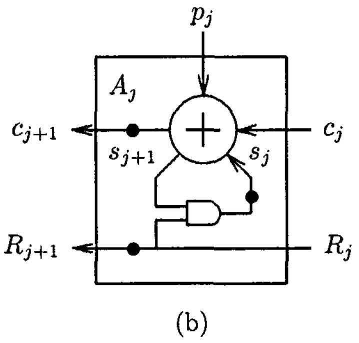


Figure 2.2: Detailed diagram of the accumulator (a) Linear array of cells. Input data is applied in a skewed fashion and carry is propagated between cells. Reset is applied to the first cell and is propagated down the array. Cells in this array are reset in a staggered fashion. (b) Detail of each cell. Solid circles are unit delay elements.

the cell. The accumulator computes

$$A = \sum_{j=1}^{L} p_j,$$

and returns the sign of A. The sign of A is available at the carry output pin of the last cell in the accumulator array. It is denoted by L/R in Fig. 2.2. A Reset signal is generated once every $L$ clock cycles. Reset is propagated along the array and each cell is reset in turn. This allows the next set of $L$ numbers to be accumulated immediately after the last number of the current set is applied to the accumulator. The latency of the accumulator is $n + L$ clock cycles. This is the number of clock cycles between the time $p_{1,1}$ is applied to cell $A_1$ and the time L/R is ready at cell $A_n$. Hence, the latency of each processor is

$$b + n + L = 3b + \lceil \log L \rceil + L.$$

For example, if the word size $b = 8$, and the vector dimension $L = 64$, we have a latency of 94 clock cycles.

## 2.3 TSVQ Architecture

The computations performed by a TSVQ can be viewed as finding a path from the root to a leaf in a binary tree. While traversing a binary tree, only one node is encountered at each level. Hence, the computations at each level can be performed by a single processor. A tree of depth $d$ can be mapped onto a linear array of $d$ processors as shown in Fig. 1.1.

Fig. 2.3 shows the architecture of a TSVQ using $d$ Single Node Processors (SNPs). The coefficients necessary for each SNP's computations are stored in memories and will in general depend on the distortion measure used. Processor $SNP_i$ adds the results of its computations to a partial index datapath and

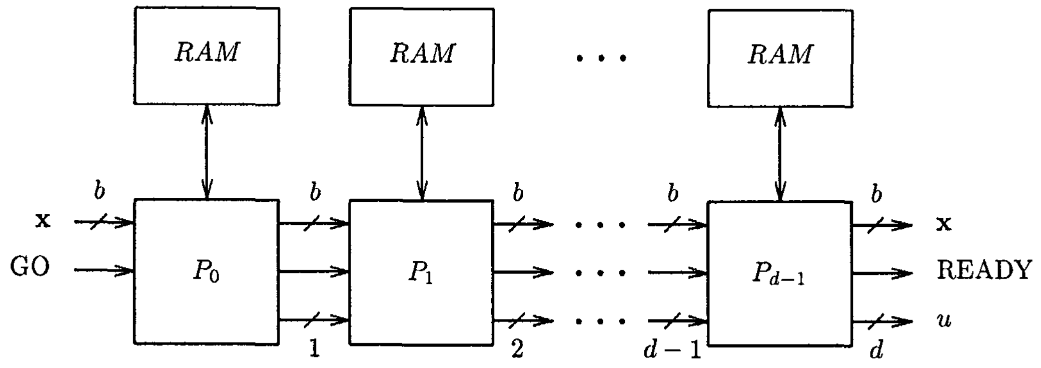


Figure 2.3: Systolic architecture for computing TSVQ. Each SNP adds its partial index to the index data-path, and generates a control signal to initiate processing by its neighbor down the tree. No global control signals are needed.

generates a Go signal to initiate processing by processor $SNP_{i+1}$. This Go signal is used to reset the accumulator in processor $SNP_{i+1}$. The final processor, $SNP_{d-1}$, returns the complete index $u$. The size of the memory is different for different processors. The first processor needs a memory of $L+1$ words to store $\beta'$ and the $L$ components of $\alpha_j$. Processor $SNP_{i+1}$ needs twice as much memory as processor $SNP_i$. The last processor needs a memory of $2^{d-1}(L+1)$ words. The throughput of this scheme is one $L$-dimensional vector per $L$ clock cycles.

A TSVQ can also be built by using one SNP and recirculating the input data $d$ times as shown in Fig. 2.4. In this case, the RAM must have an additional $\lceil \log d \rceil$ address bits to identify the level of the tree that is currently being processed. Adjacent input vectors must be separated by the latency of the TSVQ,

$$L_{TSVQ} = dL_{SNP} = d(b+n+L). \tag{2.4}$$

The throughput in this case is one $L$-dimensional vector per $L_{TSVQ}$ clock cycles. For a tree of depth $d=8$, and a vector dimension of $L=16$ (which corresponds to a bit rate of 0.5 bpp), we have $L_{TSVQ}=352$ clock cycles.

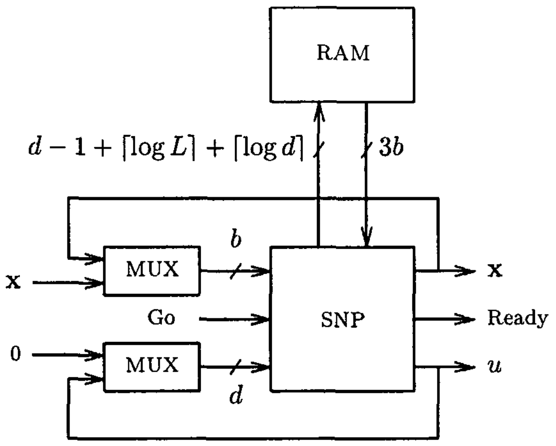


Figure 2.4: TSVQ architecture using one SNP and recirculating registers. Input vector **x** must be recirculated $d$ times, once for each level of the binary tree.

## 2.4 VLSI Implementation

The detailed block diagram of each processor is shown in Fig. 2.1. Each processor consists of a pipelined parallel multiplier, a bit-level accumulator, a data vector register, a partial index register, and a local control unit. The multiplier computes $a \times b + c$, and can process a different set of inputs each clock cycle. The products are output in skewed fashion, LSB first, every clock cycle. A bit-level accumulator adds these partial products in bit-serial fashion. The MSB of the accumulated partial products represents the processor's partial index. One of the advantages of this architecture is the absence of any comparator unit. We don't need a comparator because the multiplier can perform addition without any extra hardware. Hence we can directly implement equation (2.3) in hardware. It is not necessary to add any correction terms to the accumulator's output. The control unit keeps track of each input block of size $k \times k$ pixels and sends a reset signal to the accumulator once every $k^2$ clock cycles. The reset signal propagates through the accumulator and each of its cells resets in succeeding clock cycles. This scheme allows for the next block of skewed partial products to be accumulated immediately after the last block is applied to the accumulator. Input block sizes of $4 \times 4$ or $8 \times 8$ pixels can be quantized by this processor. An external control signal is used to select between these two modes.

A separate datapath is used to propagate the partial index through the pipeline. Each block of input vectors has a partial index tag associated with it. This partial index moves along with the input synchronously. An address for the off-chip RAM is generated from this partial index and the output of the on-chip counter. There are 8 pins in the index data path. This allows for trees of depth up to 8 to be easily constructed using these processors. These processors can also be used, together with some external logic, to build trees of depth larger than 8.

## 2.5 Simulations

This TSVQ implementation consists of one processor for each level of the tree. Interconnection and data flow between processors is simple and requires no global control signals. Fig. 2.5 illustrates the timing of all local signals between processors for the case when the block size is $8 \times 8$. The system requires a two phase non-overlapping clock. Two phase clocks avoid race conditions and permit simple logic level design. The latency time of each processor is 100 clock cycles. This includes the 64 cycles needed to read each block. If the block size is $4 \times 4$, the latency per processor is 52. Each processor generates a READY signal when its computation is completed. This READY signal also indicates the start of the

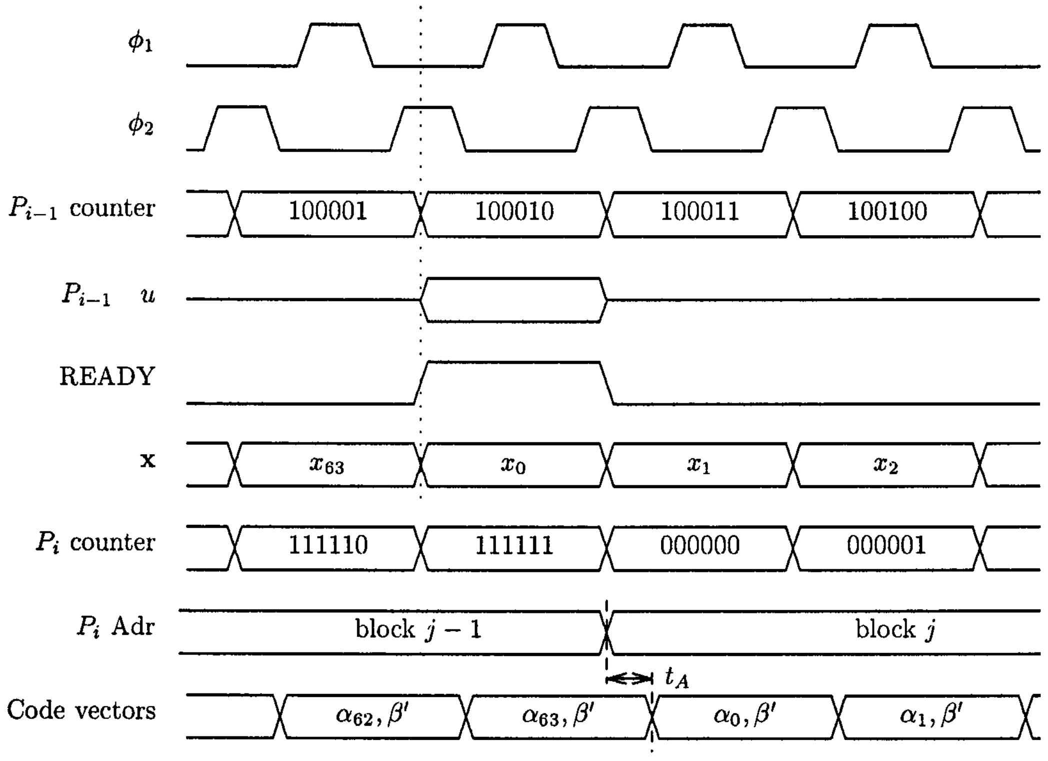


Figure 2.5: Timing diagram of signal flow between processors for input block size of 8 × 8. Dotted line shows the boundary between adjacent vectors. Coefficient memory chips must have an access time smaller that $t_A$.

delayed input vector and its partial index. This signal is used by the neighboring processor to reset its control unit. The partial index is also used as an address for the coefficient memory.

## 2.6 Fabrication and Testing

We have implemented a Single Node Processor using MOSIS' $2\mu m$ N-well process on a $7.9mm \times 9.2mm$ die [14]. Each processor contains 25,000 transistors and has 84 pins. The processors have been tested at 20 MHz. These processors can operate on either $4 \times 4$ or $8 \times 8$ blocksizes. Fig. 2.6 shows a plot of the fabricated chip.

This chip was tested using a IMS HS 1000 tester using 500 randomly generated test vectors. It was found to be fully functional at a frequency of 20 MHz. This chip has been designed using scalable ground rules. Fabricating at $0.8\mu m$ will result in an operating speed of 50 MHz.

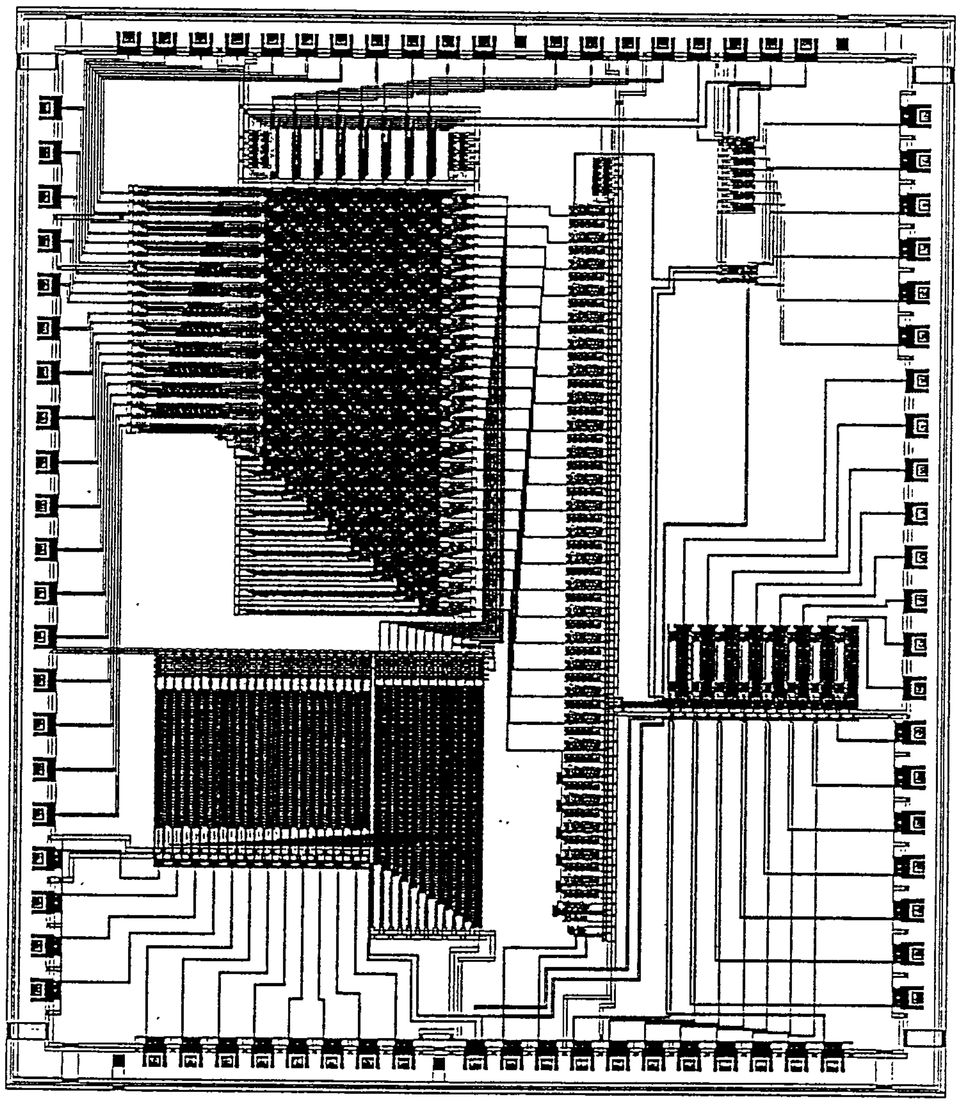


Figure 2.6: Plot of the TSVQ processor chip. Die size is $7.9mm \times 9.2mm$.

