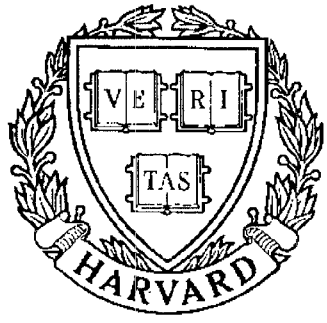
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



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

Systolic Architectures for Finite-State Vector Quantization

by R. Kolagotla, S-S. Yu and J. JáJá

TR 91-102

Systolic Architectures for Finite-State Vector Quantization

Ravi Kolagotla, Shu-Sun Yu and Joseph JáJá

Department of Electrical Engineering,
Systems Research Center, and
Institute for Advanced Computer Studies,
University of Maryland, College Park, MD 20742.

Abstract

We present a new systolic architecture for implementing Finite State Vector Quantization in real-time for both speech and image data. This architecture is modular and has a very simple control flow. Only one processor is needed for speech compression. A linear array of processors is used for image compression; the number of processors needed is independent of the size of the image. We also present a simple architecture for converting line-scanned image data into the format required by this systolic architecture. Image data is processed at a rate of 1 pixel per clock cycle. An implementation at 31.5 MHz can quantize 1024×1024 pixel images at 30 frames/sec in real-time. We describe a VLSI implementation of these FSTSVQ processors.

1 Introduction

Vector Quantization (VQ) is an important compression technique used in speech and image coding applications. VQ provides the best performance among all block structured image coding schemes for a given blocksize and bit-rate. VQ codebooks are typically structured as trees to reduce the codebook search complexity and simplify hardware implementation [1, 2, 3, 4]. Tree Search VQ (TSVQ) has an $O(\log N)$ codebook search complexity compared to the $O(N)$ complexity of VQ for a codebook of size N. 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 [5, 6]. 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 efficiency of a large rate codebook at a relatively small rate [7, 8]. 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 [8].

In this paper, we describe systolic architectures for implementing FSTSVQ in real-time. These architectures can be used in any speech or image compression application based on VQ. In section 2, we provide the general definition of an FSVQ. In section 3, we describe the architecture of a TSVQ and show how an FSTSVQ can be implemented with minimal additional hardware. In section 4, we describe the FSTSVQ architectures for speech and image coding applications. We show how the data dependencies lead to simple systolic array architectures. In section 5, we describe an architecture to convert line-scanned image data into the format required by the FSTSVQ image coding architecture. We describe the VLSI implementation of this scheme in section 6.

2 Definitions

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 Euclidean distance is the most commonly used 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 binary TSVQ, the codebook is organized in a tree structure. The input vector is compared with two codevectors at each node. Based on this comparison, one of the two branches is chosen and the codebook search space is reduced by half. This process is repeated until a leaf node is reached. The

input vector is represented by the index of the leaf codevector at the end of this traversal.

In a finite state VQ (FSVQ), performance is improved by exploiting the correlation between neighboring vectors. An L-dimensional K-state FSVQ [6] is specified by a state space $\mathcal{S} = \{1, 2, \ldots, K\}$, an initial state s_0, and three mappings:

(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.

3 Basic Architectures

In this section, we first describe the design of a Single Node Processor (SNP) which performs the computations required at each node of a binary TSVQ. A TSVQ is implemented as a linear array of SNPs. We will show how to combine TSVQ and a simple next state function generator to implement FSTSVQ.

3.1 Single Node Processor

In a binary TSVQ, the L-dimensional input vector $\mathbf{x} = (x_1, \ldots, x_L)^T$ must be compared with two codevectors at each node. Let $\mathbf{c_1} = (c_{1,1}, \ldots, c_{1,L})^T$, 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:

$$d(\mathbf{x}, \mathbf{c_1}) \geq d(\mathbf{x}, \mathbf{c_2}), \tag{1}$$

where $d(\mathbf{x}, \mathbf{c_i}), \mathbf{i} = 1, 2$ 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 (1) can be expressed as:

$$(\mathbf{x} - \mathbf{c_1})^T \mathbf{W} (\mathbf{x} - \mathbf{c_1}) - (\mathbf{x} - \mathbf{c_2})^T \mathbf{W} (\mathbf{x} - \mathbf{c_2}) \quad \geq 0 \tag{2}$$

If equation (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) to obtain:

$$\sum_{j=1}^{L} \{\alpha_j x_j\} + \beta \quad \geq 0 \tag{3}$$

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[1]. This algorithm is based on Binary Hyperplane Testing [9]. Directly implementing equation (1) requires $2(L^2 + L)$ multiplications, $2(L^2 - 1)$ additions and $L^2 + L$ words of memory storage, while implementing equation (3) requires only L multiplications, L additions, and $L + 1$ words of memory storage.

