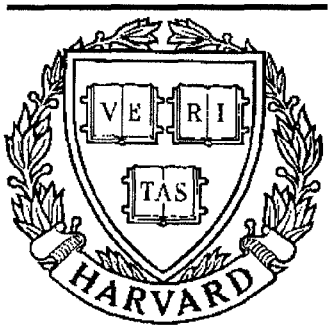
THESIS REPORT
Ph.D.

SYSTEMS
RESEARCH
CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

Design and Implementation of Systolic Architectures for Vector Quantization

by R.K. Kolagotla
Advisors: J.F. JáJá

Ph.D. 92-11

Design and Implementation of Systolic Architectures for Vector Quantization

by

Ravi K. Kolagotla

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
1992

Advisory Committee:

Professor Joseph F. JáJá, Chairman/Advisor
Associate Professor Nariman Farvardin
Assistant Professor K. J. Ray Liu
Assistant Professor Linda Milor
Professor Larry S. Davis

Abstract

Title of Dissertation: Design and Implementation of Systolic Architectures for Vector Quantization

Ravi K. Kolagotla, Doctor of Philosophy, 1992

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

Vector Quantization has emerged as an efficient data compression tool for compressing speech and image data. We develop efficient systolic architecture implementations of Tree-Search Vector Quantizers (TSVQ) and Finite-State Vector Quantizers (FSVQ). Our TSVQ architecture consists of a linear array of processors, each processor performing the computations required at one level of the binary tree. Encoding is performed in a pipeline fashion with each processor generating a portion of the path through the tree. The final processor returns the complete index. Data and control flow from processor to processor along the pipeline and no global control signals are needed. The FSVQ architecture for image coding consists of a linear array of TSVQ processors with each processor operating on a separate column of the input image. The number of processors needed depends on the latency of the TSVQ and is independent of the size of the image. We also develop implementations of Scalar and Inverse Scalar Quantizers for use in transform coding applications.

Dedication

To Hsien-Hua Lee

Acknowledgements

I would like to thank my advisor Professor Joseph F. JáJá, for his guidance and support. I had many valuable discussions with him and with Professors Nariman Farvardin and K.J. Ray Liu. I am grateful for their help and advice. I would also like to thank Professors Evaggelos Geraniotis, Hung C. Lin, Bernard Menezes, Linda Milor, and Kazuo Nakajima for helping me in various ways during my study at Maryland.

I learned many of my skills by interacting with several people at Maryland. Special thanks go to Jagannathan Narasimhan for teaching me the basics of C, to György Fekete for getting me started as a UNIX[1] super-user, to Hsien-Hua Lee for patiently answering all my hardware and system software questions, to Haisong Cai for sharing many of his UNIX secrets with me, and to Steve Miller for solving system problems that were too difficult for me.

Most of my research work was done as part of several group projects. Special thanks go to Shu-Sun Yu for the innumerable discussions we had during the course of the SQ and VQ projects, to Ching-Te Chiu for collaborating on the DCT/DST project, to Xiaonong Ran for helping me with the TSVQ project, to Yunus Hussain for helping me with the FSVQ project, to Todd Goodrich for the discussions we had during the course of the DFT project, to John Baczewski for helping me test my very first chip, and to Karl Thompson, and Wen-Bin Yang for their help with the FM synchronizer project.

Several students, both past and present, have contributed directly or indirectly to this thesis. I would like to thank Chaitali Chakrabarti, Shing-Chong

[1]UNIX is a registered trademark of UNIX System Laboratories, Inc.

Chang, Chieh-Yuan Chao, Jie Chen, Martin P. Farach, Ying-Min Huang, Vivek Khuller, Sridhar Krishnamurthy, Rajiv Laroia, Chujen Lin, Jyh-Fong Lin, Po-Yang F. Lin, Zhiming Lin, Sayed Naved, Madhura Nirkhe, Nam Phamdo, Raju Prasannappa, Ravi Ramaswami, Ivan L. M. Ricarte, Chong-Suk Rim, Andrew Robertson, Kwan-Woo Ryu, Sachidanandan Sambandan, Malathy Sethuraman, Vishnu Srinivasan, Naoto Tanabe, Hao Tian, Amarendranath Vadlamudi, Vinay Vaishampayan, Hsinshih Wang, Zhusheng Wang, and Sen-Jung Wei.

I would like to thank the University of California at Berkeley, Stanford University, Digital Western Research Laboratory, and Lawrence Livermore National Labs for developing the excellent CAD tools without which this work would not have been possible. I would like to thank Professor Chuck Sietz of CalTech for developing the I/O pads we used on most of our chips.

Finally, I would like to thank Gail Belshay, Terry Clark, Sharon Dass, Ursula Gedra, Olivia Goetz, Pam Harris, Matt Katsouros, Daw-Tung Lin, Yolanta Stawski, Kim-Thu Tran, Maggie Virkus, and Tina Wong for making EE, SRC, and UMIACS pleasant places to work for.

Contents

List of Figures

Chapter 1

Introduction

Special purpose VLSI architectures are used to achieve the high throughput rates and processing power required by real-time signal processing applications. Until recently, general purpose computers were unable to provide the processing power required for these applications. However, in the past few years, the computing power of general purpose workstations has increased by two orders of magnitude, while their cost remained relatively constant. The current generation of desktop workstations have processing powers in excess of 100 MIPS [1]. Pipelining and parallelism, once the exclusive domain of special purpose VLSI architectures, are now routinely incorporated in general purpose ASICs and workstations [2, 3]. This naturally leads to the following question: of what possible use are special purpose VLSI architectures?

While the processing power of general purpose workstations has increased tremendously, their data throughput rates have only increased modestly to about 10 Mbytes/sec. Real-time applications require the sustaining of much larger throughput rates. For example, HDTV generates about 70 Mbytes/sec of image data. Special purpose VLSI architectures that are tailored for a particular

application, can balance the computational power with interprocessor data communication to achieve real-time performance. Hence, special purpose VLSI architectures will be used in the foreseeable future because of their high throughput rates, and because of their ability to optimally balance the various components of the architecture for a specific application. However, special purpose VLSI chips have higher design and production costs because of their low volume. They are used when the real-time performance requirements justify the cost involved.

Data compression is an important signal processing task that has been the subject of extensive studies due to its many applications. The introduction of ISDN in standard commercial workstations [1] and the development of global standards for high speed data and video communications will make data compression even more important in the near future. This has led to the development of several important lossy compression techniques such as predictive coding, block transform coding, vector quantization, and sub-band coding. While a significant amount of work has been done in the development and software implementation of data compression algorithms, not much has been done in terms of real-time hardware implementation of these algorithms [4, 5]. In this thesis, we develop efficient VLSI architectures for the real-time implementation of data compression techniques.

1.1 Overview of VLSI Architectures

In this section we review the major properties of special purpose VLSI architectures that make them attractive for various real-time signal processing applications [6, 7, 8, 9].

1.1.1 Simple and regular designs

Special purpose VLSI architectures can exploit the regularity and modularity of the signal processing algorithms in order to achieve compact and efficient designs. Architectures that are assembled using building blocks can be designed faster at low cost. For example in the design of a parallel multiplier, it is sufficient to design one unit cell and replicate it N^2 times, rather than design the entire multiplier from scratch. Modular designs also reduce the chances of design errors in the final VLSI chip.

1.1.2 Localized communications

Data communications between different modules on a chip and between chips is very costly in terms of the area they occupy and the timing constraints they impose on the speed of operation of the circuit. Special purpose VLSI architectures that are optimally mapped from the underlying algorithm require significantly less interconnections than a general purpose architecture that is not optimized for a particular application. In mapping an algorithm to a special purpose architecture interconnection length in both the data and control paths is one of the major design constraints.

1.1.3 Massive concurrency

In real-time implementations, the processors must be capable of sustaining the input data rate. In typical video applications, input data arrives word serially at rates in excess of 50 Mpixels/sec. Special purpose VLSI architectures can sustain this large input rate by using extensive pipelining and parallelism in

the design. Pipelining increases the data throughput rate at the expense of an increased latency [10]. Latency is usually not a primary concern in signal processing systems where feedback loops are not present. Parallelism allows the processing of multiple data simultaneously at the expense of an increased hardware area. If each processor is small and efficient, VLSI technology allows the design of complex chips with several processors in parallel.

1.2 VLSI Implementation Techniques

VLSI implementations can be broadly classified into full-custom and semi-custom designs.

Full-custom designs use a polygon editor to directly modify the VLSI layout. This is used in cases where the layout area and speed of operation are critical. Memories and other modules that require numerous copies of the same basic circuit are usually designed using full-custom techniques. Because of arraying, any savings in the area of the basic circuit is magnified in the final layout.

In semi-custom designs, all the basic subcells are taken from a standard cell library. They are assembled using placement and routing tools based on a specified circuit schematic. This technique is best suited for the control and other portions of the chip that are much easier to describe at a schematic level and may undergo modification during the design cycle. Automatically placed and routed modules use more area than manually designed ones. This can be a major disadvantage in designs where chip area is at a premium.

The ideal approach is to manually design all critical modules and leave the smaller time-consuming modules to the automatic place and route tools.

1.3 Data Compression

The goal of data compression is to reduce the channel capacity required for transmission of signals and the storage capacity needed to store the signals. Data compression also helps in making the best use of available storage and channel capacities. Speech and images are two of the main sources of digital data. Speech data is typically generated at a rate of a few thousand samples per second, while image and video data are generated at a rate of several tens of millions of samples per second. Hence, any real-time hardware implementation of data compression algorithms must be capable of processing input data at video rates.

1.3.1 Scalar Quantization

Scalar Quantization (SQ) is a technique in which each input sample is represented by the index of the nearest quantization level from a predetermined set of quantization levels. If the input is an analog signal, SQ is part of the Analog to Digital (A/D) conversion process. In general, SQ allows the input to be in digital form. Compression is achieved by using a coarse set of quantization levels such that the number of bits required to represent the quantization level is smaller than the number of bits required to represent the original input signal.

A SQ is completely defined by its set of quantization and reconstruction levels [11, 12]. Let $x_0, x_1, \ldots, x_N$ be the ordered set of $N+1$ decision levels and $y_1, y_2, \ldots, y_N$ be the set of N reconstruction or output levels. If the input signal x, lies between x_{i-1} and x_i, the SQ encoder represents it with the index i. This index i is stored or transmitted instead of the original signal x. An SQ decoder,

or an Inverse Scalar Quantizer (ISQ), transforms index i into the reconstruction level y_i. The difference between the original signal x and its reconstruction level y_i, is the quantization error.

The average error depends on the statistics of the input signal. Optimum quantizers can be designed for various input probability distributions [13].

1.3.2 Vector Quantization

In SQ, each input sample is quantized independently. Adjacent input samples from real world signal sources usually have a high degree of correlation. This correlation can be exploited in a block structured data compression scheme, in which an input data is segmented into blocks of equal size and the samples in each block are quantized together. Vector Quantization (VQ) provides the best performance among all block structured image coding schemes for a given blocksize and bit-rate [4].

Given a k-dimensional input vector $\mathbf{x}$, a VQ encoder chooses a reproduction vector $\hat{\mathbf{x}}$ from a predetermined set of N reproduction vectors that is closest to the input vector relative to a certain distortion measure. The Euclidean distance is the most commonly used distortion measure. The reproduction vectors, also known as codevectors, are points in k-dimensional space. The input vector is then represented by the index u of codevector $\hat{\mathbf{x}}$. This index, also known as the channel symbol, is transmitted to the decoder. A VQ decoder maps this channel symbol onto its corresponding reproduction vector from the codebook [14].

Full-Search VQ

The VQ encoder must compare the input vector with each reproduction vector in the codebook. For a codebook consisting of N codevectors, the computational complexity of the encoder is $O(N)$ distance computations. For a k-dimensional input vector, assuming Euclidean distortion measure, this involves $\Theta(kN)$ multiplications and $\Theta(kN)$ additions for each input vector. At video rates, this becomes quite prohibitive even for modest values of k and N [15, 16].

Tree-Search VQ

Several techniques have been developed to reduce the encoding complexity of VQs at the expense of an increased error between the original input vector and its reconstructed output. Reduced complexity allows the design and implementation of VQs for higher dimension vectors than is possible with full-search VQs.

In a Tree-Search VQ (TSVQ), the codebook is structured as a binary or higher-order tree and the search for the nearest codevector for a given input vector is performed in stages. At each level of the tree, the input vector is compared with a few codevectors, and based on the results, the codebook search space is reduced. This process is repeated until a leaf node is reached. The input vector is then represented by the path taken through the tree by the TSVQ encoder.

In addition to a reduced signal to noise ratio performance, a TSVQ encoder requires twice as much memory as a full-search VQ to store the codebook.

Multi-Stage VQ

A Multi-Stage VQ (MSVQ) encodes an input vector in successive stages [17]. Each stage of a MSVQ is a full-search VQ with a small number of codevectors. Unlike the TSVQ, in which each stage quantizes the input vector in a small region of the codebook space, each stage in a MSVQ quantizes the error between the input vector and the reconstructed output of the previous stage. The MSVQ encoding process can be viewed as successive refinement, in which the input vector is more accurately represented as the encoder proceeds down the tree. MSVQ thus combines the low encoder complexity of a TSVQ with the smaller storage requirements of a full-search VQ at the expense of a reduced signal to noise ratio performance [18].

Predictive VQ

In a full-search VQ, each input vector is encoded independent of the neighboring vectors. In real-world sources, neighboring vectors usually have a high degree of correlation. Predictive VQ (PVQ) is a VQ with memory which makes use of inter-vector correlation to predict the current input vector based on past outputs. The difference between the actual input vector and its predicted value is then quantized using either a full-search VQ or a TSVQ [19, 20, 21, 22]. Since the error vector is constrained to lie in a smaller region of k-dimensional space than the input vector itself, it can be accurately represented with a smaller number of bits. Alternately, for the same number of transmitted bits, a PVQ provides better signal to noise ratio than a full-search VQ.