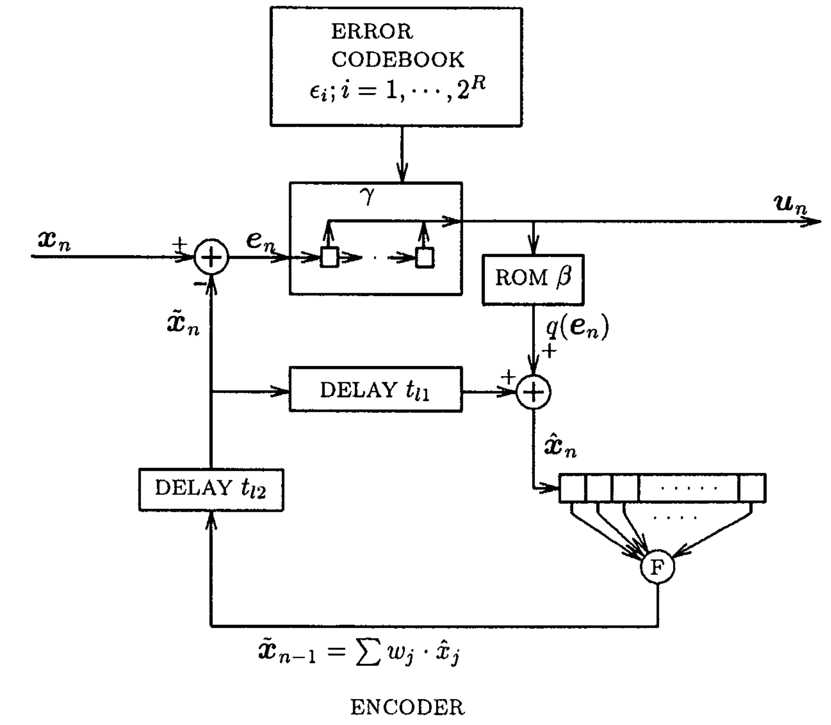
# Chapter 3

# Predictive Tree-Structured Vector Quantizers

A Predictive VQ is a VQ with memory which has been extensively studied in recent years [27, 28, 29, 30, 21, 31, 32]. Predictive VQ (PVQ) makes use of interblock correlation to predict the current input vector based on past outputs; it then vector quantizes the difference between the actual input and its predicted value. We concentrate here on the case where the difference is quantized using a TSVQ. The resulting system is called a Predictive TSVQ (PTSVQ). As mentioned earlier, TSVQ is a sub-optimal VQ which trades off computational complexity for performance. It is easily seen that PTSVQ has a much smaller complexity than PVQ with only a minor performance degradation [7, 21].

## 3.1 Definition

A block diagram of the PVQ system is shown in Fig. 1. Predictive VQ (PVQ) can be viewed as a straightforward vector extension of the traditional scalar pre-



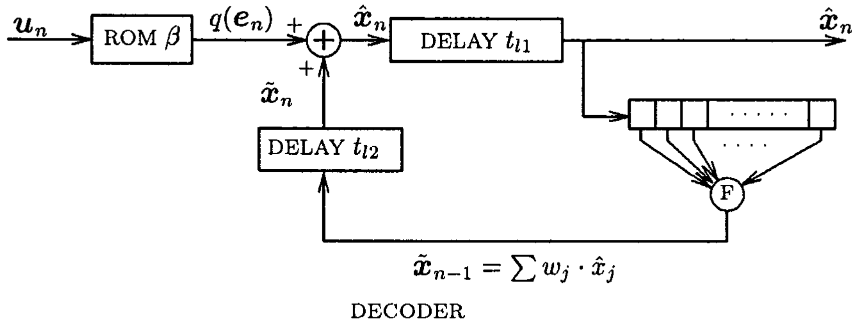


Figure 3.1: Scheme of PVQ.

dictive quantization or Delta Pulse Code Modulation (DPCM). In the encoder, a predicted vector is formed from the past reconstructed vectors, $\hat{\boldsymbol{x}}_{n-1}$, $\hat{\boldsymbol{x}}_{n-2}, \cdots$. An error vector, $\boldsymbol{e}_n$, is generated based on the difference between the predicted vector $\tilde{\boldsymbol{x}}_n$ and the actual input vector $\boldsymbol{x}_n$. This error vector is quantized using a memoryless VQ. Then, the index is transmitted over a channel.

In the decoder, this error vector is recovered from the received channel index by table lookup. The original vector is reconstructed by adding this error vector to its corresponding predicted vector. A PVQ system [27] can be formally defined as follows:

1. An encoder $\gamma$ which is a memoryless VQ that assigns to each error vector, $\boldsymbol{e}_n = \boldsymbol{x}_n - \tilde{\boldsymbol{x}}_n$, an index symbol $u_n$ from an index set $M$ to identify the closest codeword in codebook $\hat{B}$.

2. A decoder $\beta$ which is a mapping that assigns to each index $u_n$ a vector in a reproduction codebook $\hat{B}$.

3. A prediction function $f$ which predicts the input vector $\tilde{\boldsymbol{x}}_n$ based on the previous reconstructed inputs, and hence $\tilde{\boldsymbol{x}}_n = f(\hat{\boldsymbol{x}}_{n-1}, \hat{\boldsymbol{x}}_{n-2}, \cdots)$. Typically, only finite order, say $p$, of prediction is assumed to be used, i.e. the above expression can be simplified as $\tilde{\boldsymbol{x}}_n = f(\hat{\boldsymbol{x}}_{n-1}, \hat{\boldsymbol{x}}_{n-2}, \cdots, \hat{\boldsymbol{x}}_{n-p})$.

Given a sequence of input vectors and an initial prediction $\tilde{\boldsymbol{x}}_1$, the index sequence $u_n$, reproduction sequence $\hat{\boldsymbol{x}}_n$, and prediction sequence $\tilde{\boldsymbol{x}}_{n+1}$ for $n = 1, 2, \cdots$ are defined recursively as follows:

$$\begin{aligned} u_n &= \gamma(\boldsymbol{e}_n) = \gamma(\boldsymbol{x}_n - \tilde{\boldsymbol{x}}_n) \\ \hat{\boldsymbol{x}}_n &= \tilde{\boldsymbol{x}}_n + \beta(u_n), \\ \tilde{\boldsymbol{x}}_{n+1} &= f(\hat{\boldsymbol{x}}_n, \hat{\boldsymbol{x}}_{n-1}, \cdots). \end{aligned}$$

For a *linear prediction function of finite order p*, $\tilde{\boldsymbol{x}}_n$ can be expressed as

$$\tilde{\boldsymbol{x}}_n = \sum_{i=1}^{p} A_i \hat{\boldsymbol{x}}_{n-i},$$

where $A_i$ is an $L \times L$ predictor matrix for an $L$-dimensional vector.

## 3.2 Overall Architecture

In this section, we describe the mapping of PTSVQ onto a VLSI architecture for real-time image coding. This system consists of:

1. TSVQ for encoding the difference between the predicted vector and the input vector into a channel index,
2. Inverse TSVQ (ITSVQ) for decoding the channel index into its corresponding codevector, and
3. Predictor Processor (PP) which computes the residual vector, and executes the prediction process.

The overall architecture of the PTSVQ is shown in Fig. 3.2. This architecture consists of a Predictor Processor, a linear array of SNP processors which realize a binary TSVQ, and an Inverse TSVQ (ITSVQ). There are three types of basic building blocks in the system, namely the Predictor Processor (PP), the Single Node Processor (SNP) used for realizing TSVQ, and the ITSVQ processor. The ITSVQ chip can be implemented as a table lookup, using either a ROM or a PLA. The Predictor Processor subtracts the predicted vector from the input vector, buffers the past vectors, and generates the predicted vector according to the specified prediction function. Due to the similarity between the encoding

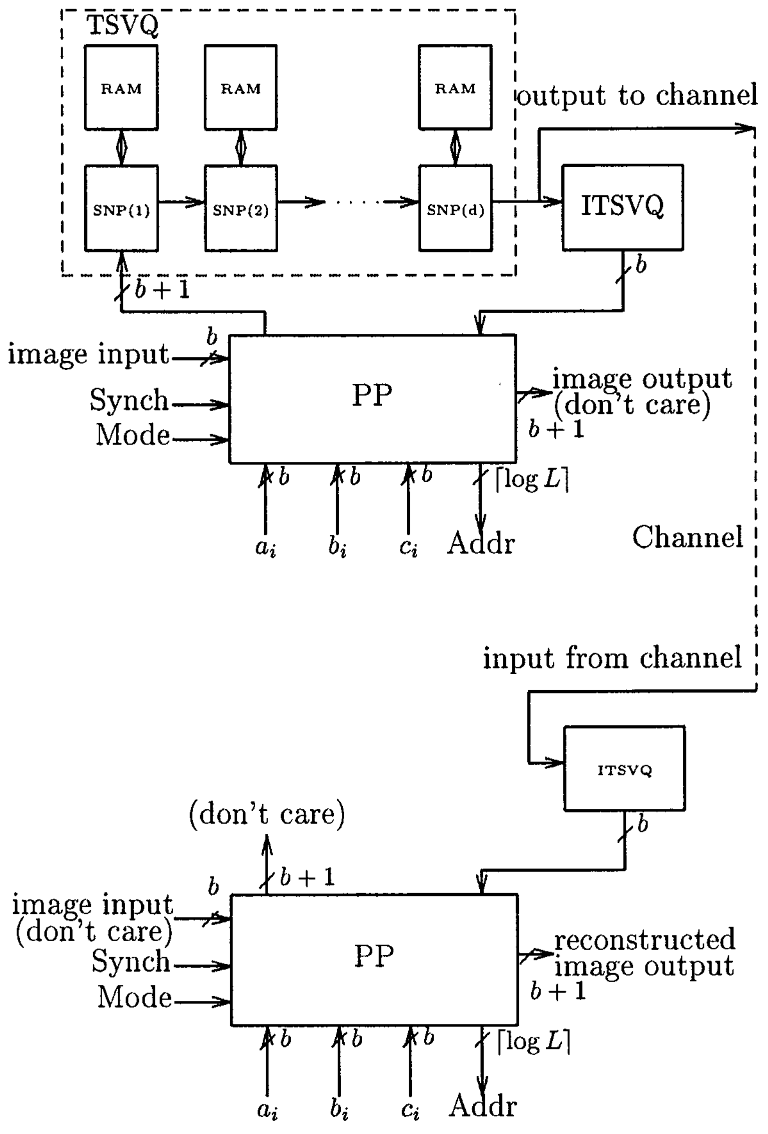


Figure 3.2: Architecture for PTSVQ system. PP is the Predictor Processor to perform the prediction function for the incoming vectors. PP is identical in both the encoder and the decoder.

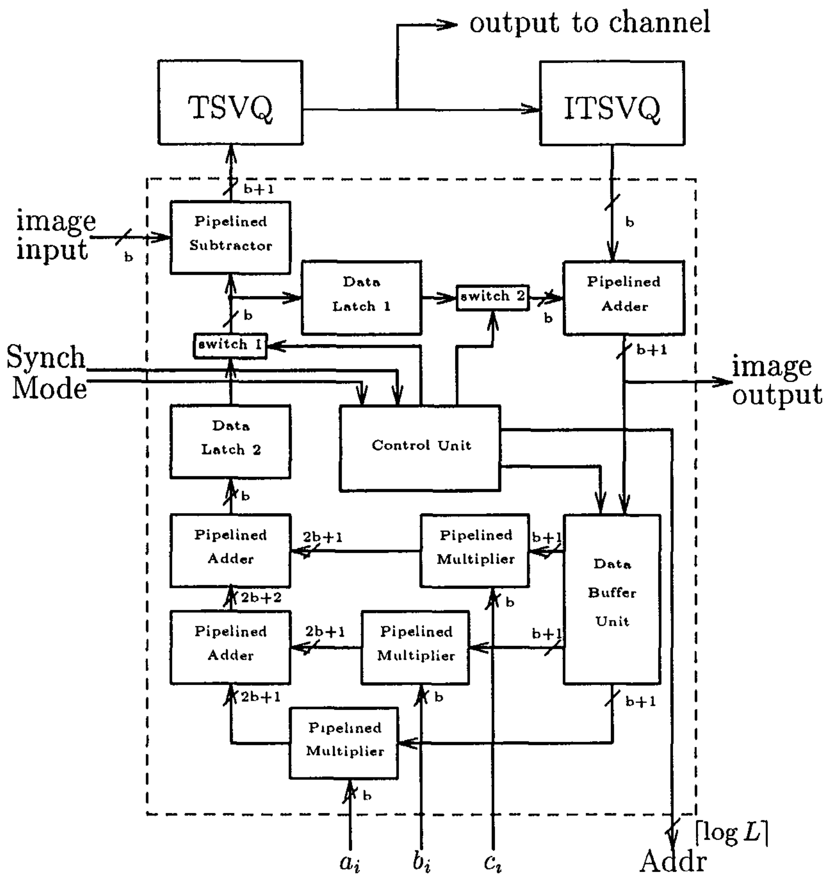
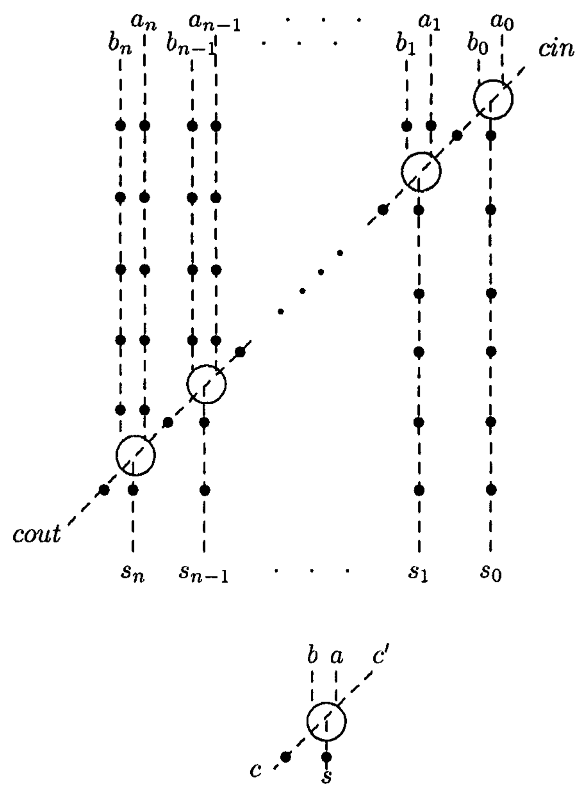


Figure 3.3: Block diagram of the Predictor Processor.

and decoding parts of the PTSVQ, the PP can be used either as an encoder or as a decoder without additional circuitry. The SNP performs the distortion computation corresponding to a node of a binary TSVQ. We now describe each of these blocks in detail.