The Single Node Processor performs the computations stated in equation (3). Its output is a '0' if equation (3) is satisfied and a '1' otherwise.

[1]Some applications use a weighting matrix $\mathbf{W}(\mathbf{x})$ that depends on the input vector $\mathbf{x}$. Equation (3) is still valid in this case, but a preprocessor is needed to compute the α and β coefficients in real-time.

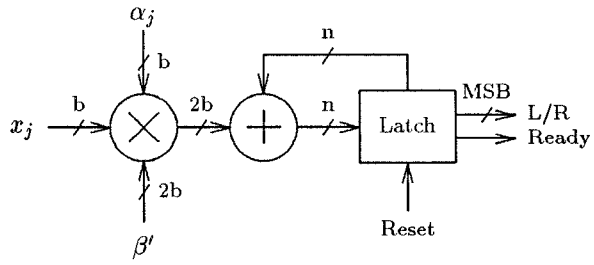


Figure 1: Block diagram of the Single Node Processor (SNP). Coefficients α_j and β' are stored in off-chip memories. The multiplier computes $p_j = \alpha_j x_j + \beta'$.

Our implementation of a SNP uses the pipelined parallel multiplier developed by McCanny and McWrither [10, 11]. We do not need a comparator unit in the SNP. The most significant bit (MSB) of the accumulated products directly represents the processor's output.

Figure 1 shows a block diagram of the SNP processor. 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 [2] β', and returns a $2b$-bit number $p_j = \alpha_j x_j + \beta'$. 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 Figure 2. Each cell consists of a full adder and three latches. Carry is propagated to the neighboring cell and sum is stored within

[2] 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.