Finite-State VQ

Finite-Stage VQ (FSVQ) is a VQ with memory in which the past outputs are used to restrict the search space for the current input vector. The entire codebook search space is divided into a set of states, with each state consisting of a small set of codevectors. The current input vector is encoded using the codevectors from one of these states. The current state of the FSVQ encoder depends on the past outputs. FSVQ achieves the efficiency of a large rate VQ at a relatively small rate.

1.4 Main Contributions

In this thesis, we develop efficient VLSI architectures for implementing Tree Search Vector Quantizers (TSVQ) and Finite State Vector Quantizers (FSVQ) in real-time. These architectures can be used in any speech or image compression application based on VQ. In this section, we describe both the TSVQ and FSVQ algorithms. We also describe the existing schemes for their hardware implementation and give an overview of our architectures.

1.4.1 Tree Search Vector Quantizers

VQ codebooks are typically structured as trees to reduce the codebook search complexity and simplify hardware implementation. A Tree Search VQ (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 [23].

Existing schemes:

Several researchers have implemented TSVQs in hardware concurrently with our work. TSVQ architectures were first proposed by Lookabaugh [24]. The scheme to exploit binary Hyperplane testing [25] for generating efficient TSVQ architectures was first proposed by Lookabaugh. Hardware implementations were proposed by Yan and McCanny [26] and Wai-Chi Fang *et.al.* [27] 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.* [28] and Madisetti *et.al.* [23, 29] have proposed TSVQ architectures.

Our contribution:

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 [30]. 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

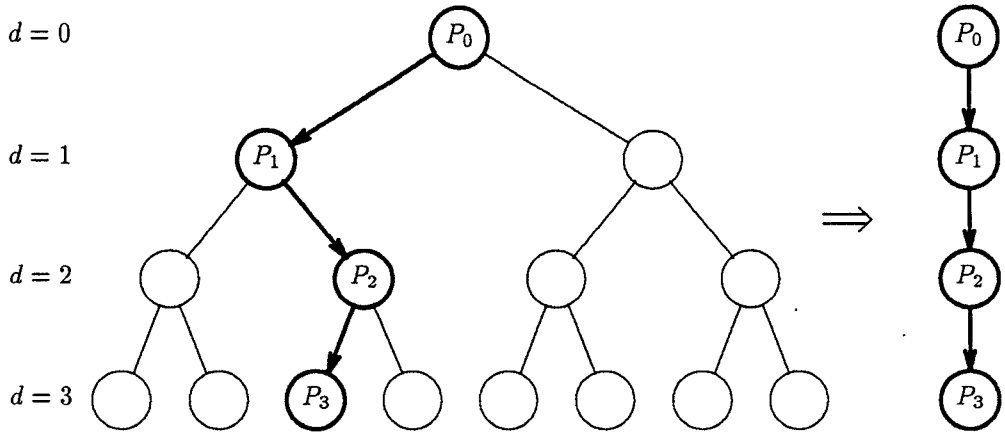


Figure 1.1: Traversal of a binary tree of depth 4, and its mapping onto a linear array of processors.

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 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.

1.4.2 Finite State Vector Quantizers

Increasing the vector dimension for a given bit-rate improves the performance of VQ. However, the resulting increase in complexity makes large vector dimension VQ or TSVQ impractical. Finite State VQ (FSVQ) is a class of VQ with memory which makes use of correlation between neighboring vectors to improve

performance for a given vector dimension and bit-rate [31]. An FSVQ consists of a super-codebook which contains a large number of codevectors and an internal state which accurately represents a small region that contains the current input vector. The quantizer codebook used for the current input vector depends on past outputs. Thus, FSVQ achieves the performance of a large rate codebook at a relatively small rate. Finite State TSVQ (FSTSVQ), which is a TSVQ with memory, exhibits a better signal to noise ratio performance than a simple TSVQ, while maintaining its $O(\log N)$ codebook search complexity. FSTSVQ can provide a better compression ratio than a TSVQ using about the same computation and memory resources [4, 32, 33, 34, 35].

Existing schemes:

Shen and Baker have proposed a FSVQ for interframe and intraframe coding [36]. They suggest a blend of classified VQ, PVQ, Product VQ and FSVQ for improved signal to noise ratio performance. They suggest implementing the VQ portion of this encoder with a custom VLSI chip set that requires a complex control scheme [37].

Our contribution:

Our architecture for implementing an FSVQ encoder in real-time for image data consists of a linear systolic array of processors. The number of processors needed is independent of the size of the image [38]. Each processor consists of a TSVQ, a next state generator, and additional memory for storing the codebooks associated with each state. This architecture can efficiently exploit the correlation between the current input block and all of its nearest causal neighbors. Data flow is

regular and there are no global control signals.

1.5 Thesis Organization

The rest of this thesis is organized as follows: In Chapter 2, we describe the multiplier and other basic building blocks used in our VQ implementations. We also describe the design and implementation of Scalar Quantizers (SQ) and Inverse Scalar Quantizers (ISQ). In Chapter 3, we describe the design and VLSI implementation of a systolic architecture for TSVQ. In Chapter 4, we develop a systolic architecture for efficiently implementing FSVQ in real time for both speech and image data. We summarize our results in Chapter 5.

Chapter 2

Basic Building Blocks

In this chapter we present some basic building blocks used in the TSVQ and FSVQ designs. The multiplier and accumulator are two of the most fundamental blocks used in any signal processing application [39]. We also present an architecture for converting line-scanned input image data into block-scanned mode. The image architectures developed later operate on input image blocks.

2.1 Multipliers

Several different multiplier designs are available for digital signal processing applications. They offer trade-offs between speed and area complexity [40]. In this section, we describe the pipelined parallel multiplier, the distributed arithmetic multiplier and the bit-serial multiplier. We also describe VLSI chip designs that incorporate these multipliers.

2.1.1 Bit-Parallel Multipliers

Parallel multipliers consist of a two-dimensional array of basic cells. Each cell consists of a full adder and latches and computes a single partial product. The complete product is obtained by accumulating the partial products generated by each cell. The multiplier can be pipelined by adding delay elements between the cells and applying the inputs in a skewed fashion [41]. Pipelining increases the speed of operation of parallel multipliers by reducing the complexity of the computations performed in each clock cycle. This is achieved at the expense of an increased latency. We have used pipelined parallel multipliers as the basic modules in both TSVQ and FSVQ. We have also used variations of these for designing synchronizers and demodulators for FM sideband receivers. Fig. 2.1 shows the block diagram of each cell. A parallel multiplier is formed as a 2-D array of these cells. Signals "ain" and "bin" are bits of the input numbers to be multiplied. Signal "ain" flows vertically through the array and signal "bin" flows horizontally through the array. Signal "sin" flows diagonally through the cells. Carry input signal "cin" flows vertically through the array. Each cell consists of a full adder which adds the partial product (product of "ain" and "bin") with the partial sum input "sin" and the partial carry input "cout". The output signals "aout", "bout", "cout", and "sout" are delayed and applied to the adjacent cells. Fig. 2.2 shows the layout of a 12-bit by 8-bit multiplier.

2.1.2 Distributed Arithmetic Multipliers

In applications where an input number must be multiplier by a known constant, the memory oriented distributed arithmetic multipliers can be used. They are

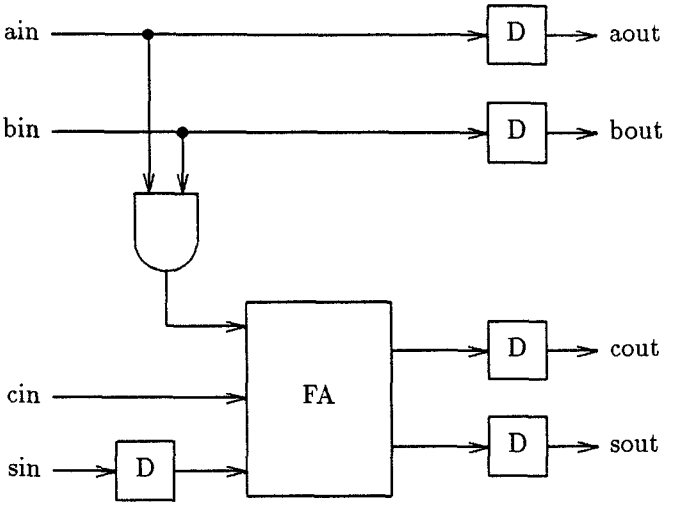


Figure 2.1: Logic diagram of the bit-parallel multiplier cell.

faster and provide more accuracy than bit-parallel multipliers [42, 43]. They also use a smaller chip area. The disadvantage with distributed arithmetic multipliers is that for similar delay constraints, they scale exponentially with the word size. Hence, fast distributed arithmetic implementations are feasible only for small word sizes.

We implemented bit-parallel distributed array multipliers for multiplication by known constants for use in a combined DCT/DST chip [44]. In this application, the input number, x, is in 12-bit precision two's complement binary, little endian format. It is multiplied with two fixed coefficients c and s to generate xc and xs. Using the distributed arithmetic method, the input number,

$$x = \sum_{0}^{10} 2^j x_j - 2^{11} x_{11},$$

can be represented as, $x = a + 2^6 b$, where,

$$a = \sum_{0}^{5} 2^j x_j,$$

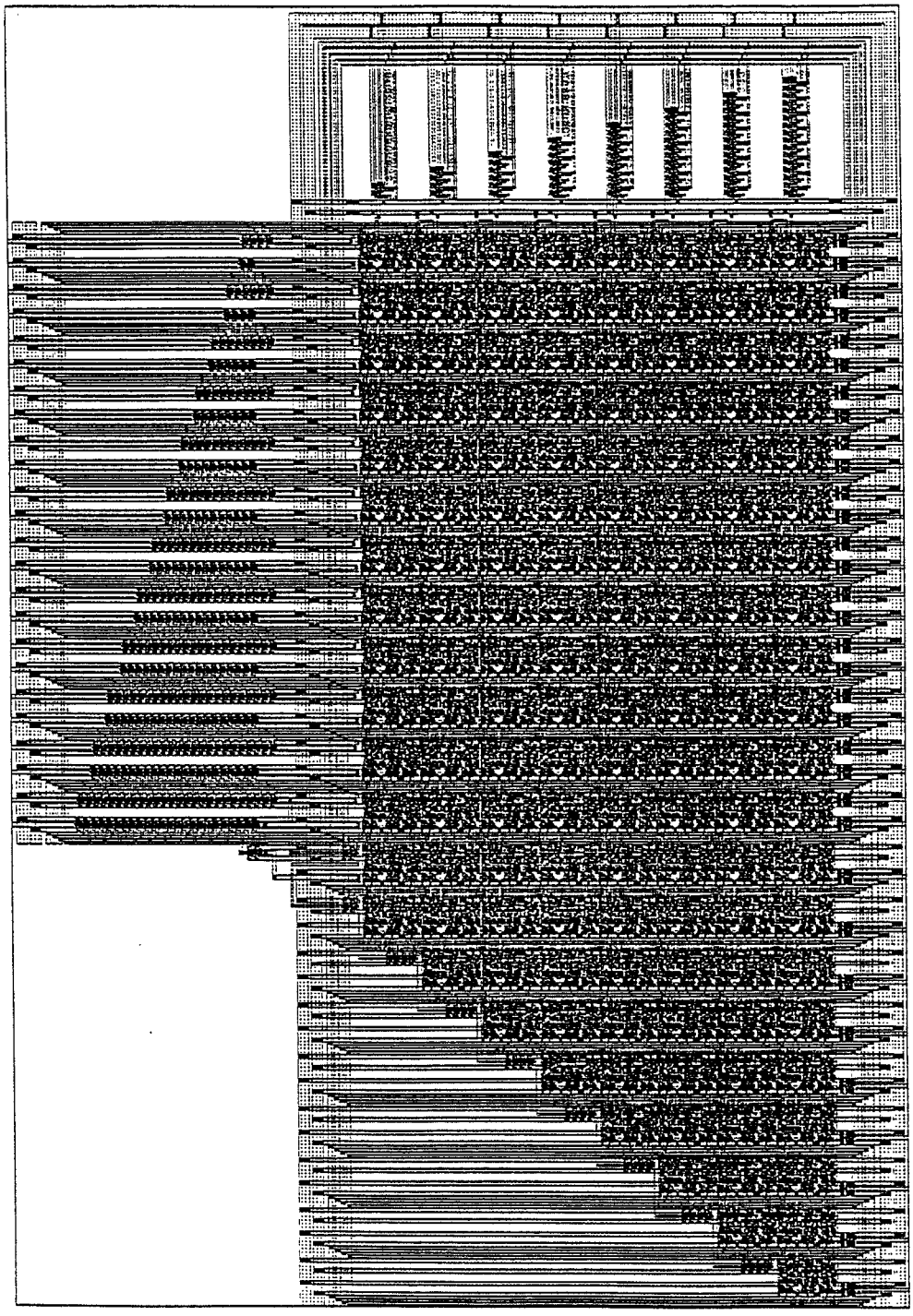


Figure 2.2: Physical layout of the pipelined parallel multiplier for 12-bit numbers and 8-bit coefficients. Area of this module is $4364\lambda \times 2913\lambda$.

and

$$b = \sum_{0}^{4} 2^j x_j - 2^5 x_{11}.$$

Partial products ac, as, bc, and bs are precomputed for all possible values of a and b and stored in memories. Multiplication is achieved by first using the appropriate input bits to address the memories. The values returned by the memories are then added together to generate the correct products as follows,

$$xc = ac + 2^6 bc,$$

and

$$xs = as + 2^6 bs.$$

Since the input is multiplied with two coefficients, the decoder circuitry can be shared between the memories. Fig. 2.3 shows the layout of this multiplier.