### 3.2.1 PTSVQ architecture

In this section, we describe how the PTSVQ can be built using a TSVQ as a building block. A detailed block diagram of the Predictor Processor is shown in Fig. 3.3. A pipelined subtractor is used to subtract the values of the predicted


$a_n$ $a_{n-1}$ ... $a_1$ $a_0$
$b_n$ $b_{n-1}$ ... $b_1$ $b_0$
cin
cout
$s_n$ $s_{n-1}$ ... $s_1$ $s_0$
$b$ $a$ $c'$
$c$
$s$

$$s \leftarrow a \oplus b \oplus c'$$
$$c \leftarrow a \cdot b + a \cdot c' + b \cdot c'$$

Figure 3.4: $n \times n$ pipelined bit-serial adder with skew and deskew latches.

vector from the corresponding input vector. The bit-level adder/subtractor, shown in Fig. 3.4, operates on skewed input data in bit-serial fashion. The result, i.e. the difference or residue vector, is then deskewed and sent to the TSVQ to generate the channel index. The predicted vector is delayed by the latency of the TSVQ and the ITSVQ modules. This delayed vector is added to the output of the ITSVQ module to generate the reconstructed vector. This reconstructed vector is then fed into a data buffer unit. The data buffer unit correctly taps the pixel values from these vectors for the linear prediction module as explained next.

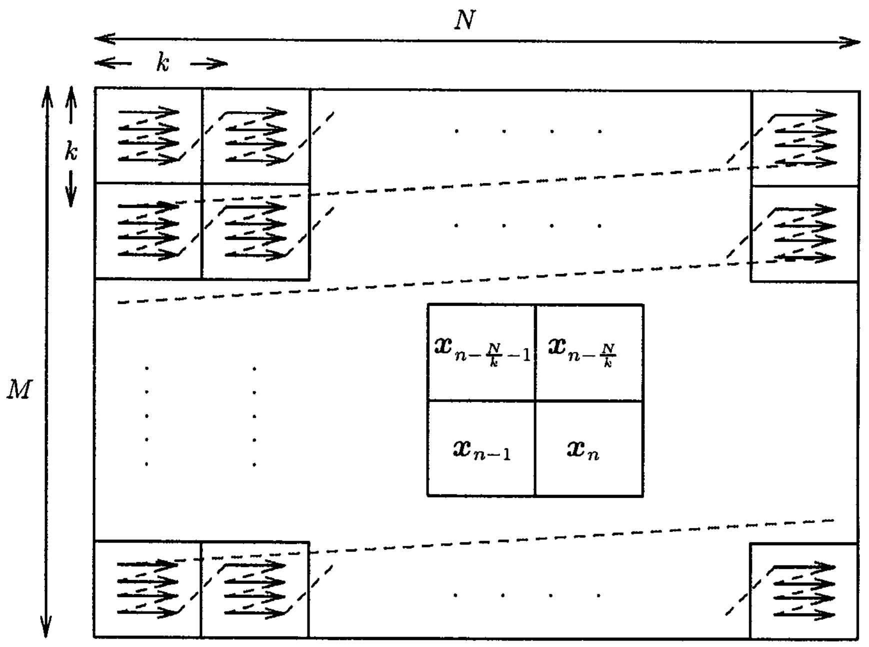


Figure 3.5: Block scan of an input image frame of size $N \times M$.

## Image Input Format

For image compression, we consider an input image of size $N \times M$ pixels such that each pixel is represented as a $b$-bit number. The input image frame is partitioned into small subblocks each of size $k \times k$. Each input frame contains $\frac{N}{k} \times \frac{M}{k}$ subblocks and each subblock can be treated as a vector, $\boldsymbol{x}$, of dimension $L = k^2$. Typically, the size of the subblock is $4 \times 4$ or $8 \times 8$ pixels. A sequence of vectors is formed by raster scanning along the consecutive rows of subblocks. Within each subblock, the pixels are scanned from left to right and top to bottom as shown in Fig. 3.5. This sequence of input subblocks can be treated as a vector-valued random process $\{\boldsymbol{x}_n\}_{n=1}^{NM/k^2}$.

## Linear Predictor Module

For each input vector, $\boldsymbol{x}_n = (x_1, x_2, \cdots, x_L)^T$, the nearest causal neighbors are $\boldsymbol{x}_{n-\frac{N}{k}-1}$, $\boldsymbol{x}_{n-\frac{N}{k}}$ and $\boldsymbol{x}_{n-1}$. Though different forms of linear prediction are possible, we choose the following form[21]. Each pixel $x_{i,j}$, $i$, $j = 1, \cdots, k$, within the vector is predicted using a linear function of past pixels as follows:

$$\tilde{x}_{i,j} = a_{i,j} \cdot y_{k,j} + b_{i,j} \cdot y_{k-1,j} + c_{i,j} \cdot y'_{k-1,k-1} + d_{i,j} \cdot y''_{i,k-1} + e_{i,j} \cdot y''_{i,k-1},$$

where $y_{k-1,j}$ and $y_{k,j}$ are the nearest pixels in the same column from the north subblock $\boldsymbol{x}_{n-\frac{N}{k}}$, $y''_{i,k-1}$ and $y''_{i,k}$ are the nearest pixels in the west subblock $\boldsymbol{x}_{n-1}$ and $y'_{k,k}$ is the lower right corner pixel of the north-west subblock $\boldsymbol{x}_{n-\frac{N}{k}-1}$ as shown in Fig. 3.6. Since $\tilde{x}_{i,j}$ is formed from a linear combination of pixels $y_{k,j}$, $y_{k-1,j}$, the quantization of subblock $\boldsymbol{x}_n$ cannot be started until subblocks $\boldsymbol{x}_{n-\frac{N}{k}-1}$, $\boldsymbol{x}_{n-\frac{N}{k}}$ and $\boldsymbol{x}_{n-1}$ are completely quantized. If the correlation between subblocks $\boldsymbol{x}_n$ and $\boldsymbol{x}_{n-1}$ is ignored, the PTSVQ system can be pipelined efficiently. The decrease in performance due to ignoring the west subblock is small over most rates [21]. Hence, we use a simpler 3-order prediction function which does not depend on pixels $y''_{i,k-1}$ and $y''_{i,k}$ in our PTSVQ architecture. Here, $\tilde{x}_{i,j}$ is defined as:

$$\tilde{x}_{i,j} = a_{i,j} \cdot y_{k,j} + b_{i,j} \cdot y_{k-1,j} + c_{i,j} \cdot y'_{k,k}. \qquad (3.1)$$

The architecture of the 3-order predictor is shown in Fig. 3.7. The three values of the pixels and their corresponding coefficients enter these multipliers simultaneously. Input data is skewed and all internal operations are performed in a bit-skewed word-parallel fashion. The precision of the numbers, $y_{k,j}$, $y_{k-1,j}$, $y'_{k,k}$, is $b$-bits. The coefficients $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$ have a precision of $(b+1)$ bits. All the multipliers and accumulators used in this architecture are pipelined at the

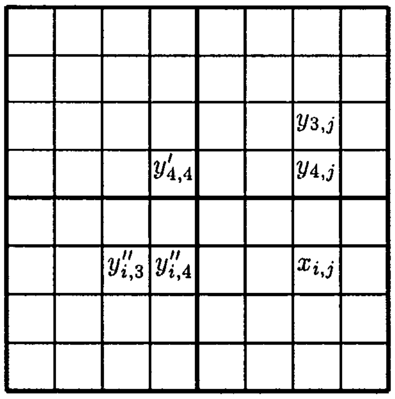


Figure 3.6: Consider the subblock of size $4 \times 4$. Pixel $x_{i,j}$ is predicted as a linear function of nearby pixels in adjacent vectors by $\tilde{x}_{i,j} = a_{i,j} \cdot y_{4,j} + b_{i,j} \cdot y_{3,j} + c_{i,j} \cdot y'_{4,4}$.

bit-level. It takes $b$ clock cycles for the multipliers to generate the first LSB of the product. Since multiplier outputs are already in skewed format, they can be directly fed into bit-level pipelined adders without additional skewing registers. The products $a_{i,j} \cdot y_{k,j}$ and $b_{i,j} \cdot y_{k-1,j}$ are added by the first adder, then the partial sum and $c_{i,j} \cdot y'_{k,k}$ are added by the second adder. To maintain full precision in all internal computations, the second adder takes a $(2b+1)$-bit product and a $(2b+2)$-bit partial sum from the first adder to form a $(2b+3)$-bit sum. At the last stage, skewing registers are used to deskew the predicted value. The total latency time for the linear predictor module is $3b+3$ clock cycles.

The linear predictor needs a memory of $3L$ words to store the coefficients $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$ in equation (3.1).

### Data Buffer and Control Units

From equation (3.1), we see that the inputs to the predictor module, $y_{k,j}$, $y_{k-1,j}$, $y'_{k,k}$, are used repeatedly; both $y_{k,j}$ and $y_{k-1,j}$ are used to compute the different prediction values in the same column and $y'_{k,k}$ is used for every pixel in the

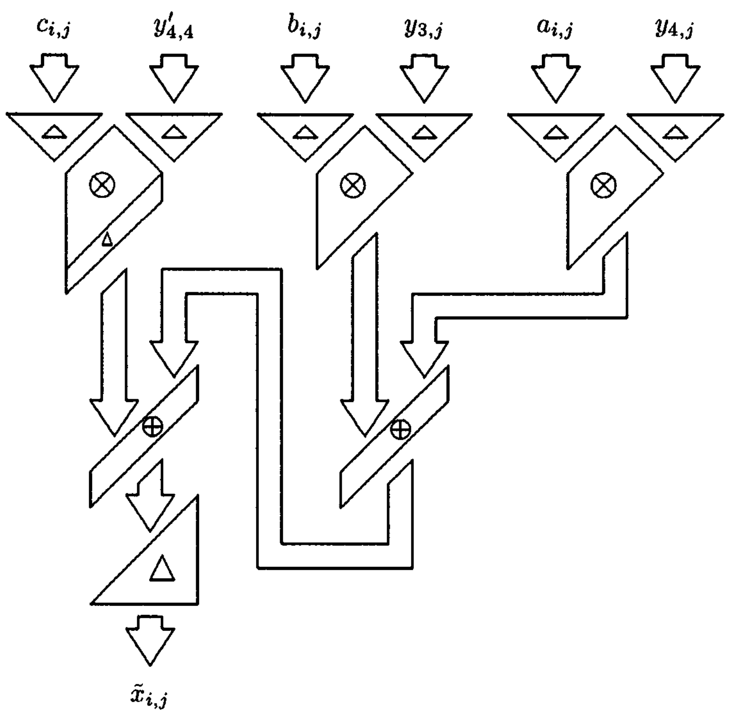


Figure 3.7: Systolic Architecture for third-order linear predictor. It performs the summation of three inner products.

subblock. A simple circuit with cyclic shift registers, shown in Fig. 3.8, is used to handle this task. The last two rows of the image subblock in the input sequence are latched into buffers. Similarly, the last pixel of the previous subblock is latched into a single-word buffer. Control signals are used to ensure that these pixels are applied to the predictor module in the correct sequence.

The control unit consists of a $\lceil 2 \log k \rceil$-bit counter and simple combinational circuitry. The counter keeps track of each input vector to indicate the current pixel position within each incoming vector. Two input control signals are necessary; MODE indicates if the processor is being used in the encoder or the decoder, and Synch indicates if the current input is a boundary subblock.

Three control signals, ctrl1, ctrl2 and ctrl3, are internally generated to switch the multiplexers to update the content of the cyclic buffer. The counter enables us to fetch the appropriate coefficients from external memory into the linear predictor module.

The boundary subblocks in the top row and the left column of the input image frame are treated differently. In the Predictor Processor, there are two switches. Once a boundary subblock is indicated by the Synch input, the predicted value entering the adder/subtractor is set to zero. The corresponding vector will pass through the adder/subtractor without modification. For better performance, these boundary subblocks can be coded using a different codebook from the one used for the residual vectors. However, this results in an increase in the size of the memory required. In our architecture, the boundary conditions of the image and the initialization of the system can be easily handled.

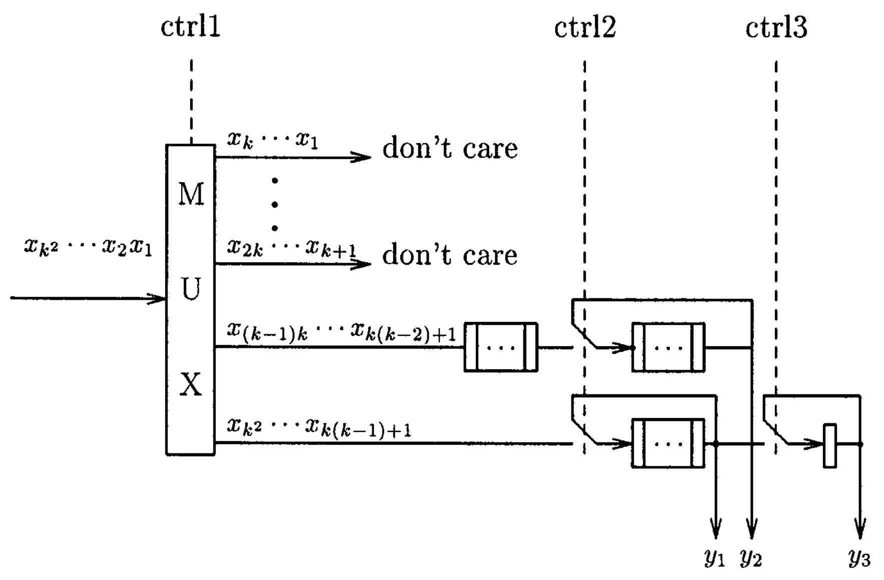


Figure 3.8: Circuit diagram of data buffer unit. It consists of cyclic shift register and multiplexers.