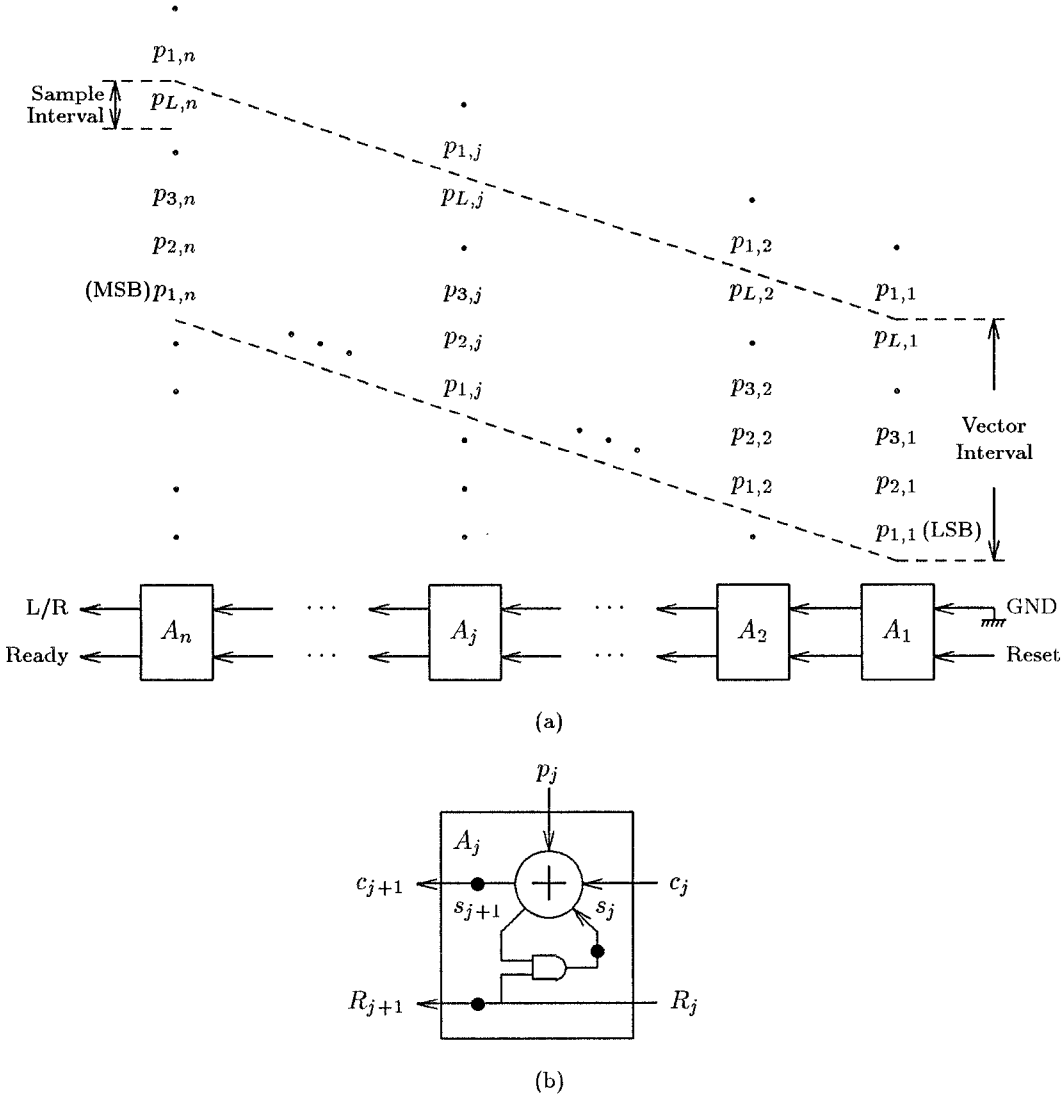


Figure 2: Detailed diagram of the accumulator (a) Linear array of cells. Input data is applied in a skewed fashion and carry is propagated between cells. Reset is applied to the first cell and is propagated down the array. Cells in this array are reset in a staggered fashion. (b) Detail of each cell. 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 Figure 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 the SNP is

$$L_{SNP} = b + n + L = 3b + \lceil \log L \rceil + L. \tag{4}$$

For example, if the word size $b = 8$, and the vector dimension $L = 16$, we have $L_{SNP} = 44$ clock cycles.

3.2 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 Figure 3.

Figure 4 shows the architecture of a TSVQ using d SNP processors. The coefficients necessary for each processor'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 generates a Go signal to initiate processing by processor SNP(i+1). This Go signal is used to reset the accumulator. The final processor SNP(d-1), returns the complete index u. The memory bandwidth is $3b$ for each processor. (Memory bandwidth can be reduced from $3b$ to b by preloading β' into on-chip registers.) 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

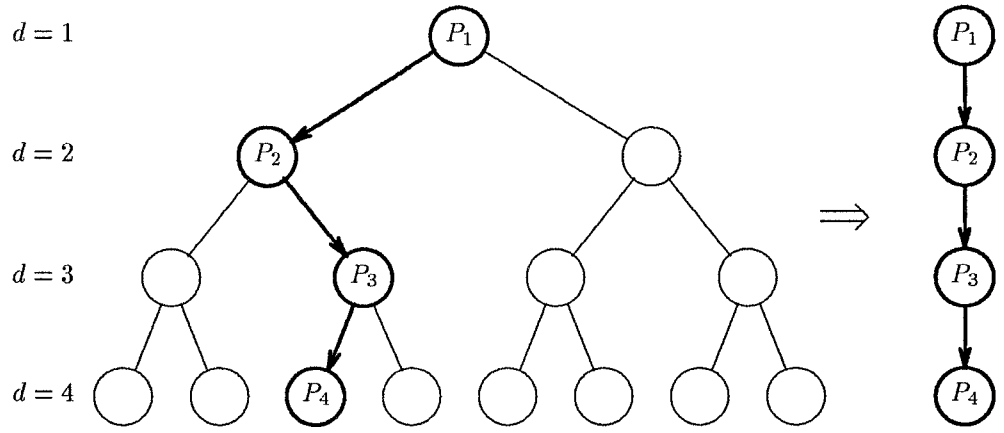


Figure 3: Traversal of a binary tree of depth $d = 4$, and its mapping onto a linear array of processors.

throughput of this scheme is one L-dimensional vector per L clock cycles. This architecture has been implemented [1] using $2\mu m$ N-well technology.