2.1.3 Bit-Serial Multipliers

Bit-serial architectures offer numerous advantages over bit-parallel architectures. Bit-serial architectures require a much smaller chip area and can be operated at a much higher clock-rate [45]. Bit serial multipliers have been developed for signed magnitude numbers [46] and for fully two's complement numbers [47, 48]. In this section, we describe the pipeline multiplier developed for the Cascadable Lattice FIR Filter. We modified the design in [47] to allow the coefficients to be programmed off-line.

The bit-serial multiplier consists of a linear array of cells. The coefficient is applied MSB first to the "bin" input during the coefficient load operation. After the coefficient is loaded, the bits of the input signal are applied LSB first to the

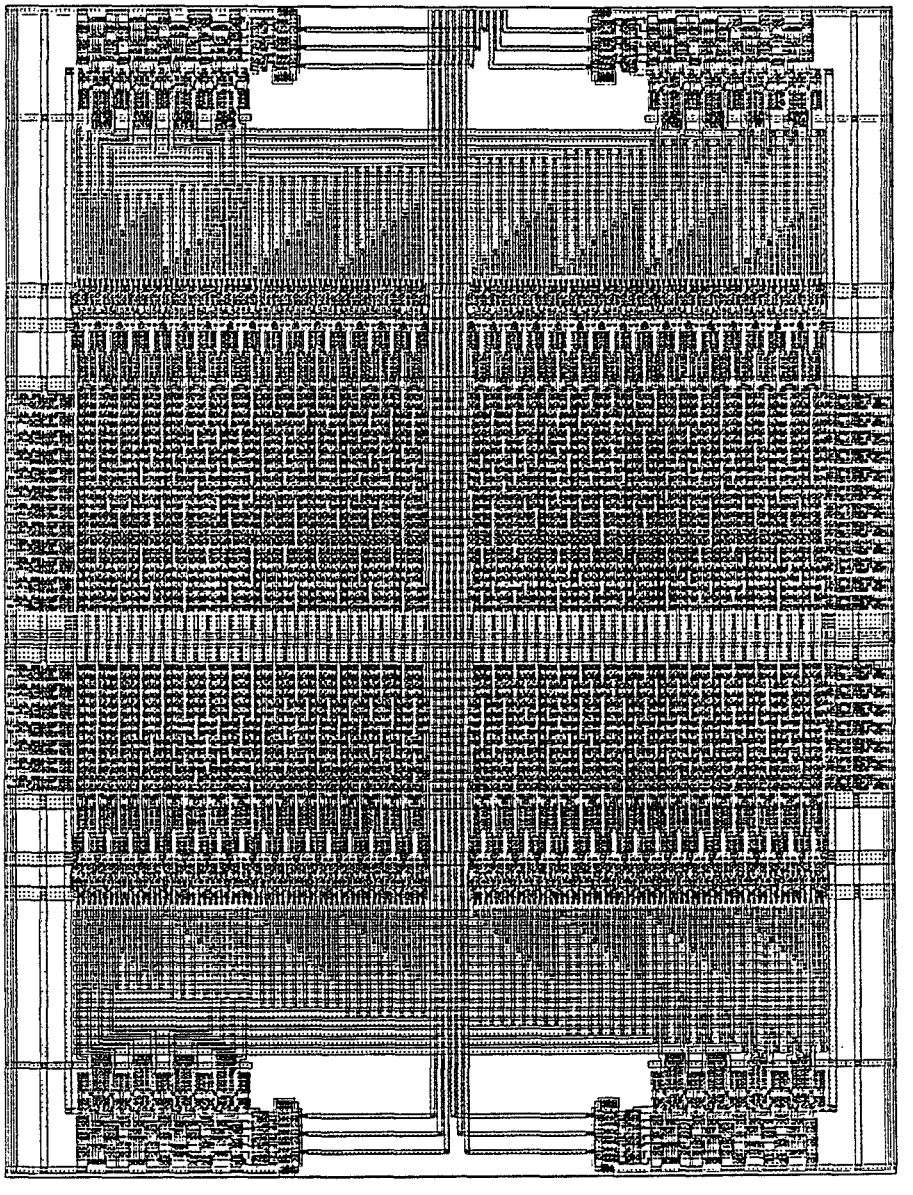


Figure 2.3: Physical layout of the distributed array multiplier for 12-bit numbers and coefficients. Area of this module is $1292\lambda \times 1646\lambda$.

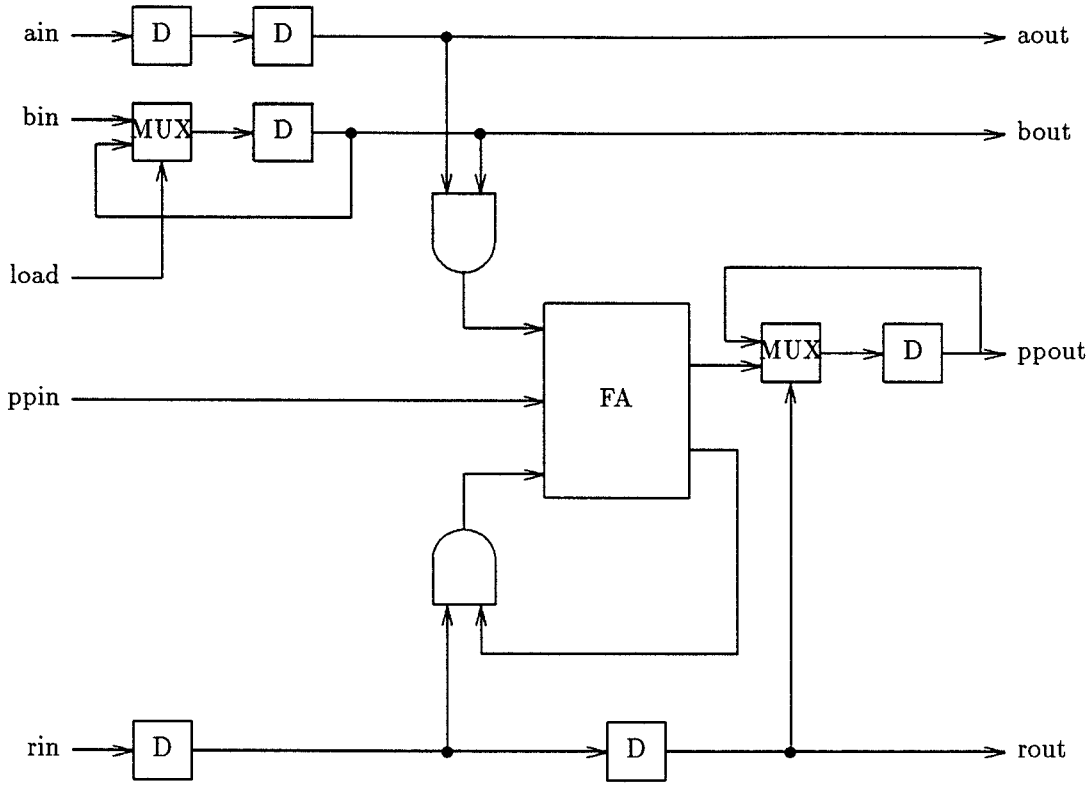


Figure 2.4: Logic diagram of the bit-serial multiplier cell.

"ain" input. The product is available at the "ppout" output. Fig. 2.4 shows the block diagram of each cell. Each cell consists of a full adder and latches to store the carry and sum bits. The sum is propagated to the neighboring cell and the carry is used in the next computations performed in the cell. Two's complement multiplication is achieved by inverting the partial product input to the last cell of the array. A separate control signal "rin" is propagated through the linear array to indicate the start of the input number stream. Fig. 2.5 shows the layout of this multiplier.

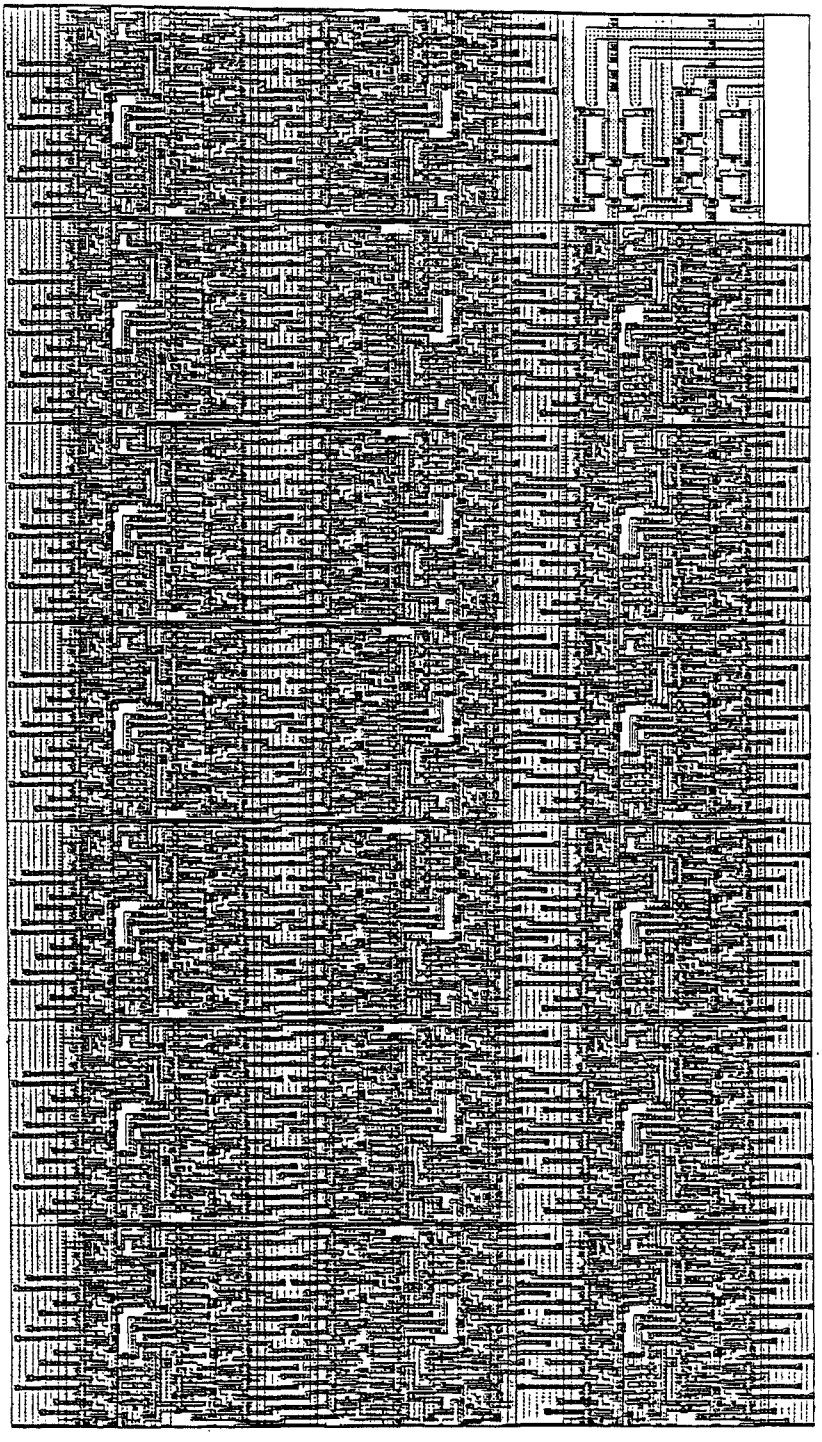


Figure 2.5: Physical layout of the bit-serial multiplier for 20-bit numbers and coefficients. Area of this module is $936\lambda \times 1680\lambda$.

2.2 Data Conversion Hardware

Input data is usually available in line-scan mode. Images are scanned pixel by pixel from left to right and top to bottom. Image compression algorithms and architectures usually operate on blocks of data at a time. In this section, we present a simple architecture for converting line-scan mode data into the format required by these architectures.

If the blocksize is $k \times k$, k adjacent lines of the input image need to be stored. This is the minimum necessary for converting line-scanned data into block-scanned mode. Fig. 2.6 shows the design of this module. In this scheme, k lines of a line-scanned input image are stored in shift registers and transposed into block-scan mode. An 2-D array of $\frac{N}{k} \times k$ shift registers, each of size k, are used to hold these k lines. Multiplexers are used to switch the output of register $R_{i,j}$ between $R_{i-1,j}$ and $R_{i,j-1}$. Initially, the outputs of register $R_{i,j}$ are connected to the inputs of register $R_{i-1,j}$, $2 \leq i \leq \frac{N}{k}$. The outputs of register $R_{1,j}$ are connected to the inputs of register $R_{\frac{N}{k},j-1}$. This configuration is shown with solid arrows in Fig. 2.6(a) and is called the horizontal mode of operation. Once the registers are filled with k lines of input, global control signals are used to switch the multiplexers in every register. Now, the outputs of register $R_{i,j}$ are connected to the inputs of register $R_{i,j-1}$. This configuration is shown with dashed arrows in Fig. 2.6(a) and is called the vertical mode of operation. During the next k^2 clock cycles, data is flushed out of these registers in block-scan mode. The last row of cells is treated differently. It is switched back from vertical to horizontal mode after k clock cycles and stays in the horizontal mode of operation for $Nk - k$ clock cycles. The remaining rows stay in the vertical

mode of operation for k^2 clock cycles and in the horizontal mode of operation for $Nk - k^2$ clock cycles. Using this procedure, the next set of k lines can be applied to this module while the current set of k lines are being flushed out.