## Timing and delay elements

Due to the presence of the feedback loop in the system, we have to insert delay elements to synchronize all intermediate computations. For simplicity, we introduce a notation and then discuss the details about the delay elements.

Let $L_s$, $L_a$, $L_\gamma$ and $L_\beta$ be the latency times through a subtractor, an adder, a VQ encoder, and a VQ decoder respectively. Let $L_{LP}$ be the latency time of the linear predictor module and the data buffer unit, and let $L_T$ to be the minimum time needed to compute the corresponding predicted value for the subblock in the next row after a block of pixels are fed in. The term $L_T$ includes the processing time along the data path including the subtractor, TSVQ, ITSVQ, the adder, data buffer unit and the linear predictor module, i.e.

$$L_T = L_s + L_\gamma + L_\beta + L_a + L_{LP}.$$

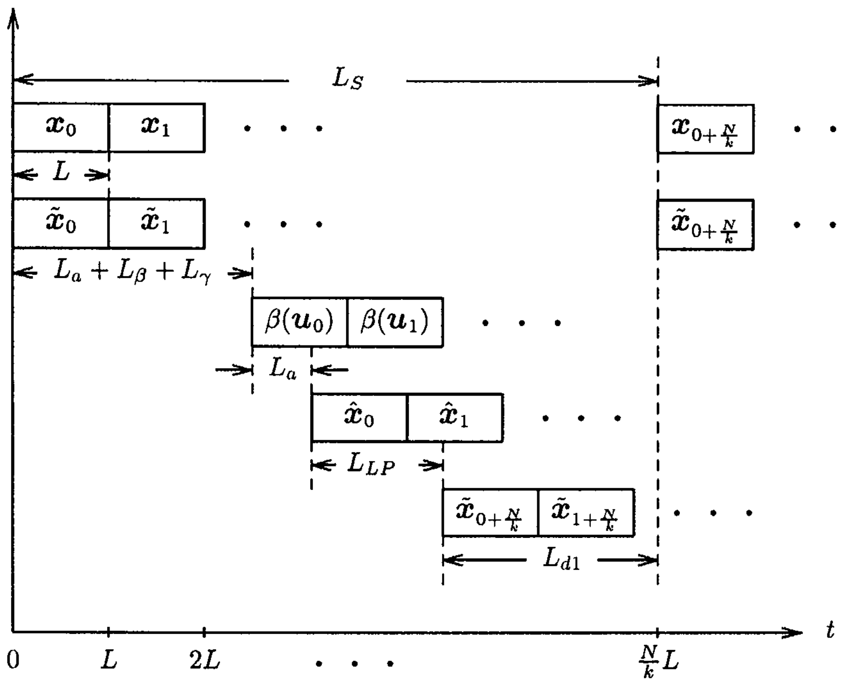


Figure 3.9: Space-time diagram of the PTSVQ system. The relative latency is shown at some interesting points.

Let $L_S$ be the input separation time between two adjacent subblocks in the same column. The space-time diagram shown in Fig. 3.9 depicts the physical meaning of the above terms and the relative timing between them.

We notice, from Fig. 3.9, that as long as the latency time $L_T$ is no larger than the separation time $L_S$, this architecture will achieve real-time performance. In order to synchronize the predicted vector and its corresponding input vector, the delay elements in the feedback loop must satisfy the following timing constraints:

$$L_s + L_\gamma + L_\beta + L_a + L_{LP} + L_{d2} = L_S,$$

and

$$L_a + L_\gamma + L_\beta = L_{d1},$$

where $L_{d1}$ and $L_{d2}$ are the delay times associated with the delay elements in the path.

Consider, for example, the case of an image of size 512 × 512 pixels where each subblock is of size 4 × 4 pixels. Then $L_S$ is $512 \times 4 = 2048$ cycles. In a TSVQ of vector dimension 16 and depth 8, $L_\beta$ and $L_\gamma$ are 1 and 384 cycles respectively; $L_{LP}$ is 42 cycles and includes three pipelined multiplications, two additions and a few data skew elements; $L_a$ is equal to 9 cycles. Hence, $L_{d1}$ and $L_{d2}$ are 394 and 1601 respectively.

## 3.3 VLSI Implementation and Testing

In this section, we describe the VLSI implementation of the chips necessary to build a PTSVQ system using $1.2\mu m$ CMOS technology. We designed these chips using Magic, Irsim, Spice and GDT tools. Spice simulations indicate that these chips can run at frequencies up to 40 MHz. We fabricated prototype version of these chips using $2\mu m$ CMOS N-Well process, and tested these prototype chips at 20 MHz.

### 3.3.1 SNP chips

We designed a Single Node Processor using $1.2\mu m$ CMOS N-Well technology. The processor contains 25,000 transistors on a $4.8mm \times 5.5mm$ die and has 84 pins. We performed logic and timing simulations at 40 MHz on this chip. A prototype version of this chip fabricated using $2\mu m$ CMOS process works at a frequency of 20 MHz [14].

### 3.3.2 Predictor Processor

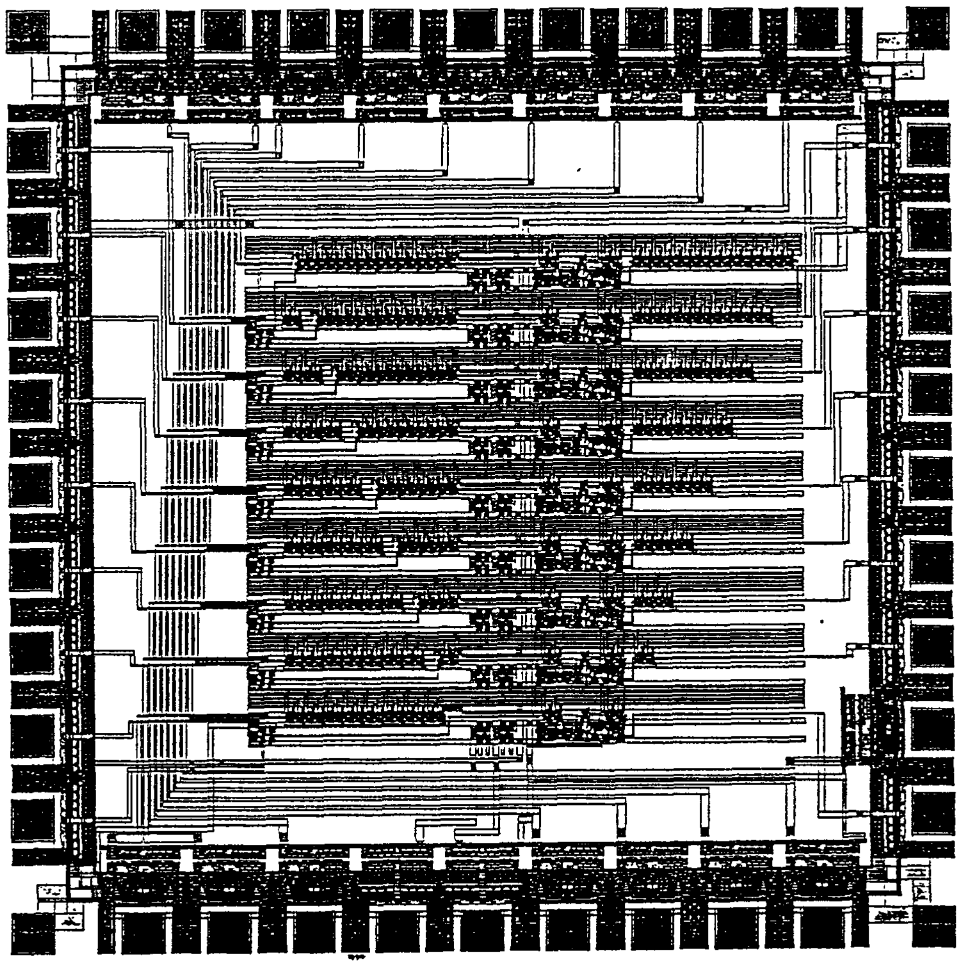


Figure 3.10: Plot of the front end processor of size $2.2mm \times 2.2mm$.

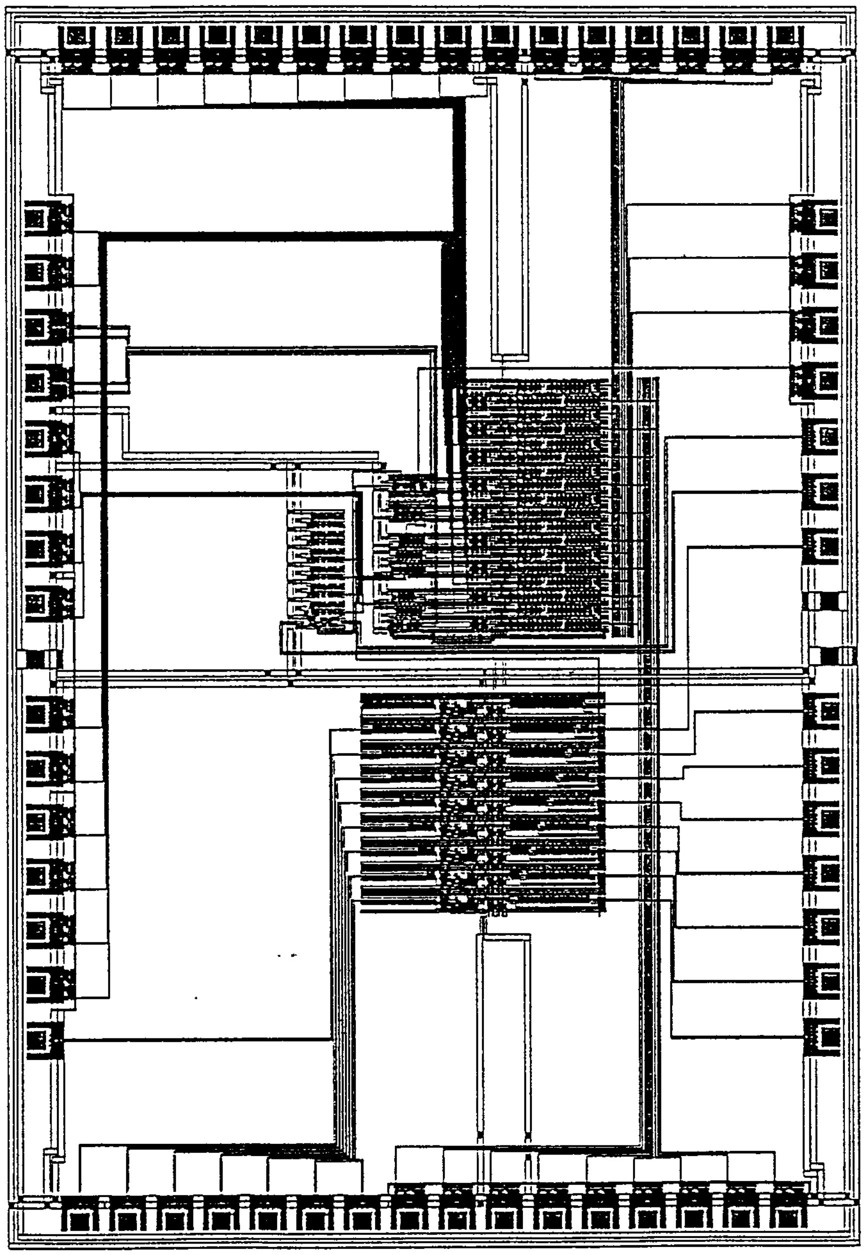


Figure 3.11: Plot of the controller of size $2.8mm \times 4.1mm$.

We partitioned the predictor processor into different submodules for ease of implementation.

1. *Front end processor.* It consists of a pipelined subtractor with skewing and deskewing elements. This module contains 1,650 transistors on a $1.3mm \times 1.3mm$ die using $1.2\mu m$ technology. Figure 3.10 shows a plot of this chip. A prototype version of this chip using $2\mu m$ technology worked at 20 MHz.

2. *Controller and Buffer.* It includes a pipelined adder with skewing and deskewing elements, the control circuit and the buffer unit. This module contains 4,000 transistors on a $2.8mm \times 4.1mm$ chip using $1.2\mu m$ technology. Figure 3.11 shows a plot of this chip. A prototype version of this chip fabricated using $2\mu m$ process worked at 20 MHz.

3. *Predictor.* It consists of the Linear Predictor Module (LPM). This module contains 36,000 transistors on a $4.7mm \times 5.5mm$ die. Figure 3.12 shows a plot of this chip. A prototype version of this chip fabricated using $2\mu m$ process worked at 20 MHz.

## 3.4 Discussion

A new Predictive TSVQ architecture is presented for real-time video coding applications. Pipelined arithmetic components are used to speed up the computation and to provide for regularity in design. This high throughput architecture is suitable for implementing a fully pipelined real-time PTSVQ system. This architecture has been implemented as a VLSI chip set using $1.2\mu m$ CMOS technology. Identical processors are used for both the encoding and decoding components.