A TSVQ can also be built by using one SNP and recirculating the input data d times as shown in Figure 5. 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,

$$L_{TSVQ} = dL_{SNP} = d(b + n + L). \tag{5}$$

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.

3.3 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

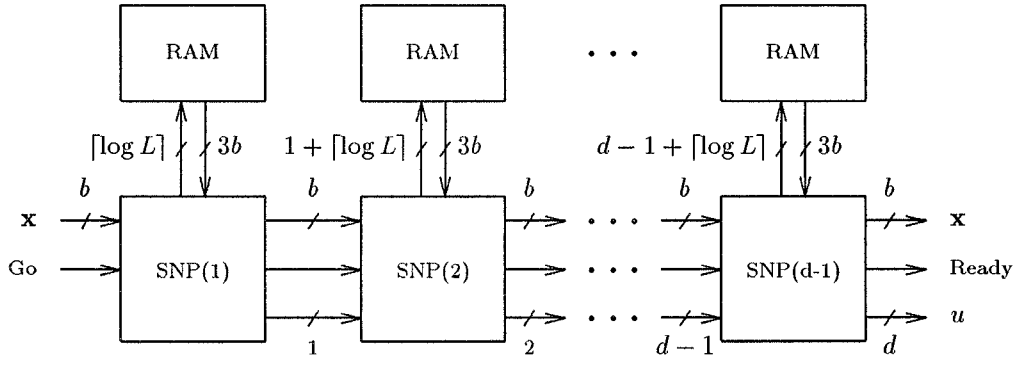


Figure 4: Systolic architecture for computing TSVQ. Each processor adds its partial index to the index data-path, and generates a control signal to initiate processing by its neighbor down the tree. No global control signals are needed.

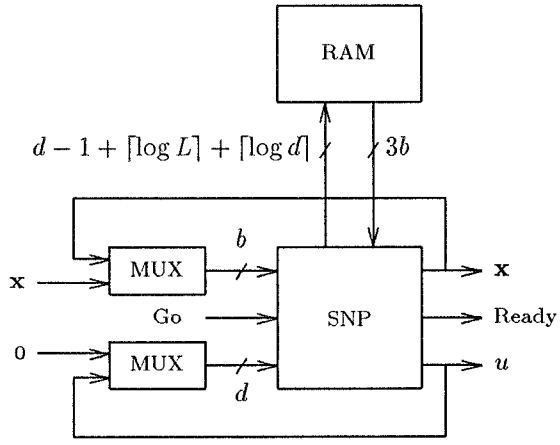


Figure 5: TSVQ architecture using one SNP and recirculating registers. Input vector $\mathbf{x}$ must be recirculated d times, once for each level of the binary tree.

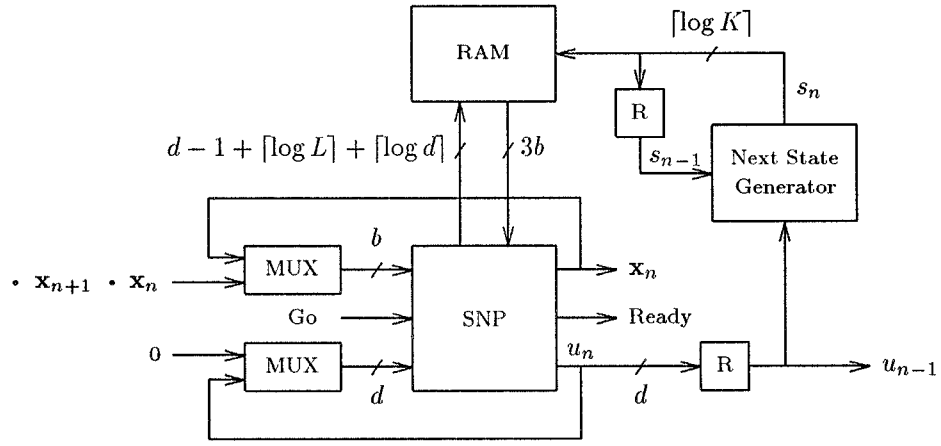


Figure 6: Block diagram of the FSTSVQ processor. R is a unit delay element. The $\lceil \log K \rceil$ bits of s_n are used as address bits for the RAM.

additional hardware for generating the next state information, is sufficient to build an FSTSVQ.