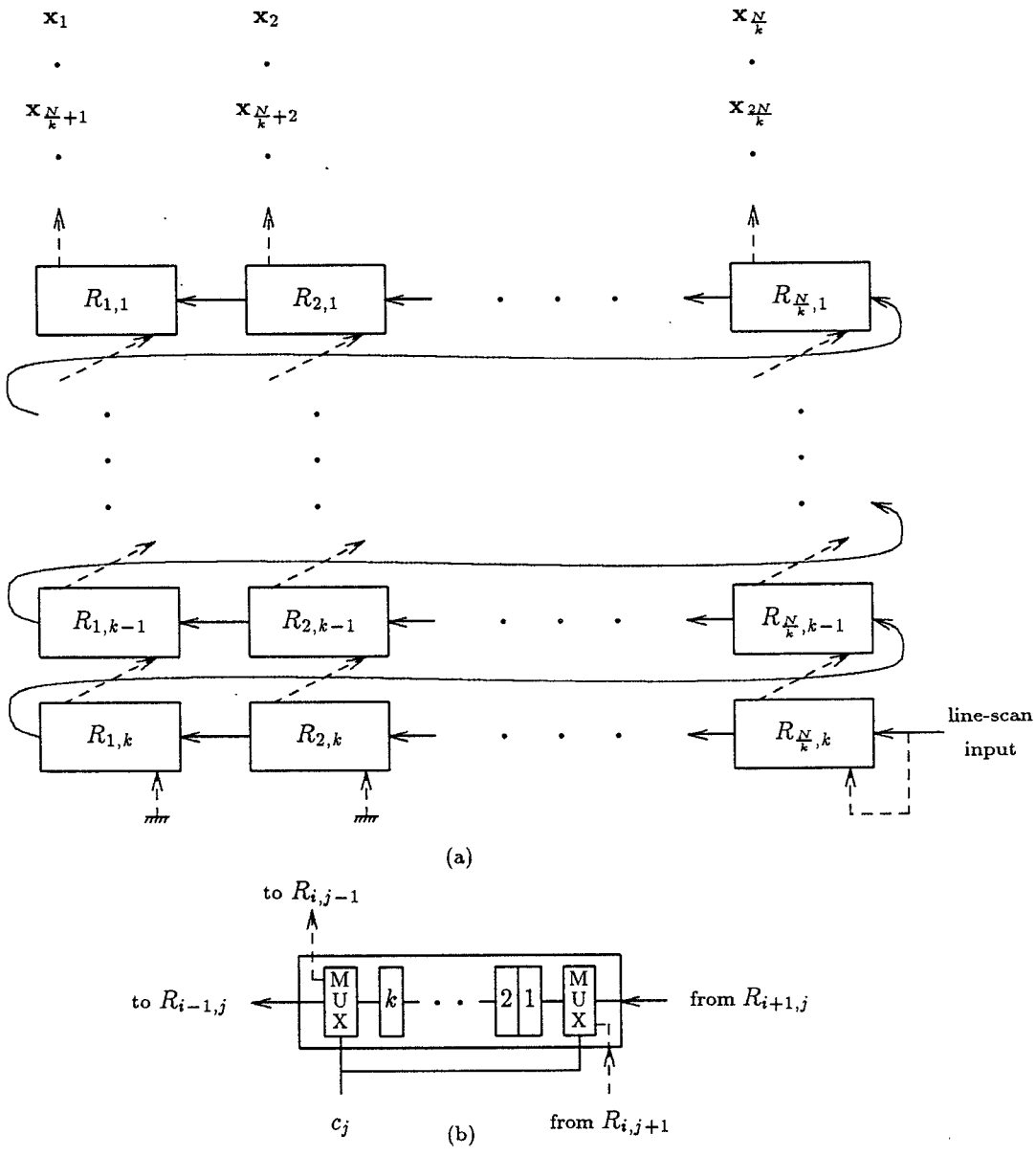


Figure 2.6: Architecture to convert line-scan image data into block-scan mode. (a) 2-D array of cells. (b) Detail of cell $R_{i,j}$ with k latches and two multiplexers. Control signal c_j switches the cell between horizontal and vertical modes of operation.

Chapter 3

Scalar and Inverse Scalar Quantizers

Scalar Quantization (SQ) is a technique of representing an analog or digital signal with a lower precision digital approximate. This results in lossy data compression of input signals. If the input is a sequence of symbols, each input symbol is quantized independently [4, 13].

3.1 Definition

A SQ is completely defined by its set of quantization and reconstruction levels [11]. Let $x_0, x_1, \ldots, x_N$ be the set of $N+1$ decision levels and $y_1, y_2, \ldots, y_N$ be the set of N reconstruction or output levels. These points are ordered such that,

$$x_0 < y_1 < x_1 < y_2 < \cdots < y_N < x_N.$$

The input-output characteristic of this SQ is a staircase function as shown in Fig. 3.1.

If the input signal x, lies between x_{i-1} and x_i, the SQ encoder represents it with the index i. The number of bits needed to uniquely specify the index i is $\log_2 N$. This index i is stored or transmitted instead of the original signal x. If

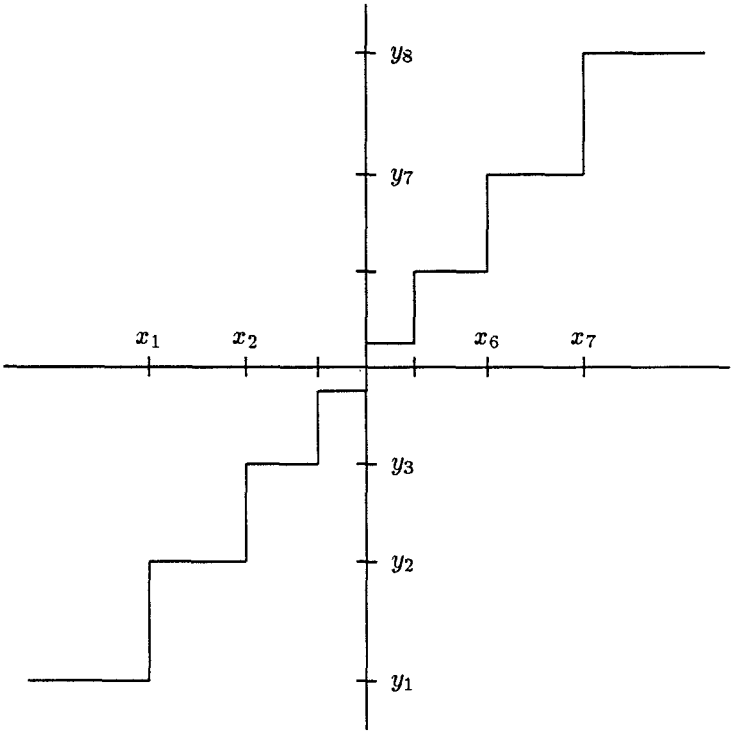


Figure 3.1: Input-Output characteristic of a SQ with $N = 8$.

the input x is an analog signal, it is first quantized using an Analog to Digital (A/D) converter and the digital output of the A/D converter is then compressed using the SQ described above.

An SQ decoder, or an Inverse Scalar Quantizer (ISQ), transforms index i into the reconstruction level y_i. The difference between the original signal x and its reconstruction level y_i, is the quantization error.

The necessary conditions for optimality in the mean squared error sense are:

- **Nearest Neighbor Condition:** For a given ISQ, the SQ is a nearest neighbor mapping. Given the reconstruction levels $y_1, y_2, \ldots, y_N$, the optimal values for the decision levels are

$$x_i = \frac{(y_i + y_{i+1})}{2}.$$

- **Centroid Condition:** For a given set of decision levels $x_0, x_1, \ldots, x_N$, the optimal reconstruction level y_i is the centroid of the interval (x_{i-1}, x_i) with respect to the input probability density.

Optimal SQ and ISQ can be designed based on the statistics of the input signal using the Lloyd-Max algorithm [49, 50].

3.2 Transform Coding

In a transform coding system an input image or speech signal block is first transformed using a linear transformation, such as the 2-D Discrete Cosine Transformation (DCT), to remove inter-pixel correlation within the block. The transform coefficients are then quantized using scalar quantizers. In general each transform

coefficient is assigned a different scalar quantizer for optimum performance. The total number of index bits for the block is divided among the different scalar quantizers depending on the statistics of the associated transform coefficients.

3.3 Architectures

In this section we discuss various architectures for the implementation of Scalar and Inverse Scalar Quantizers. We consider both an individual SQ and a set of SQs for a transform coding system.

Given an input sample x and a set of decision levels $x_0, x_1, \ldots, x_N$, a SQ returns the index of the interval the input lies in. A straightforward way is to compare the input sample with each of the N decision levels. This can be done either serially using one comparator or in parallel using N comparators. The parallel version is shown if Fig. 3.2. The architecture consists of a linear array of comparators. Comparator C_i compares the input x with the decision level x_i and returns either a "zero" or a "one" depending on which is bigger. These outputs are applied to a linear array of AND gates. If the output of AND gate A_i is a "1", then the input lies in interval "i".

For large values of N, the chip area required for N comparators becomes prohibitive. SQ can also be implemented as a table lookup operation using either ROMs or PLAs. The input signal x is applied as the input to the decoder circuitry and the index of the correct decision level is returned by the ROM. The chip area required for table lookup depends on the statistics of the input image and the size of N. For large values of N, this approach may use less area than the comparator scheme.

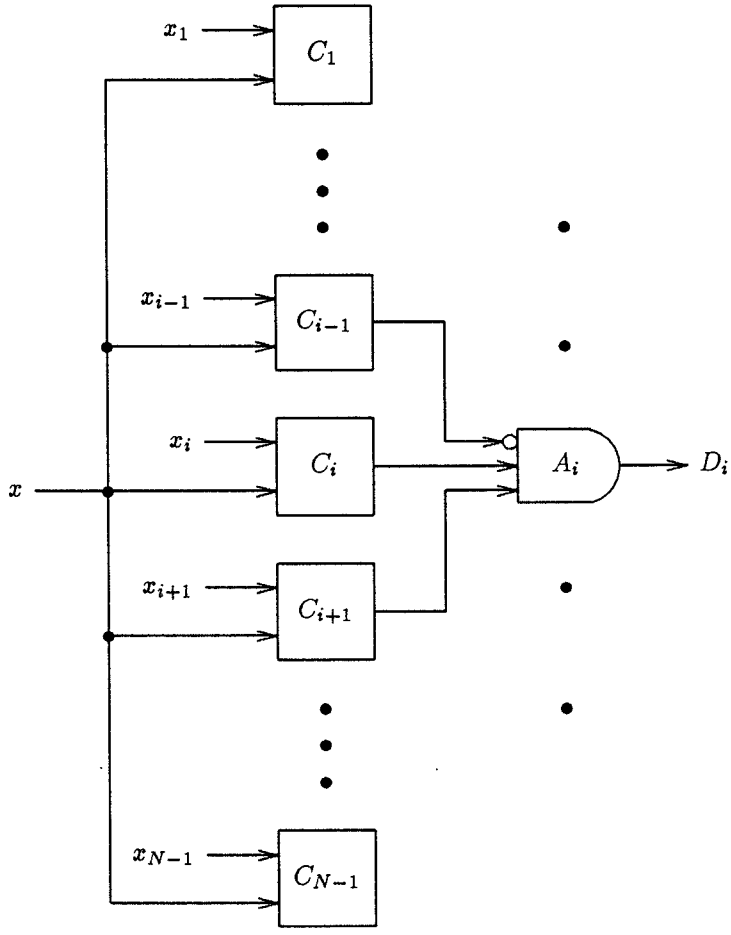


Figure 3.2: Scalar Quantizer block diagram.

3.4 VLSI Implementation

In this section we describe the design and implementation of SQ and ISQ chips for an adaptive image compression system [51]. In this system, the input image is segmented into blocks each of size 8×8 pixels and each block is transformed using a 2-D DCT chip. The transform coefficients are applied to the SQ chip for quantization. The index bits are transmitted over a satellite channel. The receiver consists of an ISQ followed by an Inverse 2-D DCT chip.

We implemented both the SQ and ISQ chips using the table lookup procedure described in the previous section. Both chips were fabricated using MOSIS $2\mu m$ N-well fabrication process. Figs. 3.3 and 3.4 show plots of the fabricated SQ and ISQ chips respectively. Both chips have been exhaustively tested at 20 MHz.