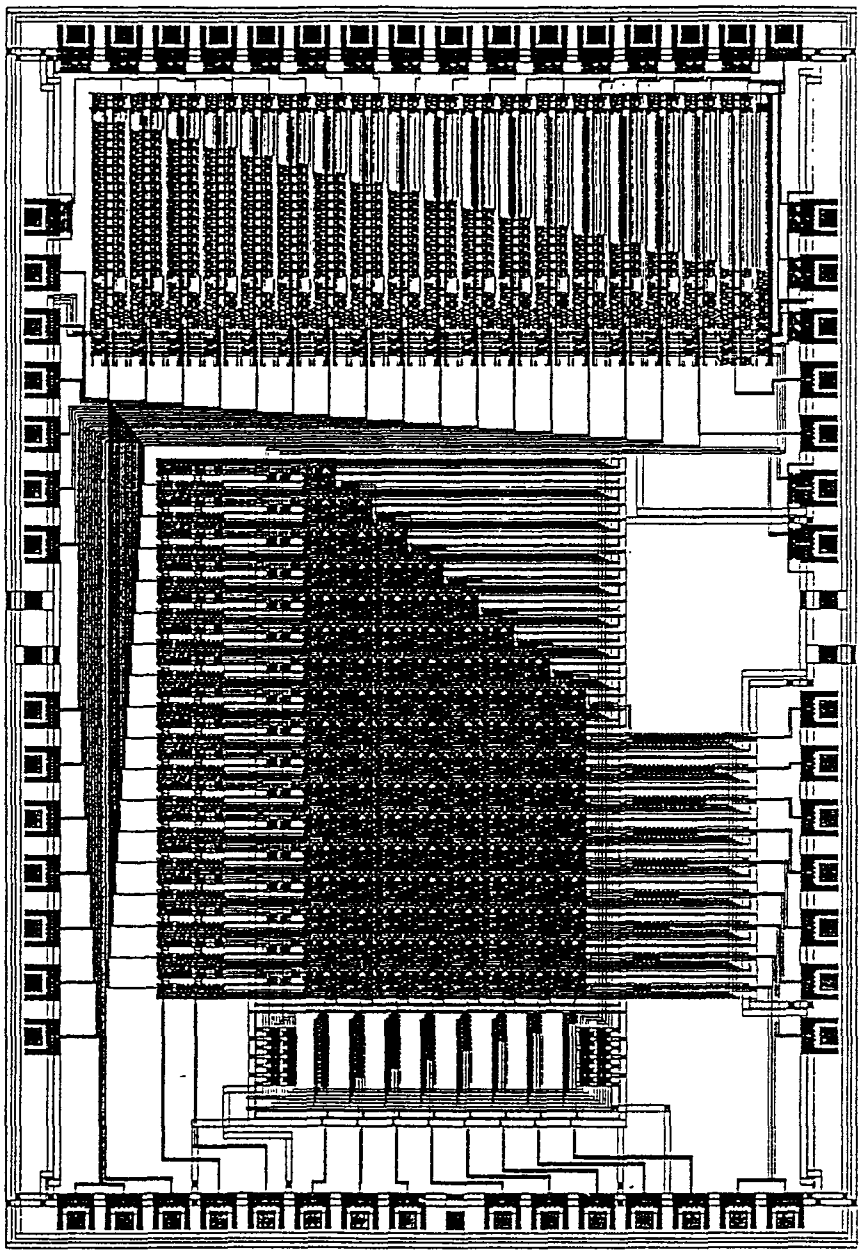


Figure 3.12: Plot of the predictor of size 7.9$mm$ $\times$ 9.2$mm$

Spice simulations indicate correct operation at 40 MHz. For a typical real-time image processing system with 30 frames/sec and $1024 \times 1024$ pixels/frame, the input pixel rate is 31.5 Mpixels/sec. This architecture is capable of processing 40Mpixels/sec and can handle the above case in real-time. We fabricated prototype versions of these chips using $2\mu m$ CMOS technology. These prototype chips work at 20 MHz. Our architecture can be extended easily to handle other classes of VQ with memory such as Trellis VQ.

# Chapter 4

# Higher Order Crossings

## 4.1 Introduction

Time series are sequences of observations or measurements ordered in time. Time series occur everywhere and anytime, in the natural sciences, engineering and even in the social sciences. Time series analysis provides an important tool for a wide variety of applications, such as tidal fluctuation, traffic flow, or electrical noise, etc. Experience shows that almost all observed time series are oscillatory, displaying the up and down property either locally or globally. Based on this basic observation, a considerable amount of work has been established [33, 9].

The information contained in the oscillation of a time series can be extracted and represented by the sequence of zero crossings. To acquire a more useful information, we can use filters, especially linear filtering, to isolate specific parameters. We follow the approach described in [9]. Typically, a family of filters is applied to a time series to generate sequences of the zero-crossing counts. It is surprising that such a sequence can provide extensive information for signal discrimination, classification, and for frequency estimation in the presence of noise[9]. We refer to such a sequence of zero crossings count as a HOC ( Higher Order Crossings ) sequence. Because HOC have an advantage due to their simplicity, data reduction and compression capabilities, they provide a useful tool that for certain applications they can be more effective than the conventional methods based on spectral analysis.

By applying different types of filters, we can have have different HOC sequences. We will examine two filtering techniques that have been successfully used in several applications including non-destructive evaluation and signal discrimination. The first is the differential filtering, and the second is the $\alpha$-filtering (AR(1) filter) to be introduced later. There are still many other useful filters that can be designed and investigated for different types of applications, but we restrict ourselves to these two filters in this chapter.

Discrimination or classification is one aspect of HOC that has been quite successful. One simple example is to decide the frequencies from the superposition of two sinusoids. With proper use of the differential filtering, the frequencies can be detected in a simple and an accurate fashion. In addition, signal classification between noise and signal can be achieved using a similar process. Other specific applications of HOC include the the ultrasonic classification of adhesive joints. HOC analysis has been shown to be quite successful in the Non-destructive evaluation (NDE) analysis for characterizing adhesive joints though commercial bond testers exists under certain restrictions [12, 34, 35, 11]. Moreover, further analysis of the HOC sequence using the similarity test, that is based on a new measurement called $\psi^2$ statistic, can quantify the resulting HOC sequence[36, 37]. It gives the discrepancy between a set of HOC parameters drawn from an acquired test signatures and that from a reliable one.

Frequency estimation in the presence of noise has been one of the well-studied problems in both the engineering and the scientific literature[38]. The novel iterative method for frequency estimation, introduced in [39], is based on HOC with sophisticated $\alpha$-filtering, and is referred as the contracting mapping (CM) method. This novel method provides a mechanism that yields very precise frequency estimates at very low computational costs compared to the conventional methods, such as the Fourier transform, the periodogram analysis[40, 41, 42], the non-linear least squares method[41, 43, 44], etc.

Though a simple schematic of the HOC processor was mentioned in [24], due to the wide applicability of HOC analysis in different areas, we designed a flexible special-purposed architecture that can adapt different filter operations for different applications. In this chapter, we present VLSI architectures for performing HOC analysis. The system consists of 1) a preprocessor, 2) a $\alpha$-filter, 3) a HOC processor, and 4) a postprocessor. The architecture of two important filters are discussed in some detail. The proposed architectures have the following features: 1) they are simple and modular, 2) they can operate in real time, 3) they are flexible in the sense that they can be adapted to handle different HOC analysis techniques, and 4) they have simple I/O requirements.

This chapter is organized as follows. In the following section, we describe the basic definitions and the corresponding algorithms, and in Section 3, we describe the architectures for a system to perform HOC analysis. In Section 4, an generic system to performing HOC analysis and the corresponding implementation issues are described in Section 5.

## 4.2 Definitions

Let X(t) be a continuous-time random signal. A sampled series x(i), $i = 1, \cdots, N$ is the digitized sequence of X(t). The mean is computed and removed from x(i) to form a zero mean sequence $\tilde{x}(i)$. The number of zero crossing, denoted by $D_1$, is detected and counted. A family of different filters can be further applied to $\tilde{x}(i)$. A family of time series is generated which yields a corresponding zero crossing count determined by the filters used. The resulting sequence of zero crossing counts corresponding to $\tilde{x}(i)$ and its filtered sequences is referred to as the higher order crossings (HOC) sequence [9, 12].

Of these filters, the difference ( differential ) filter and the $\alpha$-filter are two of the important filters used in many applications. Next, we define the difference operator, followed by a definition of the $\alpha$-filter.

A first-order Backward Difference Operator ( BDO ) is defined by

$$\nabla[x_j] = x_j - x_{j-1}. \tag{4.1}$$

A second application of the difference operator yields

$$\begin{aligned} \nabla^2[x_j] &= \nabla[\nabla[x_j]] = \nabla[x_j - x_{j-1}] \\ &= x_j - 2x_{j-1} + x_{j-2}. \end{aligned}$$

And the $k-$th application of the difference operator can be expressed as

$$\nabla^k[x_j] = \nabla[\nabla^{k-1}[x_j]]. \tag{4.2}$$

It can also be shown that

$$\nabla^k[x_j] = \sum_{i=0}^{k} C_i^k (-1)^i x_{j-i} \tag{4.3}$$

where $C_i^k = \frac{k!}{i!(k-i)}$.

We define $D_i^d$ as the number of zero crossing of the series $\nabla^i[\tilde{x}_j]$ for $j = 1, 2, \cdots, N$.

The $\alpha$-filter of order $p$ for a sequence is defined as

$$\mathcal{L}_\alpha^p(x)_t = x_t + \alpha x_{t-1} + \alpha^2 x_{t-2} + \cdots + \alpha^p x_{t-p}, \quad \alpha \in R. \tag{4.4}$$

We refer to the corresponding family of higher order crossings $\{D_i^{\alpha,p}\}$, $\alpha \in R$, as the HOC sequence from the $\alpha$-filter of order $p$. Under some conditions, the sequences $\{E(D_i^d)\}_{i=1}^{\infty}$ and $\{E(D_i^{\alpha,p})\}_{i=1}^{\infty}$ determine the spectrum up to a constant[9]. Consider the case $p = 1$ and $\alpha = -1$, the $\alpha$-filter becomes the difference operator as described in equation (4.1), i.e. $\mathcal{L}_{-1}^1 = \nabla$.

The terms classification or discrimination between two signals mean to determine from HOC sequences the degree of similarity between two time series, or between a given time series and a hypothesized one, or between different sections from the same signal. For the sake of clarity, we simply use $\{D_k\}$ which omits the superscript in the HOC sequences corresponding to different filtering operations. Consider the observed HOC sequence $\{D_k\}$. A general form for similarity or distance measures can expressed as

$$(\mathbf{D} - E[\mathbf{D}])^T \mathbf{C}^{-1} (\mathbf{D} - E[\mathbf{D}]) \tag{4.5}$$

where $\mathbf{D} = (D_1, D_2, \cdots, D_K)$, and $\mathbf{C}$ is a $K \times K$ weight matrix. It is known for reasonably large time series the simple HOC tends to increase monotonically with a high probability,

$$D_1 < D_2 < \cdots < D_k \cdots.$$

A useful statistic that simply quantifies the similarity between two different processes is the so called $\psi^2$ statistic which is defined next. For the given HOC sequences $\{D_k\}$, we define the increments as

$$\begin{aligned}\Delta_k &= D_1, \quad k=1 && (4.6)\\ &= D_k - D_{k-1}, \quad k=2,\cdots,K-1 && (4.7)\\ &= (N-1) - D_{K-1}, \quad k=K && (4.8)\end{aligned}$$

where $\sum_{k=1}^{k=K} \Delta_k = N-1$. When N is large enough (eg. $N \geq 200$), then with a high probability $0 \leq \Delta_k \leq N-1$, and $\sum_{k=1}^{k=K} \Delta_k = N-1$. Let $m_k = E[\Delta_k]$. A general similarity measurement is defined as follows:

$$\psi^2 = \sum_{k=1}^{k=K} \frac{(\Delta_k - m_k)^2}{m_k}, \tag{4.9}$$

where the parameter $K$ refers to the number of "classes" or "categories". The $\psi^2$ statistic can be used as a distance measure from a process with prescribed parameters $m_k$'s. Experience indicates that very few $D_j$'s are needed for successful discrimination.

Assume that we are given an input sequence x(i) for $i = 1, 2, \cdots, N$ with a mean $\theta$. To summarize, we have the following simplified algorithm describing the procedure to compute the $D_i$'s from the input sequence and its corresponding $\psi^2$ statistic from the predefined process with prescribed parameters $m_k$'s.

1. Compute the mean of the sequence as follows:

$$\theta = \frac{1}{N}\sum_{i=1}^{N} x(i). \tag{4.10}$$

2. Form the new sequence, $\tilde{x}$, with mean removed from each of the input sample. Hence,

$$\tilde{x}(i) = x(i) - \theta \tag{4.11}$$

where $i = 1, 2, \cdots, N$.

3. Compute the corresponding HOC sequence, $D_1, D_2, \cdots, D_K$, for the zero-meaned sequence $\tilde{x}_j$.
4. Compute the vector of increments, $\Delta_k$, for $k = 1, 2, \cdots, K$.
5. Compute the corresponding $\psi^2$ statistic.

We observe that the input sequence can be independently processed before the original HOC is computed. Hence, we will design efficient architectures for either filtered or unfiltered input sequence based on HOC analysis.

## 4.3 Architectures

In this section, we give the overall architecture for the entire system including the removal of the mean, the $\alpha$-filtering operations, the computation of the zero-crossing, and the evaluation of the $\psi^2$ statistic. We first present an architecture of the preprocessor and show how it is used to compute the mean and then remove it from the input sequence. Then, we describe systolic architectures for the $\alpha$-filtering in detail. Next, we show how a linear systolic architecture can be used to compute the HOC sequence. Finally, an architecture designed to compute the $\psi^2$ statistic from the previously computed zero-crossings, $\{D_k\}$ is described.

### 4.3.1 System overview

The overall architecture consists of four functional blocks, as was shown in Fig 4.1. The four major functional blocks consist of (1) the preprocessor, (2) the $\alpha$-filter, (3) the HOC Processor, and (4) the Postprocessor. Each of these

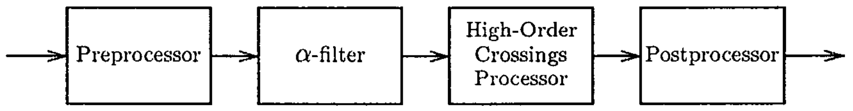


Figure 4.1: HOC overall architecture

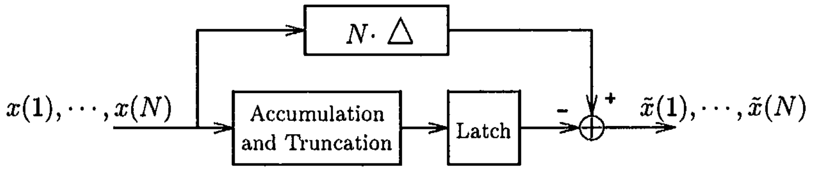

$N\cdot\Delta$
$x(1),\cdots,x(N)$
Accumulation and Truncation
Latch
-
+
$\tilde{x}(1),\cdots,\tilde{x}(N)$

Figure 4.2: Block diagram of the Preprocessor that computes and removes the mean from a sequence of $N$ inputs.

blocks are locally connected so that no global control is required. For detailed operations of their blocks, we describe each of them in the following sections.