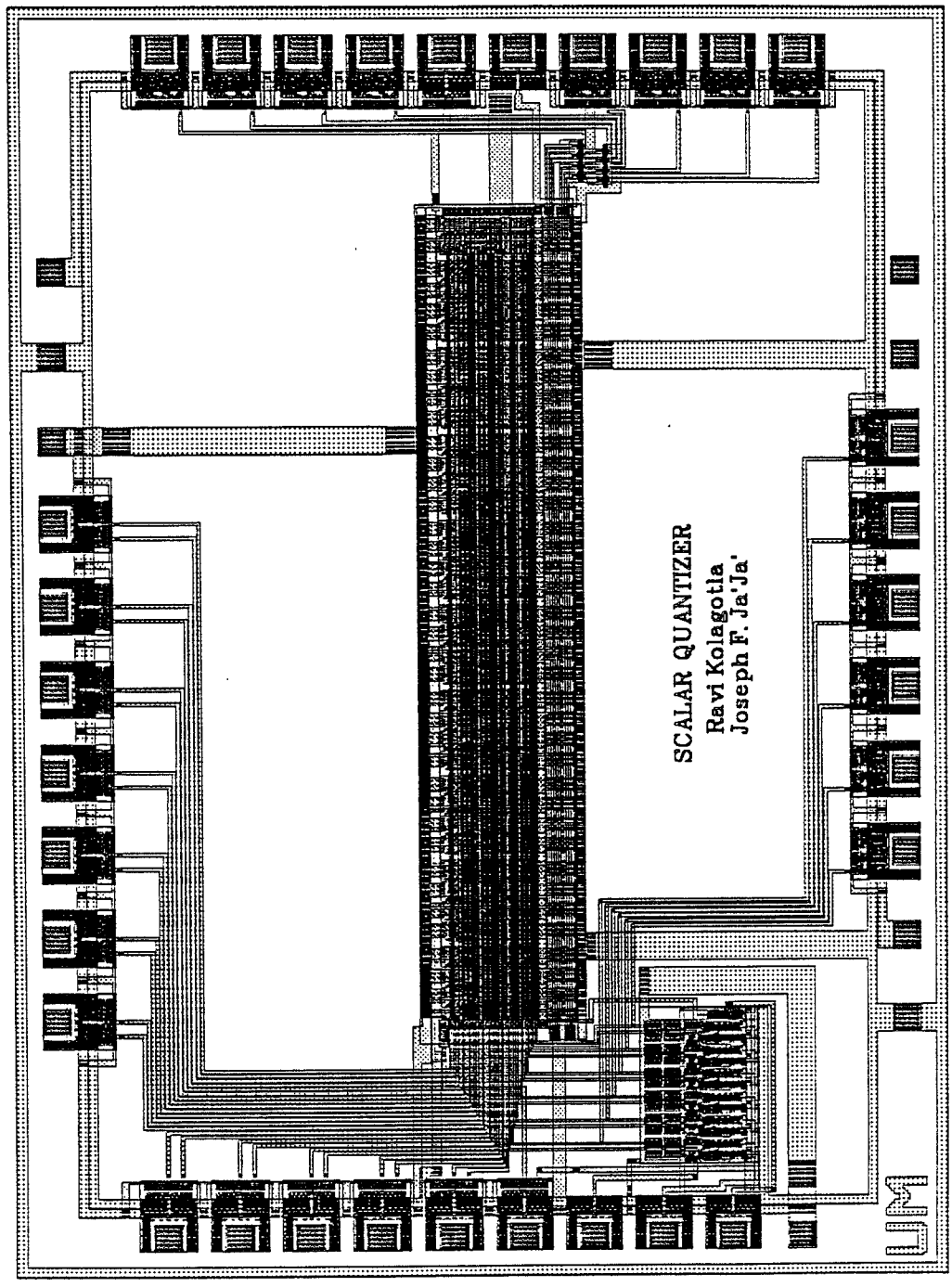


Figure 3.3: Photograph of the SQ chip. Die size is $4.6mm \times 6.8mm$.

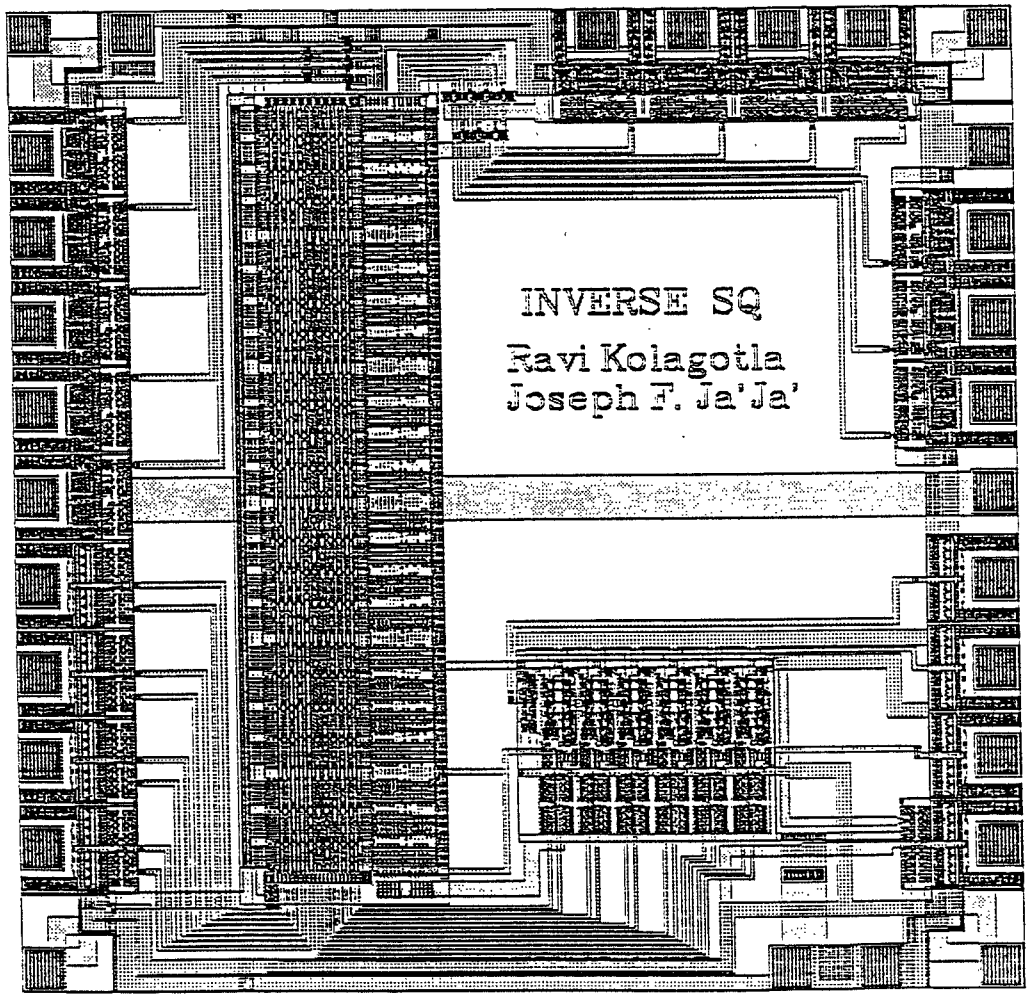


Figure 3.4: Photograph of the ISQ chip. Die size is $2.22mm \times 2.25mm$.

Chapter 4

Tree Search Vector Quantizers

A Tree-Search 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.

4.1 Definition

At each stage of a binary TSVQ, the input vector is compared with two code-vectors. 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:

$$d(\mathbf{x}, \mathbf{c_1}) \geq d(\mathbf{x}, \mathbf{c_2}), \tag{4.1}$$

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 (4.1) can be expressed as:

$$(\mathbf{x} - \mathbf{c_1})^T \mathbf{W} (\mathbf{x} - \mathbf{c_1}) - (\mathbf{x} - \mathbf{c_2})^T \mathbf{W} (\mathbf{x} - \mathbf{c_2}) \quad \geq 0 \tag{4.2}$$

If equation (4.2) 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 (4.2) to obtain [15]:

$$\sum_{j=1}^{L} \{\alpha_j x_j\} + \beta \quad \geq 0 \tag{4.3}$$

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 α and β 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 (4.3) is still valid in this case, but a preprocessor is needed to compute the α and β coefficients in real-time. The same simplification can be derived for the case of the Itakura-Saito distortion measure as well [26].

This algorithm is based on Binary Hyperplane Testing [25]. Directly implementing equation (4.1) requires $2(L^2 + L)$ multiplications, $2(L^2 - 1)$ additions and $L^2 + L$ words of memory storage, while implementing equation (4.3) requires only L multiplications, L additions, and $L + 1$ words of memory storage.

4.2 Single Node Processor

The Single Node Processor (SNP) performs the computations stated in equation (4.3). Its output is a '0' if equation (4.3) is satisfied and a '1' otherwise. The SNP contains of a parallel multiplier [41] 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. 4.1 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 α_j and x_j, and a $2b$-bit number β', 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 β. 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. 4.2. Each cell consists of a full adder and three latches. Carry is propagated to the neighboring cell and sum is stored within

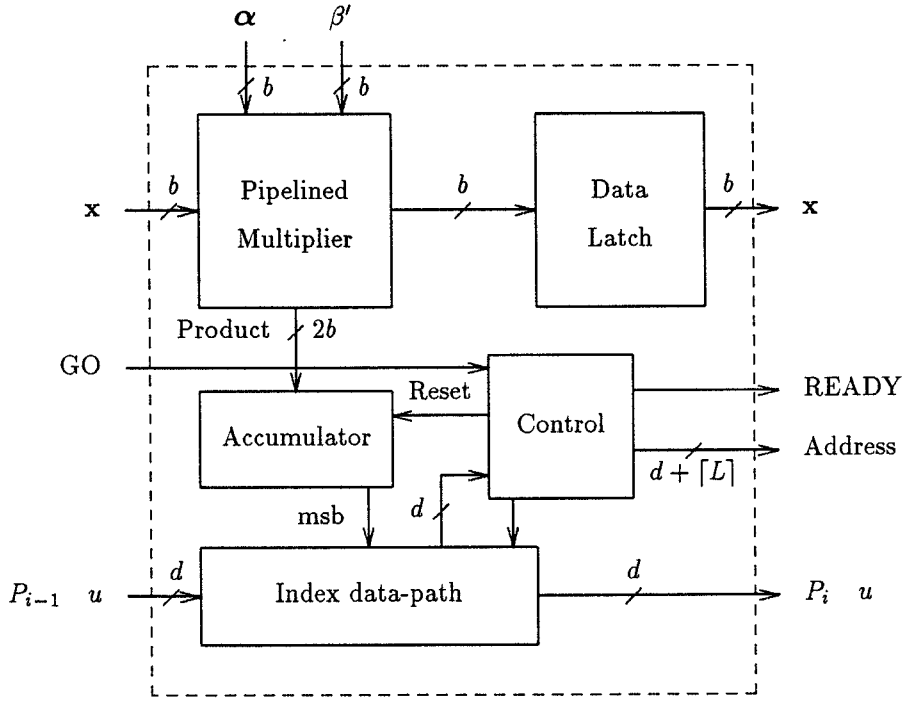


Figure 4.1: Detailed block diagram of each processor. Each processor's READY output must be connected to the GO input of its neighbor. Only the most significant b bits of β' are applied to the processor. The least significant b bits are set to zero internally.

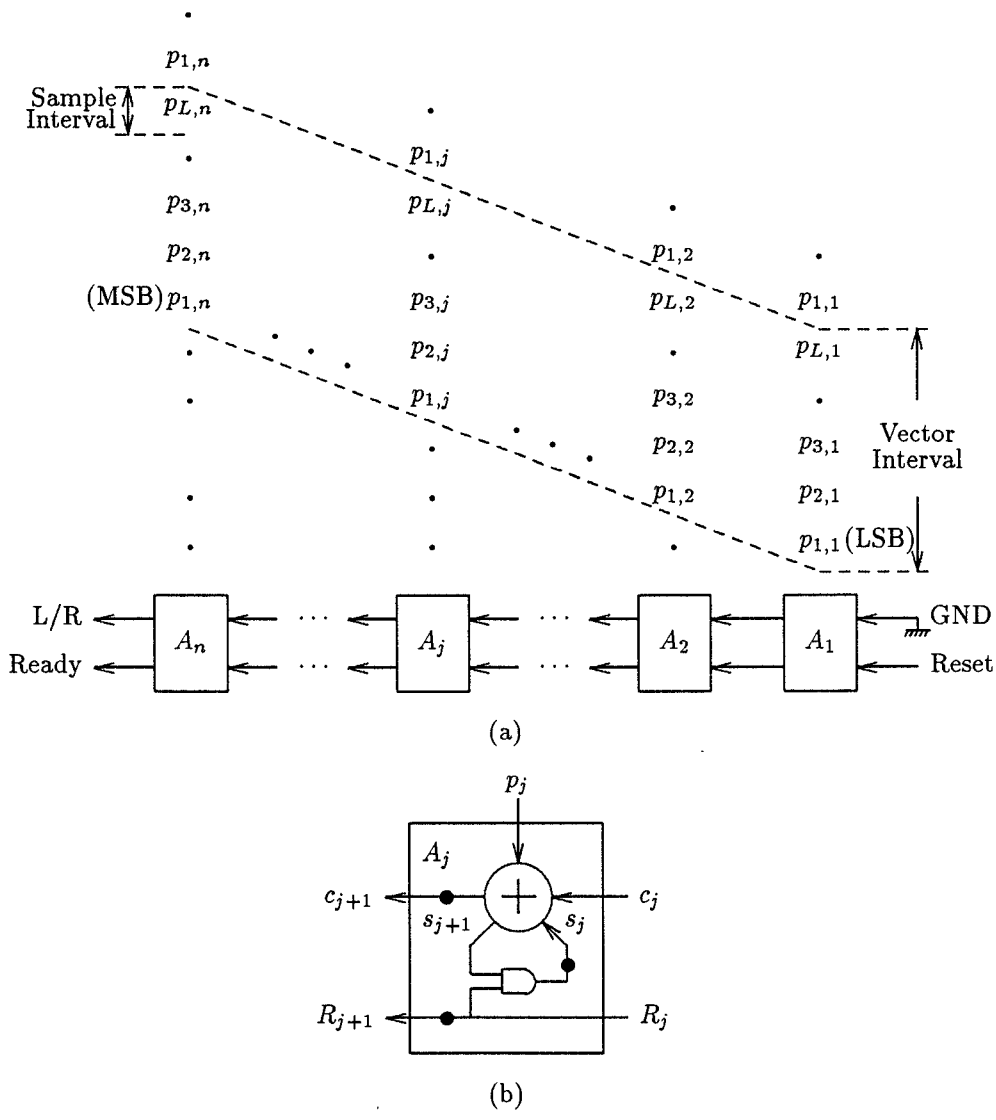


Figure 4.2: Detailed diagram of the accumulator (a) Linear array of cells. Input data is applied in a skewed fashion and carry is propagated between cells. Reset is applied to the first cell and is propagated down the array. Cells in this array are reset in a staggered fashion. (b) Detail of each cell. 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. 4.2. 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.

4.3 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. 4.3 shows the architecture of a TSVQ using d Single Node Processors (SNPs). The coefficients necessary for each SNP's computations are stored in

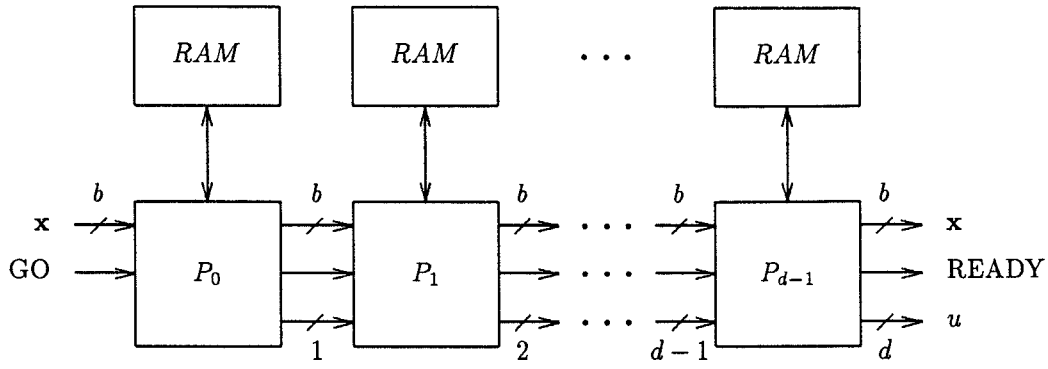


Figure 4.3: Systolic architecture for computing TSVQ. Each SNP adds its partial index to the index data-path, and generates a control signal to initiate processing by its neighbor down the tree. No global control signals are needed.

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 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 β' and the L components of α_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. 4.4. 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,

$$L_{TSVQ} = dL_{SNP} = d(b+n+L). \tag{4.4}$$

The throughput in this case is one L-dimensional vector per L_{TSVQ} clock cycles.

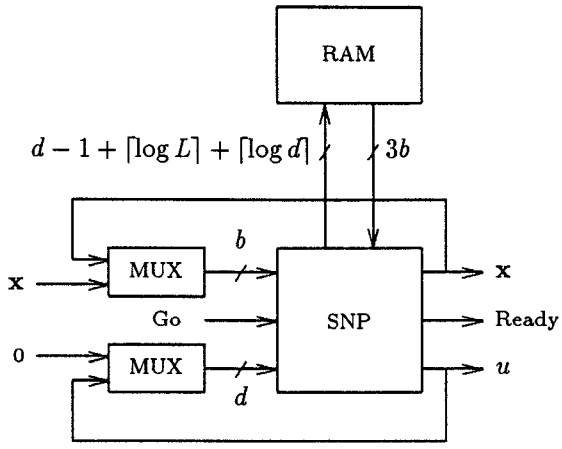


Figure 4.4: TSVQ architecture using one SNP and recirculating registers. Input vector **x** must be recirculated d times, once for each level of the binary tree.

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.

4.4 VLSI Implementation

The detailed block diagram of each processor is shown in Fig. 4.1. 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 (4.3) 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 4×4 or 8×8 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.

4.5 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. 4.5 illustrates the timing of all local signals between processors for the case when the block size is 8×8. 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.

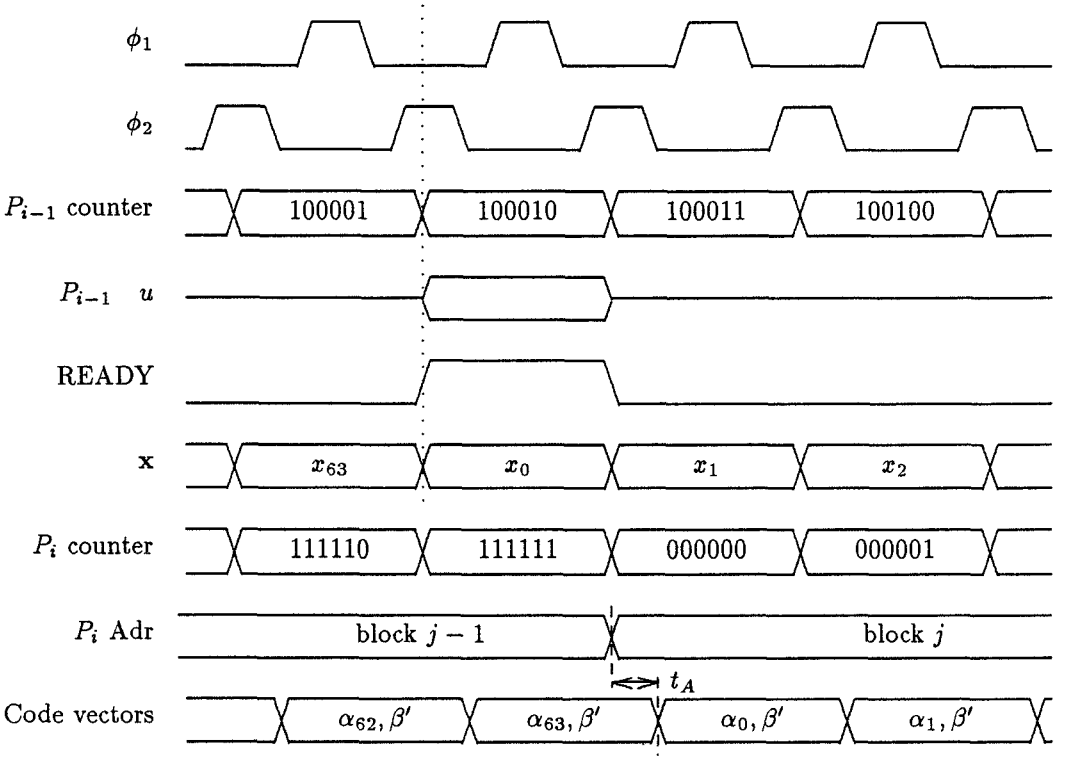


Figure 4.5: Timing diagram of signal flow between processors for input block size of 8×8. Dotted line shows the boundary between adjacent vectors. Coefficient memory chips must have an access time smaller that t_A.

This includes the 64 cycles needed to read each block. If the block size is 4×4, 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 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.

4.6 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 [30]. Each processor contains 25,000 transistors and has 84 pins. The processors have been tested at 20 MHz. These processors can operate on either 4×4 or 8×8 blocksizes. Fig. 4.6 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.

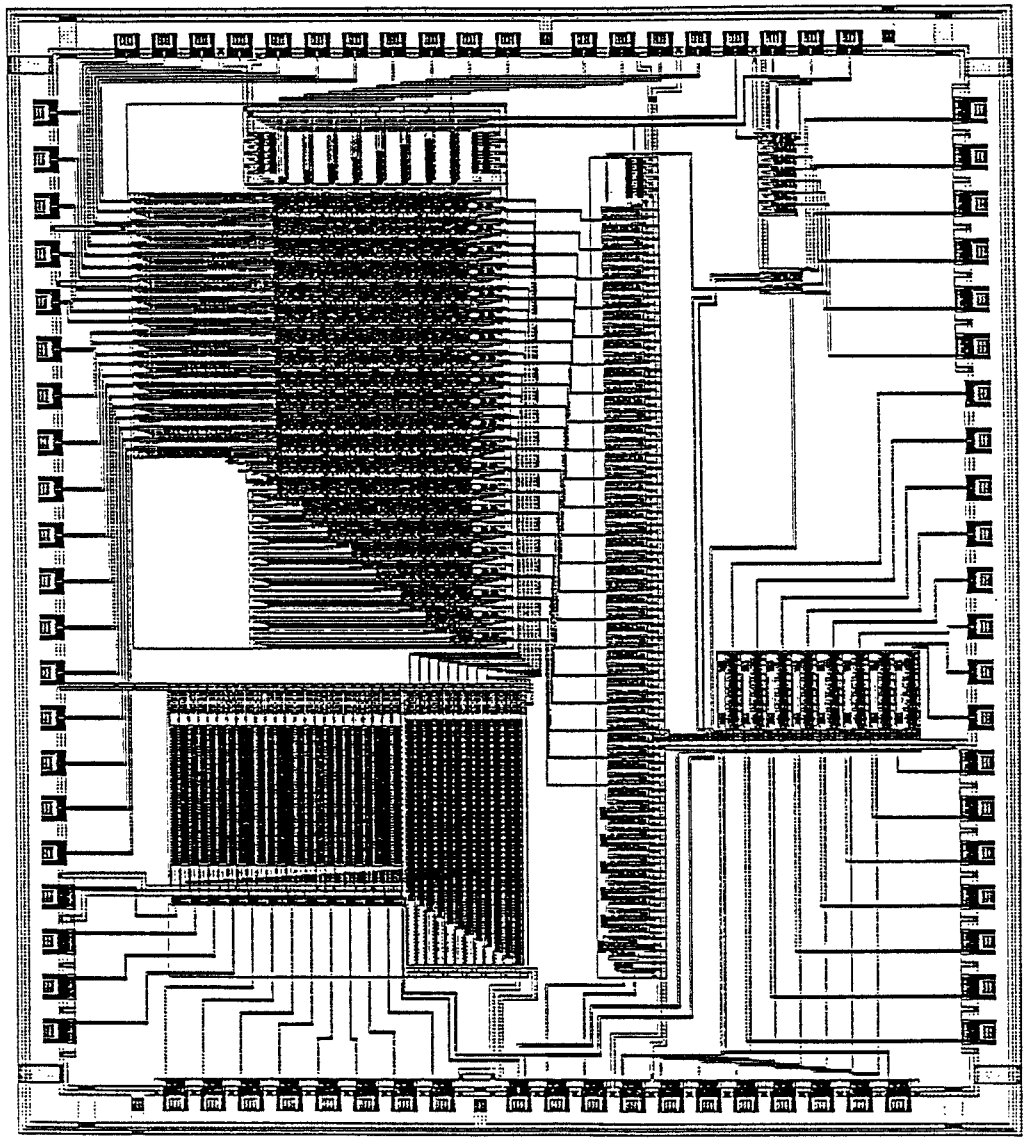


Figure 4.6: Plot of the TSVQ processor chip. Die size is $7.9mm \times 9.2mm$.

Chapter 5

Finite State Vector Quantizers

Finite State VQ (FSVQ) is a class of VQ with memory which makes use of correlation between neighboring vectors to improve performance for a given vector dimension and bit-rate [31]. An FSVQ consists of a super-codebook which contains a large number of codevectors and a finite state machine. Each state in an FSVQ represents a small region of the super-codebook. The current state of the FSVQ encoder depends on past outputs. By restricting the codebook search space to a small region for each input vector, FSVQ achieves the performance of a large rate codebook at a relatively small rate [32].

5.1 Definition

Given an input vector $\mathbf{x}$, a VQ encoder chooses a reproduction vector $\hat{\mathbf{x}}$ from a predetermined set of reproduction vectors (or codevectors) that is closest to the input vector relative to a certain distortion measure. The input vector is then represented by the index u of codevector $\hat{\mathbf{x}}$. This index, also known as the channel symbol, is transmitted to the decoder. A VQ decoder maps this channel

symbol onto its corresponding reproduction vector from the codebook.

In a finite state VQ (FSVQ), performance is improved by exploiting the correlation between neighboring vectors. An L-dimensional K-state FSVQ is specified by a state space $\mathcal{S} = \{1, 2, \ldots, K\}$, an initial state s_0, and three mappings [31]:

(1) $\alpha : \mathbb{R}^L \times \mathcal{S} \rightarrow \mathcal{N}$: finite-state encoder,

(2) $\beta : \mathcal{N} \times \mathcal{S} \rightarrow \hat{\mathcal{A}}$: finite-state decoder,

(3) $f : \mathcal{N} \times \mathcal{S} \rightarrow \mathcal{S}$: next state function.

Here, $\mathcal{N} \triangleq \{1, 2, \ldots, N\}$ is the finite channel alphabet of size N and $\hat{\mathcal{A}}$ is the reproduction space. Given a sequence of L-dimensional input vectors $\{\mathbf{x}_n\}$, the FSVQ encoder determines the sequence of reproduction vectors $\{\hat{\mathbf{x}}_n\}$, the sequence of channel symbols $\{u_n\}$, and the sequence of states $\{s_n\}$ according to:

$$u_n = \alpha(\mathbf{x}_n, s_n), \quad n = 0, 1, \ldots,$$

$$\hat{\mathbf{x}}_n = \beta(u_n, s_n), \quad n = 0, 1, \ldots,$$

$$s_{n+1} = f(u_n, s_n), \quad n = 0, 1, \ldots.$$

Given the initial state and the channel symbol sequence, the FSVQ decoder can track the state sequence, because the next state depends only on the present state and the output channel symbol. The set of reproduction vectors $\mathcal{C}_l \triangleq \{\beta(u, l), u \in \mathcal{N}\}$, is the codebook associated with state l; obviously, $\hat{\mathcal{A}} = \bigcup_{l=1}^{K} \mathcal{C}_l$.

An FSVQ can be interpreted as a set of K VQs, one VQ associated with each state, and a finite state machine which selects one of these VQs to encode the given input vector. Similarly, an FSTSVQ can be interpreted as a set of K TSVQs.