### 4.3.2 Preprocessor

A preprocessor is designed to perform the computation in equations (4.10) and (4.11), i.e. compute the mean of the input sequence and subtract the mean from each of the inputs. The length of the input sequence is restricted to be of the form $N = 2^p$. The division can be implemented by right-shifting the $p$ least significant bits (LSB) without compromising the accuracy. Hence, internal precision is well maintained and the circuit complexity is reduced. The input is buffered in an First-In-First-Out buffer until the mean is generated. Then, the subtraction is applied to each buffered input sequence as stated in equation (4.11). This yields an zero-mean sequence to apply to the next Higher-Order Crossing Processor. The architecture is shown in Fig. 4.2.

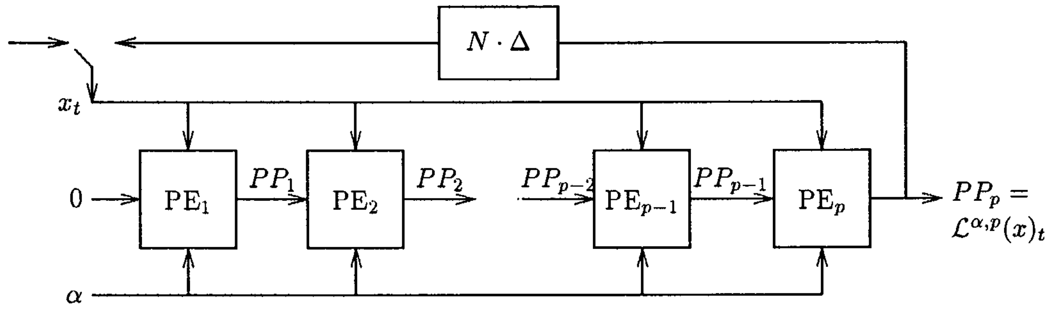


Figure 4.3: Block diagram of the TYPE I realization.

## 4.3.3 The $\alpha$-filter

The $\alpha$-filter defined in equation (4.4) can be implemented directly; however, the corresponding hardware cost is unnecessarily high. We propose two types of architectures for implementing the $\alpha$-filter resulting in simple and regular circuits.

### Type I

Note that the direct implementation of equation (4.4) requires $p$ different coefficient and input values. From equation (4.4), one can obtain

$$\begin{aligned}
\mathcal{L}^p_\alpha(x)_t &= x_t + \alpha x_{t-1} + \alpha^2 x_{t-2} + \cdots + \alpha^p x_{t-p} \\
&= \underbrace{x_t + \alpha(\underbrace{x_{t-1} + \alpha(\underbrace{x_{t-2} + \cdots + \alpha(\underbrace{x_{t-p-2} + \alpha(\underbrace{x_{t-p-1} + \alpha x_{t-p}}_{PP_1}))}_{PP_2}\cdots))}_{\vdots}}_{PP_p} .
\end{aligned}$$

Therefore, each $PP_i$ term can be expressed as follows:

$$PP_i = \alpha \cdot PP_{i-1} + x_i .$$

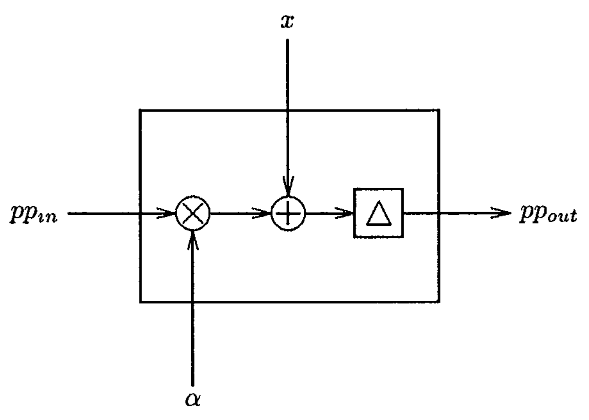


Figure 4.4: Block diagram of a single processor for TYPE I realization.

The above algorithm can be easily mapped into a linear array of systolic processors as shown in Fig. 4.3. Each processor PE$_i$, as shown in Fig. 4.4 performs only multiply/add operations, as indicated by $PP_i = \alpha \cdot PP_{i-1} + x_i$. The input $x_t$ is fed into each processor simultaneously. There is feedback path where the filtering operation can be applied to the inputs repeatedly to generate the family of HOC sequences. Hence, the I/O requirement is greatly simplified so that only one coefficient $\alpha$ and the current input $x_t$ is sufficient to compute the output.

## Type II

A simple implementation can be derived by the following relationship:

$$\begin{aligned} \mathcal{L}^p_\alpha(x)_t &= x_t + \alpha x_{t-1} + \alpha^2 x_{t-2} + \cdots + \alpha^p x_{t-p} \\ &= x_t + \alpha(x_{t-1} + \alpha x_{t-2} + \cdots + \alpha^{p-1} x_{t-p} + \alpha^p x_{t-p-1}) - \alpha^{p+1} x_{t-p-1} \\ &= x_t + \alpha \mathcal{L}^p_\alpha(x)_{t-1} - \alpha^{p+1} x_{t-p-1} \qquad (4.12) \end{aligned}$$

We can map and realize the above recurrence equation into the following architecture as shown in Fig. 4.5. Similarly, the feedback path in the realizations

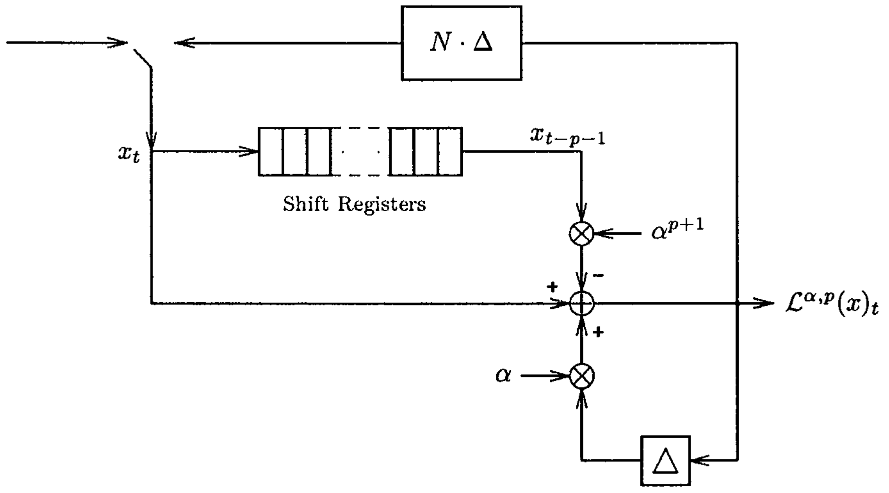


Figure 4.5: Block diagram of the Type II realization.

is to buffer the filtered outputs so that the filtering operation can be applied to the sequence repeatedly to generate the family of HOC sequences.

### Note

As we can see that when the order p tends to be infinitive, the last term in equation (4.12) vanishes. As the I/O requirement is considered, the type I realization needs only one coefficient and the data input, while it needs $p$ multiplications totally. For the type II realization, the number of multiplications can be greatly reduced to 2, while it needs two coefficients and the data input.

## 4.3.4 Zero Crossing Processor

The processor that detects the sign difference between two consecutive inputs and triggers the counter to accumulate the total number of zero-crossing from the input sequence. The sign detection mechanism is built using an exclusive-or gate and a delay element to compare the MSB bits between the consecutive

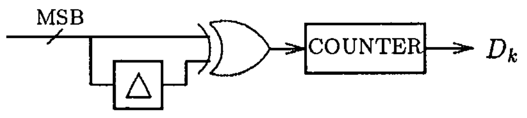


Figure 4.6: Scheme of Zero Crossing Processor.

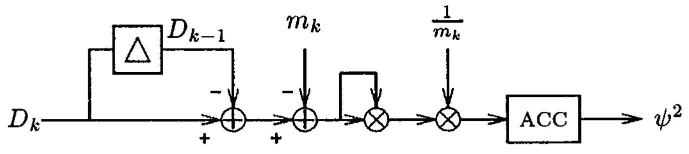


Figure 4.7: Block diagram of the Postprocessor which performs the $\psi^2$ statistic as described in equation (4.9).

numbers from the filter outputs. The sign discrepancy could trigger the counter to record the number of zero crossings correctly.

### 4.3.5 Postprocessor

A Postprocessor is designed to perform the computation in equation (4.9), i.e. compute the $\psi^2$ statistic. The inputs $D_1, D_2, \cdots, D_8$, are the corresponding zero-crossing counts after performing the filtering operations on the input sequence repeatedly from last stage. The proposed architecture, as shown in Fig. 4.7, can perform the computation described in equation (4.9) in a systolic fashion. The quantities $\frac{1}{m_k}$ can be precomputed and no division is required.

## 4.4 Design Methodology

We present an overall system architecture for HOC analysis based on the basic building blocks examined earlier. In general, there are two approaches for implementing the proposed system. The first uses a high-performance general purpose signal processor, and the second uses ASICs. Compared to ASICs, the major advantage of using a general purpose DSP is that it is programmable and hence quite flexible. On the other hand, the ASIC implementation would be the most efficient approach because it is dedicated to one specific application. We believe that the HOC analysis is a powerful tool that can be used in many applications. Hence, we propose a suitable architecture with off-the-shelf chips with the aim of achieving the dual goals of high performance and maximum flexibility.

The previously defined blocks can be mapped onto a generic system as shown in Fig. 4.8, which is referred to as the HOC-$\psi^2$ system. The HOC-$\psi^2$ system is not only capable of executing the function of each of the above architectures, but also provides the computational capability to perform many variations of HOC analysis.

### 4.4.1 Scheme for data computation and transfer

Conceptually, the operations can be modeled as data movement between storage elements where the appropriate data processing is done in between such transfer. The simplest example is the data transfer between two registers where none of any computation operation is done here. A complete example is the multiplication of two numbers, where the two input registers release their data to the input ports of the multiplier, then the multiplication operation is executed, and finally the result is latched into its internal storage element.

Based on the above model, the controller can easily generate all the timing and control signals for the entire system. A detailed timing diagram is shown in Fig. 4.9. Whenever such a transfer is initiated, the controller asserts an active

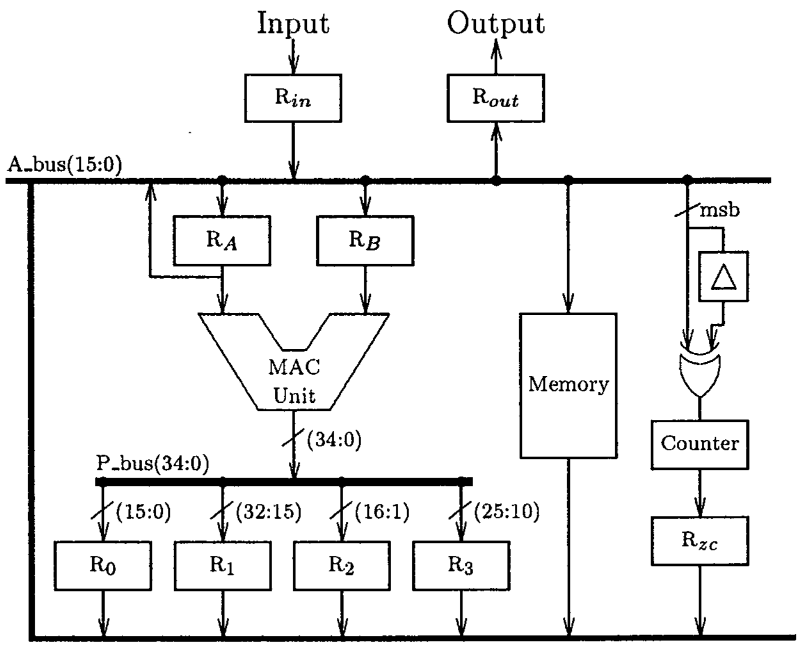


Figure 4.8: Block diagram of the Generic Processor.

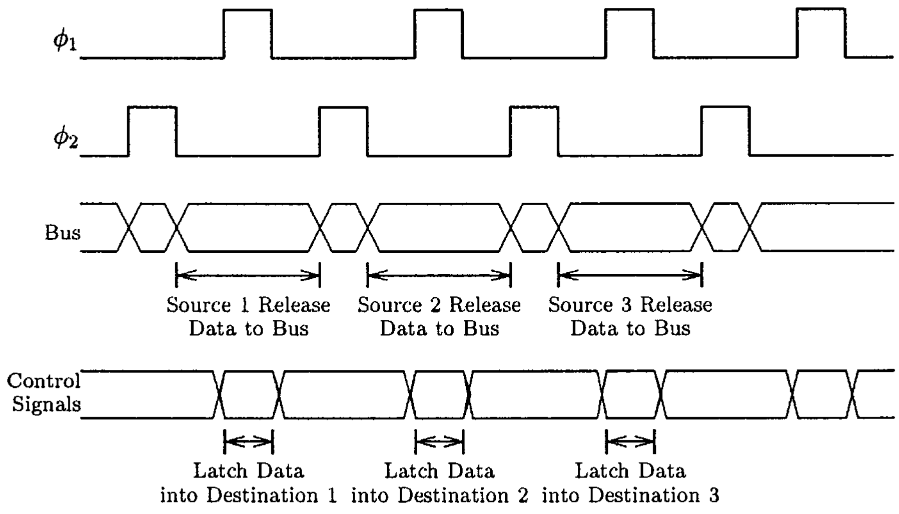

$\phi_1$
$\phi_2$
Bus
Source 1 Release Data to Bus
Source 2 Release Data to Bus
Source 3 Release Data to Bus
Control Signals
Latch Data into Destination 1
Latch Data into Destination 2
Latch Data into Destination 3

Figure 4.9: Timing diagram.

signal to release data from the source device during the consecutive $\phi_2$ phases, so that the data is assured valid for computation elements during this period of time. After the results coming out from the computation elements become valid, the controller then asserts the signal, $\phi_1$, to activate the storage elements to latch the valid data. For instance, when executing the multiplication of two numbers held in registers A and B, the registers are acting as the source devices and the internal multiplier-accumulator is acting as the destination storage device. While storing the product from accumulator into the register, the internal accumulator is acting as source device and the register is acting as a destination device. In such a manner, the transfer can be synchronized and flow smoothly to enhance the throughput of the system.

### 4.4.2 Data Path Architecture

Most of the data transfer and operations occur via the 16-bit internal buses, the A_bus(15:0) and B_bus(15:0), as shown in Fig. 4.8. Another 35 bit data path, P_bus(34:0), is designed on which the computed results from the multiplier-accumulate (MAC) unit can be appropriately loaded into register 0-3 and then transferred into its destination.

## 4.5 Design and Implementation

The HOC-$\psi^2$ system is composed of one multiplier-accumulator, the FIFO storage components, a set of registers, and a counter. In addition, a finite state machine controller is designed to handle the timing and data flow in the system. The functions of these elements are described as follows:

### 4.5.1 Components

#### Multiplier and Accumulator

The computation core of the whole system consists of a $16 \times 16$-bit parallel multiplier-accumulator. The two multiplier inputs are fed into two ports via two internal buses from selected data sources. Because the bottleneck of the whole processing depends on the multiplier, the multiplier and accumulator must be performed at each cycle without idling. The chip we used provides attractive features, including high-speed, low-power, $16 \times 16$-bit multiplication, 35-bit internal precision, and also easy to interface. Either a multiply or a multiply-accumulate function can be performed as specified so that the content of the accumulator can be added or subtracted from the subsequent result.

#### Storage Elements

There are mainly two types of storage elements, FIFO and registers. As to the computation schemes used in the HOC, the data is used in the first-come-first-serve manner. In each step of processing, the consecutive data is processed in sequential order. Though random access memory provides more flexibility, it is not a necessary feature in our applications. The FIFO is sufficient to satisfy the function to store the input and intermediate data, and hence the address decoding mechanism to fetch the memory is simplified. The length of FIFO is determined by the maximum length of the input data sequence.

Two types of registers are used in this design. One is the simple register where data transfer is made possible between components. The other is pipelined register where the date can be organized effectively so that different points of input data can be tapped for use in the filter design.

The internal registers are designed to store intermediate results. Some registers are used either as a temporary buffer or in specific application to save in the hardware complexity. For example, $R_0$, $R_1$, and $R_3$ can truncate 11, 1, 6 Least Significant Bits (LSB) to be used as divider or preserve the precision. $R_4$ is used as sign recorder to keep track of the sign bit of the data sequence. Thus the counter will count up properly. $R_{zc}$ mainly stores the zero-crossing count from the data sequence. $R_A$ and $R_B$ can be used either for multiplication or for initial values. $R_{in}$ and $R_{out}$ are used as I/O buffers.

#### Counter

The zero-crossing detection mechanism is formed by tapping the Most Significant Bit (MSB) of the stream of data to the counter. The transition between zero and one, *i.e.* sign change, of the consecutive data will trigger the counter to count up. Hence, the effective number of zero-crossings is kept in the counter.

### 4.5.2 Finite State Machine Controller

The Controller mainly consists of a finite state machine which can produce the control signals to activate or deactivate each of the devices at the appropriate time. The control signals, can be categorized into four groups. They include the signals which are used to monitor the operation of the MAC unit, read/write FIFO elements, latch/output registers, and device reset respectively. The controller can easily be set to map the algorithm into the hardware. The detailed timing is appropriately handled in such a way that all data transfers between components are assured to be performed correctly.

The controller can be implemented using either a microprocessor with flexible

I/O port structure such as Intel 87C51 or with a Field Programmable Gate Array (FPGA). In our implementation, the 87C51 plus a simple decoder and a circuit of latches have been used to generate the 0 signals to meet the timing requirement. In addition, the flexibility in programming the microprocessor provides the extra dimension to adapt HOC to other applications.

### 4.5.3 I/O interface

In performing the HOC analysis, the I/O requirements are modest. An RS-232 interface is sufficient to communicate with the external world. The number of I/O pins is greatly reduced to a minimum. The serial communication capability is a built-in function in the 87C51 microprocessor which makes that processor more attractive for our requirement. With the built-in serial input and output mechanisms, the serial-to-parallel or parallel-to-serial conversion is handled automatically inside the microprocessor.

### 4.5.4 Control Sequence

In the following, we state the control sequences at the register-transfer level in order to explain all the operations. We will present the program of the control sequence and show how to map the HOC algorithms into the HOC-$\psi^2$ system efficiently. Also, the $\alpha$-filter is implemented on the above architecture.

#### System initialization and computing mean

As described in Section 3.2, the first part of the operations is to compute the mean of the input sequence and remove the mean from each input. Before any further processing, the system is first initialized with the instructions as shown

| Step | Operation |
|---|---|
| 1 | counter $\leftarrow$ reset |
| 2 | FIFO $\leftarrow$ reset |
| 3 | $\text{R}_{inB}$ $\leftarrow$ '1' |
| 4 | $\text{R}_{inA}$ $\leftarrow$ '0' |
| 5 | ACC $\leftarrow$ $\text{R}_{inA} \times \text{R}_{inB}$ |

Table 4.1: Instructions to initialize the system.

in Table 4.1.

After the system is initialized, we start to compute the mean of the input sequence where the input $x_i$, $i = 1, 2, \cdots, 2048$, is sequentially fed into $\text{R}_{inA}$. For each input $x_i$, it is buffered into $\text{FIFO}_A$, also $x_i$, multiplied by 1, is added into ACC. The Steps 1 to 3 are iterated as many as 2048 times. Then, the sum is computed in the accumulator. To reduce the hardware complexity in our application, the division of the sum is simply executed by right-shifting the 11 LSB, which are held in the register $\text{R}_0$. Hence, when the data transfer from ACC into $\text{R}_0$ causes the division of the sum in ACC by 2048, *i.e.* the mean of the whole input sequence, to be computed. The corresponding instructions are shown in Table 4.2.

### Removing the mean

To subtract the mean from each of the inputs, the program listed in Table 4.3 is sufficient. The basic idea is to fetch the mean, with dummy multiplication by 1, into ACC, then perform the multiplication-and-subtraction on the $x_i$ which

| Step | Operation |
|---|---|
| 1 | $\mathrm{R}_{inA} \leftarrow$ input $x_i$ |
| 2 | $\mathrm{ACC} \leftarrow \mathrm{R}_{inA} \times \mathrm{R}_{inB}$ |
| 3 | $\mathrm{FIFO} \leftarrow \mathrm{R}_{inA}$ |
| 4 | $\mathrm{R}_0(\mathrm{mean}) \leftarrow \mathrm{ACC}(11{:}26)$ |

Table 4.2: Instruction to compute the mean from the input sequence.

is fetched from the FIFO. The result is stored back into the FIFO for later processing. Simultaneously, the sign bit is latched into $\mathrm{R}_4$ to trigger the counter. Steps 1 to 4 are repeated 2048 times until the mean is removed from each of the inputs and the number of zero crossings is counted. Finally, the zero-crossing count is stored for later use as shown in steps 5 and 6. Hence, the FIFO is used as a ring of data buffer where the data is maintained in the right order.

### $\alpha$-filter

We demonstrate how to implement the difference operator which is the special case of the $\alpha$-filter, $\alpha = -1$. To perform the equation (4.1), subtraction is performed for each of the $x_i$ as shown in step 3 where the $x_{i-1}$ is buffered in the temporary register $\mathrm{R}_A$. Next, the result is stored into the FIFO for the next filtering operation and its sign-bit is used to trigger the counter. The instruction is listed in Table 4.4. The general case of the $\alpha$-filter can be extended from the above example with the use of the pipeline register to tap any of the previous data.

| Step | Operation |
|---|---|
| 1 | $ACC \leftarrow R_0(=\text{mean}) \times R_{inB}(=1)$ |
| 2 | $ACC \leftarrow FIFO(x_i) \times R_{inB}(=1)$ - $ACC(=\text{mean})$ |
| 3 | $R_1 \leftarrow ACC$ |
| 4 | $FIFO \leftarrow R_1$<br>$R_4 \leftarrow R_1$ |
| 5 | $R_{ZC} \leftarrow$ Counter |
| 6 | $FIFO_B \leftarrow R_{ZC}$ |

Table 4.3: Instructions to perform the mean removal from each of the input sequence.

| Step | Operation |
|---|---|
| 1 | $R_A \leftarrow R_{inA}(=0)$ |
| 2 | $ACC \leftarrow R_A(=x_{i-1}) \times R_{inB}\ (=1)$ |
| 3 | $R_A \leftarrow FIFO(=x_i)$<br>$ACC \leftarrow FIFO \times R_{inB}(=1)$ - $ACC(=x_{i-1})$ |
| 4 | $R_1 \leftarrow ACC(1{:}16)$ |
| 5 | $FIFO \leftarrow R_1$<br>$R_4 \leftarrow R_1$ |
| 7 | $R_{ZC} \leftarrow$ Counter |
| 8 | $FIFO_B \leftarrow R_{ZC}$ |

Table 4.4: Instructions to perform the filter operation on the sequence of data.

| Step | Operation |
|---|---|
| 1 | $R_A \leftarrow R_{inA}$ |
| 2 | $ACC, R_A \leftarrow FIFO \times R_{inB} - ACC$ |
| 3 | $R_2 \leftarrow ACC$ |
| 4 | $FIFO \leftarrow R_2$ |

Table 4.5: Instructions to compute $\Delta_i$

**Computing the $\psi^2$ statistic**

Up to now, we have collect the important values $D_i$'s. We use the above architecture to compute the $\psi^2$ statistic for signal discrimination purposes as expressed in equation (4.9). In table 4.5, the $\Delta_i$ is computed according to equations (4.7)-(4.8). Finally, the $\psi^2$ statistic is computed as described in Table 4.6.

### 4.5.5 Simulation and Board Layout

We have used the PCB layout tools provided by Mentor Graphics to design and simulate our board. A top-down design methodology was adopted for our design. Each of the components not existing in the component library can be treated as black boxes and further described in high level description language. The behavior model of the components in the system was described using VHDL. There are well-defined elements from the component library as well as user-defined function boxes. The timing can be specified and verified with the data sheets. Then extensive simulation of these modules was done separately. The HOC-$\psi^2$ system is further assembled and simulated using QuickSim tool. The schematic is shown in Figure 4.10 and the corresponding board layout is shown

| Step | Operation |
|---|---|
| 1 | $\mathrm{R}_A \leftarrow$ input $m_k$ |
| 2 | $\mathrm{ACC} \leftarrow \mathrm{R}_{inA} \times \mathrm{R}_{inB}$ |
| 3 | $\mathrm{ACC} \leftarrow \mathrm{FIFO} \times \mathrm{R}_{inB}$ - ACC |
| 4 | $\mathrm{R}_2 \leftarrow \mathrm{ACC}$ |
| 5 | $\mathrm{R}_A \leftarrow \mathrm{R}_2$ |
| 6 | $\mathrm{R}_B \leftarrow \mathrm{R}_2$ |
| 7 | $\mathrm{ACC} \leftarrow \mathrm{R}_A \times \mathrm{R}_B$ |
| 8 | $\mathrm{R}_2 \leftarrow \mathrm{ACC}$ |
| 9 | $\mathrm{R}_A \leftarrow \mathrm{R}_2$ |
| 10 | $\mathrm{R}_{inA} \leftarrow$ input $\frac{1}{m_k}$ |
| 12 | $\mathrm{R}_B \leftarrow \mathrm{R}_{inA}$ |
| 13 | $\mathrm{ACC} \leftarrow \mathrm{R}_A \times \mathrm{R}_B$ |
| 14 | $\mathrm{R}_2 \leftarrow \mathrm{ACC}$ |
| 15 | $\mathrm{FIFO} \leftarrow \mathrm{R}_2$ |

Table 4.6: Instructions to compute $\psi^2$.

in Figure 4.11.

## 4.6 Discussion

In this chapter, we presented the architectures for a system that can perform the HOC analysis efficiently. We also presented a detailed design of a prototype HOC-$\psi^2$ PCB system using off-the-shelf components. The designed board provides flexibility and programmability for various applications. As the prototype system uses off-the-shelf components, the performance is restricted by the microprocessor-based controller and the elements used. The microprocessor-based controller was set to run at the clock rate 11MHz. We believe that the performance will improve greatly if the system is implemented in VLSI circuitry. The decision of whether or not to use ASICs depends on the ultimate use of the circuit and the required specifications.

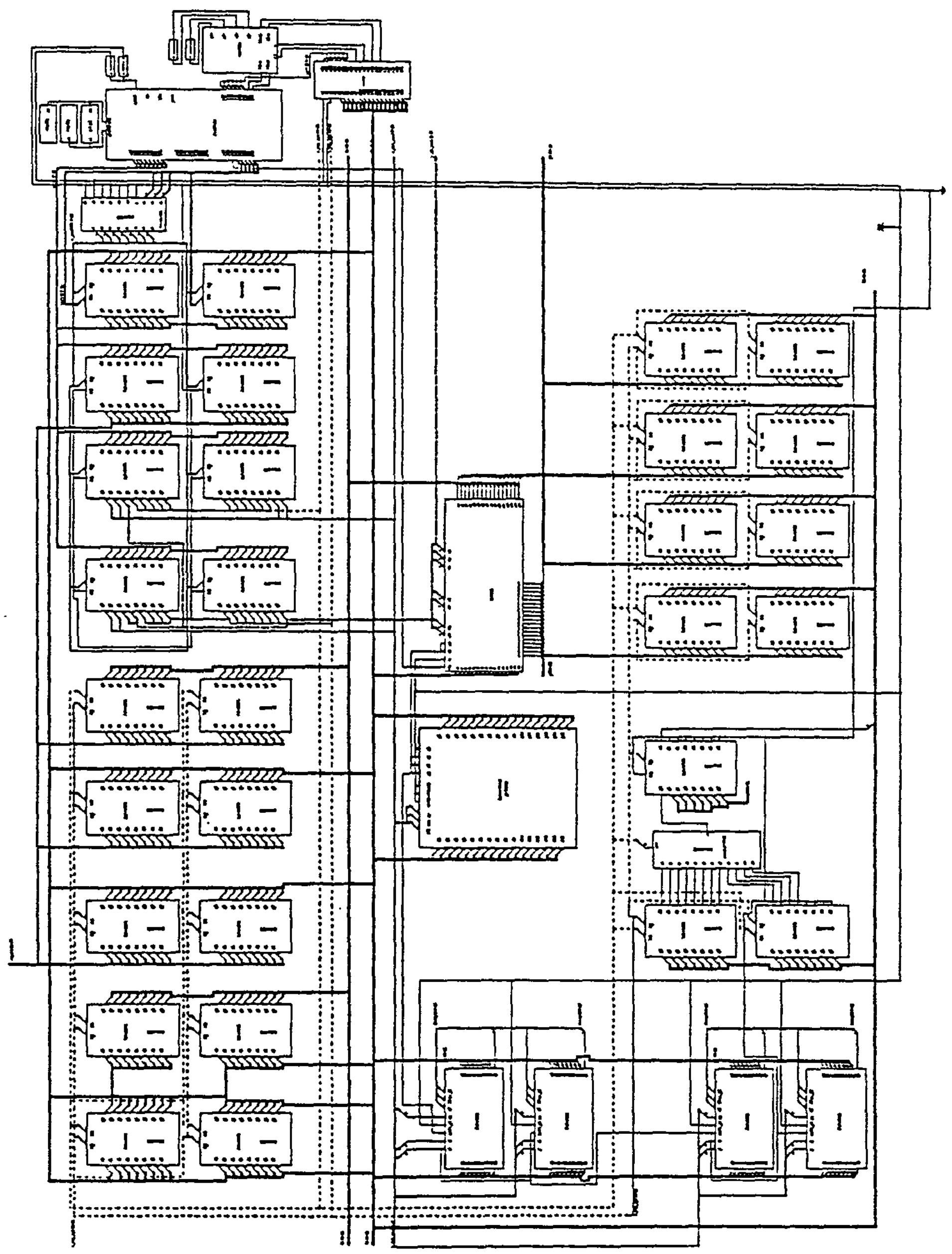


Figure 4.10: Plot of the schematic for HOC.