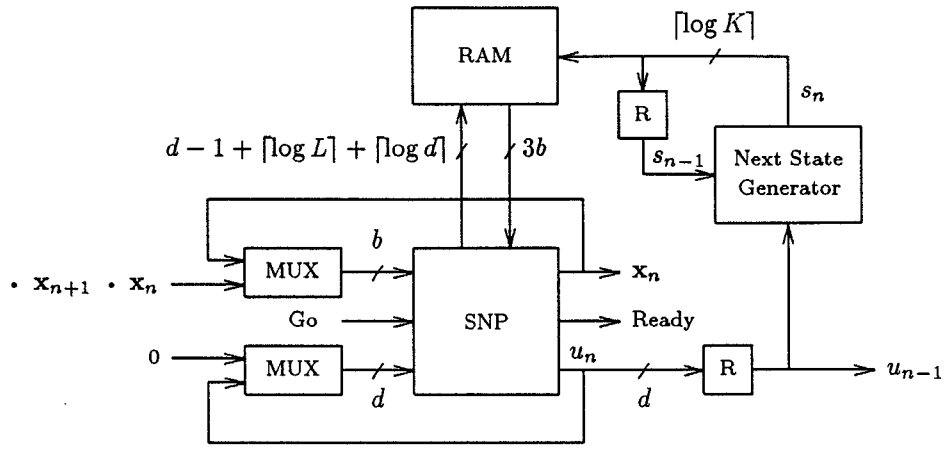


Figure 5.1: Block diagram of the FSTSVQ processor. R is a unit delay element. The $\lceil \log K \rceil$ bits of s_n are used as address bits for the RAM.

5.2 Basic FSTSVQ architecture

Given a sequence of L-dimensional input vectors $\{\mathbf{x}_n\}$, we are required to generate a sequence of channel symbols $\{u_n\}$, and a sequence of states $\{s_n\}$. Since the next state of the FSTSVQ depends on the present state and the current output channel symbol, the encoding of vector $\mathbf{x}_{n+1}$ cannot start until the encoding of vector $\mathbf{x}_n$ is complete. One TSVQ processor, together with additional hardware for generating the next state information, is sufficient to build an FSTSVQ.

Fig. 5.1 shows a block diagram of the FSTSVQ processor. The next state function module determines s_n, given u_{n-1} and s_{n-1}, and can be implemented by a simple table lookup using a PLA. This state information is stored in register R and is also used to choose the right codebook for quantizing vector $\mathbf{x}_n$. A memory of total size $2^d(L+1)K$ words is needed. Channel symbols u_n are d bits wide.

5.3 FSTSVQ for speech and image coding

In this section we describe how our FSTSVQ architecture can be used for speech and image coding applications. The general architecture presented above can directly be used for speech coding. For image coding applications, we define a 2-D extension of FSTSVQ and describe a systolic array architecture for efficient hardware implementation.

5.3.1 FSTSVQ for speech coding

In speech coding applications, an L-dimensional vector is formed from a group of L adjacent speech samples. The resulting sequence of vectors $\{\mathbf{x}_n\}$ is quantized to obtain the output sequence of channel symbols $\{u_n\}$. The architecture of the general FSTSVQ described above can directly be used for speech coding applications. Adjacent speech samples are separated by the latency of the FSTSVQ quantizer,

$$L_{FSTSVQ} = L_{TSVQ} + L_{NSG}, \tag{5.1}$$

where L_{NSG} is the latency of the next state generator. The next state generator is implemented as a simple table lookup and $L_{SNP} = 1$ clock cycle. For a tree depth of $d = 4$ and a vector dimension of $L = 8$, $L_{FSTSVQ} = 141$. If speech waveforms are sampled at 8 KHz, this architecture running at a clock speed of 0.14 MHz can quantize them in real-time. If FSTSVQ is used to quantize speech LSP parameters with an update rate of 22.5 msec, a clock speed of less than 5 KHz is required.

5.3.2 FSTSVQ for image coding

In image coding applications, an input image of size $N \times M$ pixels is partitioned into blocks each of size $k \times k$ pixels as shown in Fig. 5.2. Each block is interpreted as a vector of dimension $L = k^2$. A block-scan of the input image frame generates a sequence of L-dimensional vectors $\{\mathbf{x}_n\}_{n=1}^{NM/k^2}$. Unlike the 1-D case, each input vector has more than one adjacent preceding neighbor. For efficient encoding of an input vector $\mathbf{x}_n$, it is essential to exploit its correlation with the adjacent vectors in the north ($\mathbf{x}_{n-\frac{N}{k}}$), west ($\mathbf{x}_{n-1}$), and northwest ($\mathbf{x}_{n-\frac{N}{k}-1}$) directions. In turn, vector $\mathbf{x}_n$ affects the state of the FSTSVQ while quantizing vectors $\mathbf{x}_{n+1}$, $\mathbf{x}_{n+\frac{N}{k}}$, and $\mathbf{x}_{n+\frac{N}{k}+1}$.

The current state $\mathbf{s}_n$ associated with vector $\mathbf{x}_n$ is defined as a three component vector $\mathbf{s}_n = (s_n^W, s_n^{NW}, s_n^N)$, where s_n^W, s_n^{NW}, and s_n^N are the substates associated with vectors $\mathbf{x}_{n-1}$, $\mathbf{x}_{n-\frac{N}{k}-1}$, and $\mathbf{x}_{n-\frac{N}{k}}$ respectively. The next state generator determines these substates according to:

$$s_n^W = f_1(u_{n-1}, \mathbf{s}_{n-1}), \tag{5.2}$$

$$s_n^{NW} = f_2(u_{n-\frac{N}{k}-1}, \mathbf{s}_{n-\frac{N}{k}-1}), \tag{5.3}$$

$$s_n^N = f_3(u_{n-\frac{N}{k}}, \mathbf{s}_{n-\frac{N}{k}}), \tag{5.4}$$

where f_1, f_2, and f_3 are the three next state functions.

Equations (5.2), (5.3), and (5.4) suggest the data dependency graph shown in Fig. 5.3(a). Each node in this figure represents the quantization of a vector that corresponds to a block of the image. The arcs indicate precedence constraints between various quantizations. Computations that can be performed concurrently are shown between dashed lines. Projecting this graph in the vertical direction leads to a simple linear array structure as shown in Fig. 5.3(b).

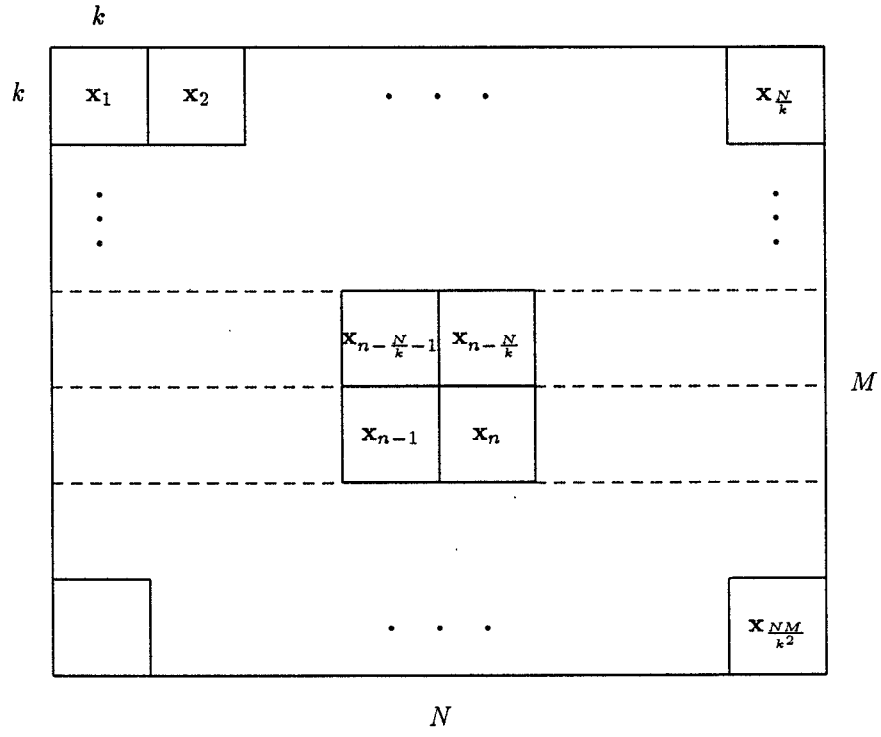


Figure 5.2: Block-scan of an input image of size $N \times M$. The internal state of the FSVQ while quantizing vector $\mathbf{x}_n$ depends on the quantized outputs of vectors $\mathbf{x}_{n-1}$, $\mathbf{x}_{n-\frac{N}{k}}$, and $\mathbf{x}_{n-\frac{N}{k}-1}$.

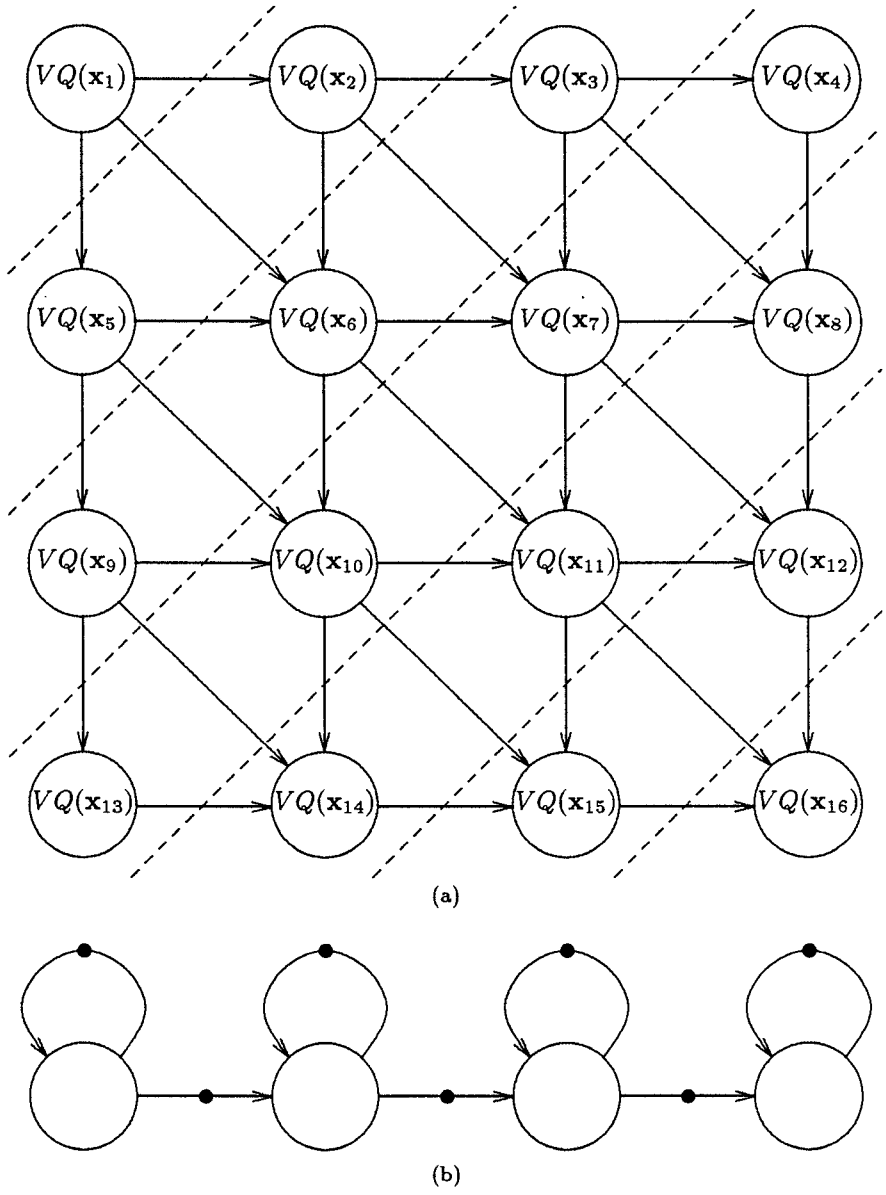


Figure 5.3: (a) Dependency graph of the FSTSVQ for image coding for an image of size $N = M = 4k$. Each circle represents the quantization of one input vector. (b) Projection in the vertical direction. Solid circles are unit delay elements.

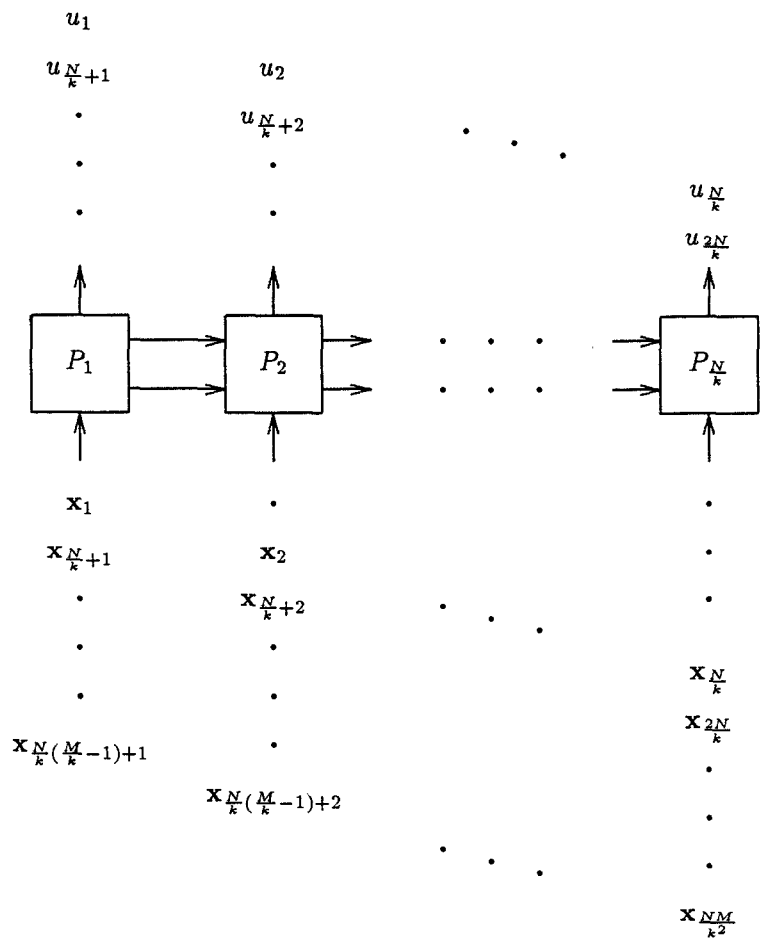


Figure 5.4: Systolic architecture using $\frac{N}{k}$ processors arranged as a linear array. Input data blocks are applied to this array in a skewed fashion.