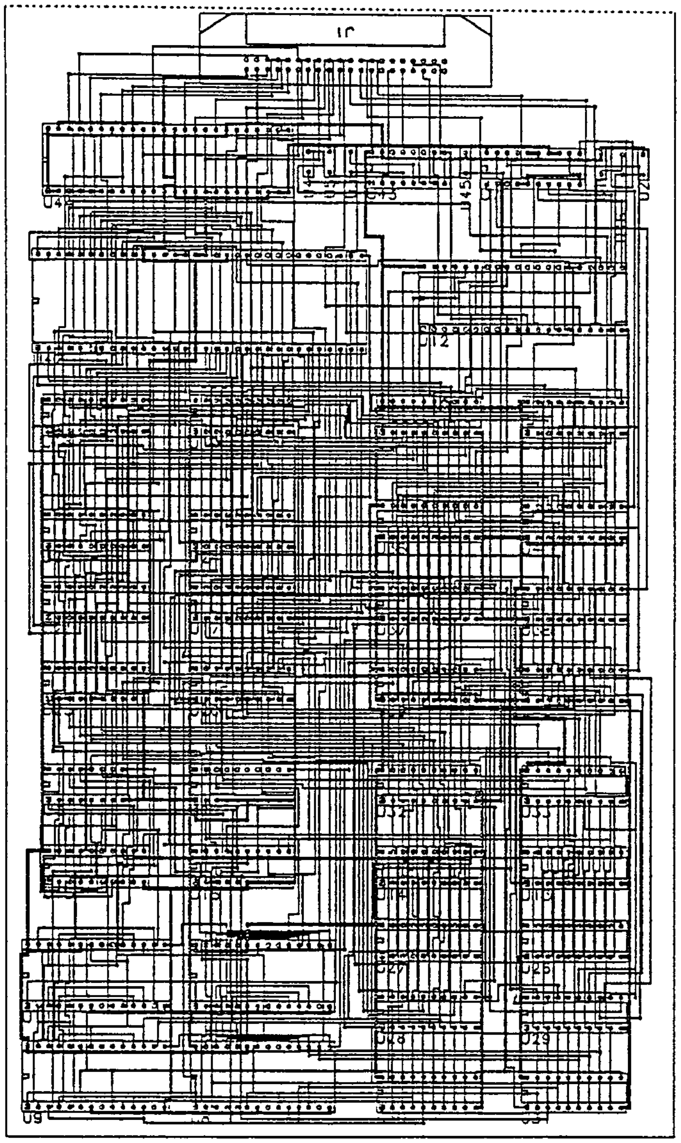


Figure 4.11: Plot of the PCB layout for HOC.

# Chapter 5

# Conclusions

In this dissertation several systolic architectures are proposed for some real-time signal processing applications and implementations, namely, Tree-Structured Vector Quantizer, Predictive Tree-Structured Vector Quantizer, and High Order Crossings System. The systolic concept is mainly adapted in designing these architectures for their simplicity, regularity, high concurrency, local communication, and high throughput.

We presented an architecture and VLSI implementation of a TSVQ. The Tree-Structured Vector Quantizer is mapped into a linear array of identical processor elements. The TSVQ architecture uses identical processors at each level of the binary tree. The architecture is fully pipelined, and latency is 100 clock cycles per processor when the block size is $8 \times 8$ pixels. These processors have been fabricated using $2\mu m$ N-well process through MOSIS. The die size is $7.9mm \times 9.2mm$. The processor chips have been thoroughly tested and found to be fully functional at a frequency of 20 MHz. Using these TSVQ processors, the VQ-DCT-SQ system can process 1 pixel/clock or 160 Mbits/sec. Using two such systems in parallel, we can achieve a data rate of 320 Mbits/sec.

A new Predictive TSVQ architecture is presented for real-time video coding applications. Pipelined arithmetic components are used to speed up the computation and to provide for regularity in design. This high throughput architecture is suitable for implementing a fully pipelined real-time PTSVQ system. This architecture has been implemented as a VLSI chip set using $1.2\mu m$ CMOS technology. Identical processors are used for both the encoding and decoding components. Spice simulations indicate correct operation at 40 MHz. For a typical real-time image processing system with 30 frames/sec and $1024 \times 1024$ pixels/frame, the input pixel rate is 31.5 Mpixels/sec. This architecture is capable of processing 40Mpixels/sec and can handle the above case in real-time. We fabricated prototype versions of these chips using $2\mu m$ CMOS technology. These prototype chips work at 20 MHz. Our architecture can be extended easily to handle other classes of VQ with memory such as Trellis VQ.

We presented the architectures for a system that can perform the HOC analysis efficiently. We also presented a detailed design of a prototype HOC-$\psi^2$ PCB system using off-the-shelf components. The designed board provides flexibility and programmability for various applications. As the prototype system uses off-the-shelf components, the performance is restricted by the microprocessor-based controller and the elements used. The microprocessor-based controller was set to run at the clock rate 11MHz. We believe that the performance will improve greatly if the system is implemented in VLSI circuitry.

# Bibliography

[1] J. W. Cooley and J. W. Tukey, "An alogrithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, Apr. 1965.

[2] K. Marrin, "SPARC scales up," *SunWorld*, pp. 82–88, July 1992.

[3] C. Peterson, J. Sutton, and P. Wiley, "iWarp: A 100-MOPS, LIW microprocessor for multicomputers," *IEEE Micro*, p. 26, June 1991.

[4] H. T. Kung, "Why systolic architectures?," *IEEE Trans. Computers*, pp. 37–46, Jan. 1982.

[5] N. Jayant, "Signal compression: Technology targets and research directions," *IEEE J. Select. Areas Commun.*, vol. 10, pp. 796–818, June 1992.

[6] N. S. Jayant and P. Noll, *Digital coding of waeforms: principles and applications to speech and video.* Prentice Hall, 1984.

[7] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression.* Kluwer Academic Press, 1992.

[8] N. Nasrabadi and R. King, "Image coding using vector quantization: A review," *IEEE Trans. Commun.*, vol. COM-36, pp. 957–971, Aug. 1988.

[9] B. Kedem, *Time Series Analysis by Higher Order Crossings.* IEEE Press, 1994.

[10] T. Li, *Multiple Frequency Estimation in Mixed-Spectrum Time Series By Parametric Filtering.* PhD thesis, University of Maryland, 1992.

[11] P. Dickstein, Y. Segal, E. Segal, and A. Sinclair, "Statistical pattern recognition techniques: A sample problem of ultrasonic determination of interfacial weakness in adhesive joints," *J. NDE*, vol. 8, pp. 27–35, 89.

[12] P. Dickstein, J. Spelt, and A. Sinclair, "Application of a higher order crossing feature to non-destructive evaluation: A sample demostration of sensitivity to the condition of adhensive joints," *Ultrasonics*, vol. 29, pp. 355–365, Sept. 1991.

[13] R. Jain, A. Madisetti, and R. L. Baker, "An integrated circuit design for pruned tree-search vector quantization encoding with an off-chip controller," *IEEE Trans. on Circuits and Systems for Video Technology*, pp. 147–158, June 1992.

[14] R. K. Kolagotla, S.-S. Yu, and J. F. JáJá, "VLSI implementation of a tree searched vector quantizer," *IEEE Trans. Signal Processing*, vol. 41, pp. 901–905, Feb. 1993.

[15] T. Lookabaugh, "Architectures for tree structured vector quantization." Unpublished work, May 1987.

[16] D. Y. Cheng and A. Gersho, "A fast codebook search algorithm for nearest-neighbor pattern matching," in *Proc. IEEE Int'l. Conf. on Acoustics, Speech and Signal Processing*, pp. 265–268, 1986.

[17] M. Yan and J. McCanny, "A bit-level systolic architecture for implementing a VQ tree search," *Journal of VLSI Signal Processing*, vol. 2, pp. 149–158, Nov. 1990.

[18] W. C. Fang, C. Y. Chang, and B. J. Sheu, "Systolic tree-structured vector quantizer for real-time image compression," in *VLSI SIgnal Processing IV* (K. Yao, ed.), Nov. 1990.

[19] T. Markas, J. Reif, W. Elliot, and E. Elliot, "Memory-shared parallel architectures for vector quantization algorithms." Private communication, Nov. 1991.

[20] A. Madisetti, R. Jain, R. L. Baker, and R. Dianysian, "Architectures and integrated circuits for real time vector quantization of images," in *Proc. IEEE Int'l. Conf. on Acoustics, Speech and Signal Processing*, pp. V–677–680, 1992.

[21] E. A. Riskin, *Variable Rate Vector Quantization of Images.* PhD thesis, Stanford Univerity, May 1990.

[22] J. Kraus, J. Reimers, and K. Grüger, "A VLSI chip set for DPCM coding of HDTV signals," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 3, pp. 302–308, Aug. 1993.

[23] S. S. Yu, R. K. Kolagotla, and J. F. JáJá, "VLSI architectures and implementation of predictive tree-searched vector quantizers for real-time video compression," in *Proceedings of the International Conference on Application-Specific Array Processors*, pp. 481–495, Aug. 1992.

[24] N. N. Hsu and D. G. Eitzen, "Higher-order crossings – a new acoustic emission signal processing method," *Porgress in Acoustic Emission IV*, pp. 59–66, 1988.

[25] G. Davidson, P. Cappello, and A. Gersho, "Systolic architectures for vector quantization," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 36, pp. 1651–1664, Oct. 1988.

[26] J. V. McCanny and J. G. McWhirter, "Completely iterative, pipelined multiplier array suitable for VLSI," *IEE Proc.*, vol. 129, pp. 40–46, Apr. 1982.

[27] P. Chang and R. M. Gray, "Gradient algorithms for design predictive vector quantization," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 679–690, Aug. 1986.

[28] A. Haoui and D. Messerschmitt, "Predictive vector quantizer," in *Proc. IEEE Int'l. Conf. on Acoustics, Speech and Signal Processing*, 1984.

[29] V. Cuperman and A. Gersho, "Vector predictive coding of speech at 16 kbits/s," *IEEE Trans. Commun.*, vol. COM-33, pp. 685–696, July 1985.

[30] H. M. Hang and J. W. Woods, "Predictive vector quantization of images," *IEEE Trans. Commun.*, vol. COM-33, no. 11, pp. 1208–1219, 1985.

[31] V. Bhaskaran, "Predictive VQ schemes for grayscale image compression," in *Proc. GLOBECOM '87*, pp. 11.8.1–11.8.6, Dec. 1987.

[32] J. W. Modestino and Y. H. Kim, "Adaptive entropy-coded predictive vector quantization of images," *IEEE Trans. Signal Processing*, vol. 40, pp. 633–644, Mar. 1992.

[33] E. Slutzky, "The summation of random causes as the source of cyclic processes," *Econometrica*, vol. 5, no. 1, pp. 105–146, 1937.

[34] C. Guyott and P. Cawley, "The ultrasonic vibration characteristics of adhesive joints," *J. Acoust. Soc. Am.*, vol. 83, no. 2, pp. 632–640, 1988.

[35] P. Dickstein, S. Girshovich, Y. Sternberg, A. Sinclair, and H. Leibovitch, "Ultrasonic feature-based classification of the interfacial condition in composite adhesive joints," *J. Res. NDE*, vol. 2, pp. 207–224, 1990.

[36] T. Anderson, *An Introduction to Multivariate Statistical Analysis(2nd ed.)*. New York: Wiley, 1984.

[37] B. Kedem and E. Slud, "On goodness of fit of time series models: An application of higher order crossings," *Biometrika*, vol. 68, pp. 551–556, Aug. 1981.

[38] S. Kay and S. Marple, "Spectrum analysis – a modern perspective," *Proc. IEEE*, vol. 69, pp. 1380–1419, Nov. 1981.

[39] B. Kedem and S. Yakowitz, "On the contraction mapping method for frequency detection," Tech. Rep. SRC TR 92–45, University of Maryland, 1992.

[40] P. Whittle, "The simultaneous estimation of time series' harmonic components and covariance structure," *Trab. Estad.*, vol. 3, pp. 43–57, 1952.

[41] A. M. Walker, "On the estimation of a harmonic component in a time series with stationary independent residuals," *Biometrika*, vol. 58, pp. 21–36, 1971.

[42] A. M. Walker, "On the estimation of a harmonic component in a time series with stationary independent residuals," *Adv. Appl. Prob.*, vol. 5, pp. 217–241, 1973.

[43] E. Hannan, "The estimation of frequency," *J. Appl. Prob.*, vol. 10, pp. 510–519, 73.

[44] P. Stoica and A. Nehorai, "The summation of random causes as the source of cyclic processes," *Circuits, Systems, Signal Processing*, vol. 8, no. 1, pp. 3–15, 1989.