Consider a linear array of $\frac{N}{k}$ processors as shown in Fig. 5.4. The $\frac{N}{k}$ blocks of a given row are applied to these $\frac{N}{k}$ processors in a skewed fashion. Each processor quantizes the blocks in its column, one at a time. Processor P_n quantizes $\mathbf{x}_n$, $\mathbf{x}_{n+\frac{N}{k}}$, $\mathbf{x}_{n+\frac{2N}{k}}$ and so on.

Fig. 5.5 shows the detailed block diagram of each processor. The TSVQ unit consists of a SNP and recirculating registers. The next state generator uses the previous state $\mathbf{s}_{n-\frac{N}{k}}$ and channel symbol $u_{n-\frac{N}{k}}$ to generate the partial substates

$s^W_{n-\frac{N}{k}+1}$, s^{NW}_{n+1}, and s^N_n. Substates $s^W_{n-\frac{N}{k}+1}$ and s^{NW}_{n+1} are propagated to processor P_{n+1}. Substate s^N_n is used, together with s^W_n and s^{NW}_n from processor P_{n-1}, to generate state $\mathbf{s}_n$. The total number of states is K^3 and the memory size is $2^d(L+1)K^3$ words.

For certain applications, the correlation in the northwest direction is ignored to simplify the codebook design process. The current state is then defined as a two component vector $\mathbf{s}_n = (s^W_n, s^N_n)$, and the total number of states is K^2. The architecture presented above can be used for these applications by modifying the next state generator and ignoring the s^{NW}_n signal path in Fig. 5.5. The memory size requirement reduces to $2^d(L+1)K^2$ words.

5.4 Improvements to the FSTSVQ architecture

The systolic architecture presented above uses $\frac{N}{k}$ processors. The latency of the FSTSVQ unit for quantizing one input vector is L_{FSTSVQ} as defined in equation (5.1). Since input data is usually available in line-scan mode, vector $\mathbf{x}_{n+\frac{N}{k}}$ is separated from vector $\mathbf{x}_n$ by Nk clock cycles. Hence, each processor is idle for $Nk - L_{FSTSVQ}$ clock cycles in this scheme. This idle time can be reduced if each processor is used to process $\lfloor \frac{Nk}{L_{FSTSVQ}} \rfloor$ adjacent blocks. Hence, only $\frac{N}{k} / \lfloor \frac{Nk}{L_{FSTSVQ}} \rfloor = \lceil \frac{L_{FSTSVQ}}{k^2} \rceil$ processors are needed in the linear array. For a blocksize of 4×4 pixels and a tree depth of $d = 8$, 22 processors are needed in the linear array. Note that the number of processors is independent of the size of the image. It depends only on the blocksize and the latency of the FSTSVQ processors.

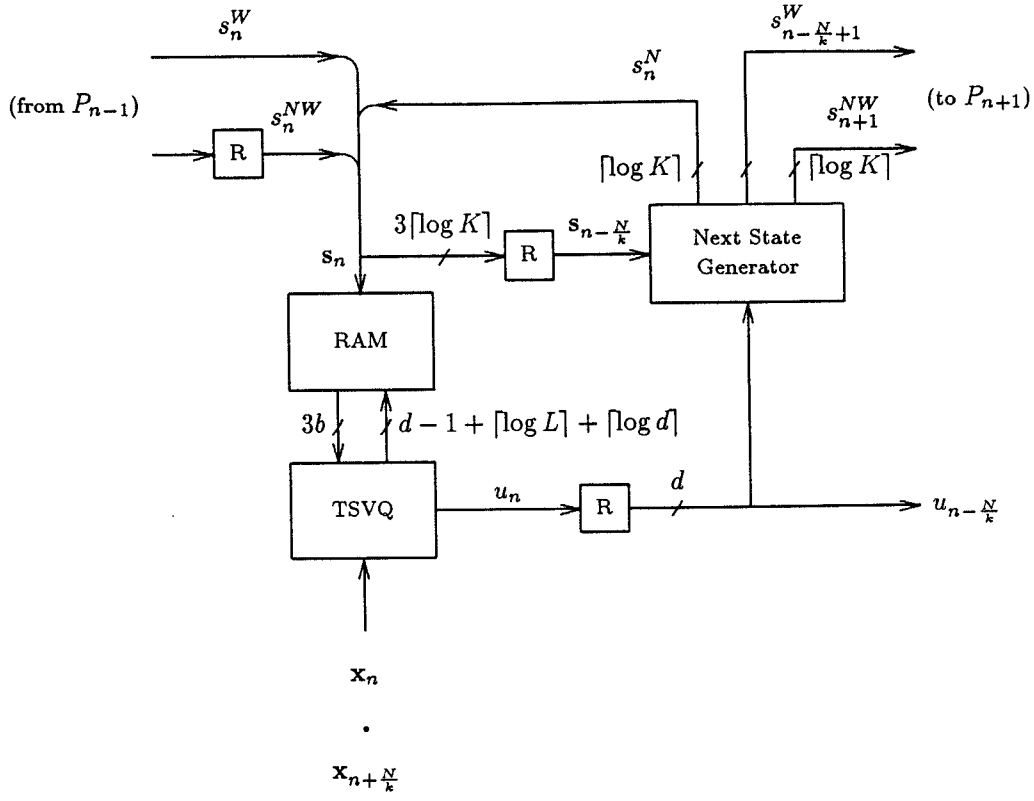


Figure 5.5: Detailed block diagram of processor P_n. The current state $\mathbf{s}_n = (s_n^W, s_n^{NW}, s_n^N)$ is composed of partial substates from the west and northwest (from the previous processor), and the north (from previous computations in this processor).

The d bits of the channel symbol u_n are computed sequentially. The next state generator determines the partial next states for the neighboring vectors based on this symbol and the current state. If the next state functions are restricted to depend only on the first few bits of the channel symbol, their computations can be partially overlapped with those of the SNP processor. This pipelining increases the throughput of this architecture.

Chapter 6

Conclusions

Data compression is an indispensable tool used to make efficient use of available channel capacity and storage resources. Real-time implementations of data compression algorithms make compression feasible for video signal processing and other high speed data communications of the future.

In this dissertation, we have presented efficient systolic architectures for the real-time implementation of Tree-Search and Finite-State Vector Quantizers. 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 8×8 pixels. These processors have been fabricated using $2\mu m$ N-well. The processor chips have been thoroughly tested and found to be fully functional at a frequency of 20 MHz. Fabrication at the state of the art $0.6\mu m$ technology will result in a 70 MHz speed of operation.

We have developed systolic architectures for computing FSTSVQ in real-time on speech and image data. The speech coding architecture uses one processor, and has an average throughput of L/L_{FSTSVQ} samples per clock cycle. An implementation at 0.14 MHz can quantize speech data sampled at 8 KHz in real-

time. The image coding architecture uses a linear array of $\lceil \frac{d(b+n+L)}{k^2} \rceil$ processors and has a throughput of 1 pixel per clock cycle. At 30 frames/sec, the pixel rate for 1024 $\times$ 1024 images is 31.5 Mpixels/sec. An implementation at 31.5 MHz can quantize 1024 $\times$ 1024 size images in real-time. HDTV images have a typical pixel rate of 70 Mpixels/sec. These FSTSVQ architectures fabricated using $0.6\mu m$ technology will be capable of providing the throughput necessary to implement image compression for HDTV applications.

We have also implemented Scalar and Inverse Scalar Quantizer chips for use in transform coding applications.

During the course of designing these architectures, we have developed several basic modules that have been successfully used in other signal processing applications. Our parallel multiplier and accumulator units have been used to develop synchronizer and demodulator chips for FM sideband demodulation. Our bit-serial multiplier designs were used to implement cascadable lattice FIR filters, and our distributed arithmetic multipliers have been successfully used in implementing unique DCT/DST architectures.

Bibliography

[1] M. Cappel, "Viking finds land," *SunWorld*, p. 13, June 1992.

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

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

[6] S. Y. Kung, *VLSI Array Processors.* Prentice Hall, 1988.

[7] C. Chakrabarti, *VLSI Architectures for Real-Time Signal Processing.* PhD thesis, University of Maryland, 1990.

[8] R. J. Offen, *VLSI Image Processing.* McGraw Hill, 1985.

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

[10] J. P. Hayes, *Computer architecture and organization.* McGraw-Hill, 1988.

[11] A. Gersho, "Principles of quantization," *IEEE Trans. Circuits Syst.*, vol. CAS-25, pp. 427–436, July 1978.

[12] A. Gersho, "On the structure of vector quantizers," *IEEE Trans. Infor. Theory*, vol. IT-28, pp. 157–166, Mar. 1982.

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

[14] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantization design," *IEEE Trans. Commun.*, vol. COM-28, pp. 84–95, 1980.

[15] 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.

[16] S. Panchanathan and M. Goldberg, "A systolic array architecture for image coding using adaptive vector quatization," *IEEE Trans. Circuits Syst. Video Tech.*, vol. 1, pp. 222–229, June 1991.

[17] B. Juang and A. Gray, "Multiple stage vector quantization for speech coding," in *Proc. IEEE Int'l. Conf. on Acoustics, Speech and Signal Processing*, pp. 597–600, 1982.

[18] P. A. Ramamoorthy, B. Potu, and T. Tran, "Bit-serial VLSI implementation of vector quantizer for real-time image coding," *IEEE Trans. Circuits Syst.*, vol. CAS-36, pp. 1281–1289, Oct. 1989.

[19] 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.

[20] 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.

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

[22] 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," Tech. Rep. SRC TR 92–48, University of Maryland, 1992.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] R. K. Kolagotla, S.-S. Yu, and J. F. JáJá, "VLSI implementation of a tree searched vector quantizer," *IEEE Trans. Signal Processing*, Apr. 1993.

[31] J. Foster, R. M. Gray, and M. O. Dunham, "Finite-state vector quantization for waveform coding," *IEEE Trans. Infor. Theory*, vol. IT-31, pp. 348–359, May 1985.

[32] Y. Hussain, *Design and Performance Evaluation of a Class of Finite-State Vector Quantizers.* PhD thesis, University of Maryland, 1992.

[33] R. Aravind and A. Gersho, "Image compression based on vector quantization with finite memory," *Optical Engineering*, vol. 26, pp. 570–580, July 1987.

[34] Y. Hussain and N. Farvardin, "Variable-rate finite-state vector quantization of images," in *Proc. IEEE Int'l. Conf. on Acoustics, Speech and Signal Processing*, 1991.

[35] Y. Hussain and N. Farvardin, "Variable-rate finite-state vector quantization and applications to speech and image coding," *IEEE Trans. Signal Processing*, Feb. 1993.

[36] H. H. Shen and R. L. Baker, "A finite state/frame difference interpolative vector quantizer for low rate image sequence coding," in *Proc. IEEE Int'l. Conf. on Acoustics, Speech and Signal Processing*, pp. 1188–1191, 1988.

[37] R. Dianysian and R. Baker, "A VLSI chip set for real time vector quantization of speech sequences," in *Proc IEEE Intl. Symp. Circuits System.*, pp. 221–224, May 1987.

[38] R. K. Kolagotla, S.-S. Yu, and J. F. JáJá, "Systolic architectures for finite-state vector quantization," in *Proc. Int'l Conf. App. Specific Array Processors*, 1992.

[39] R. J. Higgins, *Digital SIgnal Processing in VLSI.* Prentice Hall, 1990.

[40] G. K. Ma and F. J. Taylor, "Multiplier policies for digital signal processing," *IEEE ASSP Magazine*, vol. 7, pp. 6–20, Jan. 1990.

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

[42] S. A. White, "Applications of distributed arithmetic to digital signal processing: A tutorial review," *IEEE ASSP Mag.*, pp. 4–19, July 1989.

[43] M. T. Sun, T. C. Chen, and A. M. Gottlieb, "VLSI implementation of a 16×16 discrete cosine transform," *IEEE Trans. Circuits Syst.*, pp. 610–617, Apr. 1989.

[44] C. T. Chiu, R. K. Kolagotla, K. J. R. Liu, and J. F. JáJá, "VLSI implementation of real-time parallel DCT/DST lattice structures for video communications," in *IEEE Workshop on VLSI Signal Processing*, 1992.

[45] S. G. Smith and P. B. Denyer, *Serial-Data Computation.* Kluwer Academic, 1988.

[46] L. B. Jackson, J. F. Kaiser, and H. S. McDonald, "An approach to the implementation of digital filters," *IEEE Trans. Audio and Electroacoustics*, pp. 413–421, Sept. 1968.

[47] R. F. Lyon, "Two's complement pipeline multipliers," *IEEE Trans. Commun.*, pp. 418–425, Apr. 1976.

[48] G. L. Baldwin, B. L. Morris, D. B. Fraser, and A. R. Tretola, "A modular, high-speed serial pipeline multiplier for digital signal processing," *IEEE J. Solid-State Circuits*, p. 400, June 1978.

[49] S. P. Lloyd, "Least squares quantization in PCM." Unpublished Bell Laboratories Technical Note. Published in the March 1982 special issue on quantization of the *IEEE Trans. Information Theory*, 1957.

[50] J. Max, "Quantizing for minimum distortion," *IRE Trans. Inform. Theory*, vol. IT-6, pp. 7–12, Mar. 1960.

[51] J. F. JáJá and T. Chiang, "A prototype VLSI system for data compression." Final Report, MIPS Project, 1991.