Figure 6 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.

4 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.

4.1 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,

$$L_{FSTSVQ} = L_{TSVQ} + L_{NSG}, \tag{6}$$

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.

4.2 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 Figure 7. 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:

$$s_n^W = f_1(u_{n-1}, \mathbf{s}_{n-1}), \tag{7}$$

$$s_n^{NW} = f_2(u_{n-\frac{N}{k}-1}, \mathbf{s}_{n-\frac{N}{k}-1}), \tag{8}$$

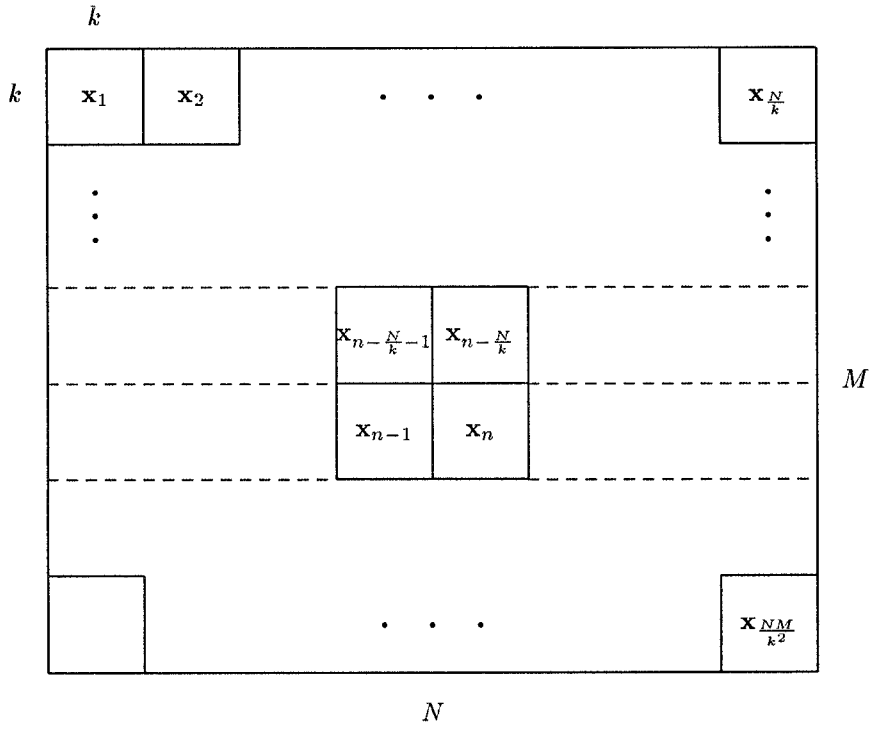


Figure 7: Block-scan of an input image of size $N \times M$. The internal state of the FSVQ while quantizing vector $\mathbf{x}_n$ depends on the quantized outputs of vectors $\mathbf{x}_{n-1}$, $\mathbf{x}_{n-\frac{N}{k}}$, and $\mathbf{x}_{n-\frac{N}{k}-1}$.

$$s_n^N = f_3(u_{n-\frac{N}{k}}, \mathbf{s}_{n-\frac{N}{k}}), \tag{9}$$

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

Equations (7), (8), and (9) suggest the data dependency graph shown in Figure 8(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 Figure 8(b).

Consider a linear array of $\frac{N}{k}$ processors as shown in Figure 9. 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.

Figure 10 shows the detailed block diagram of each processor. The TSVQ unit consists of a SNP and recirculating registers as shown in Figure 5. 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 Figure 10. The memory size requirement reduces to $2^d(L+1)K^2$ words.

4.3 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 (6). 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

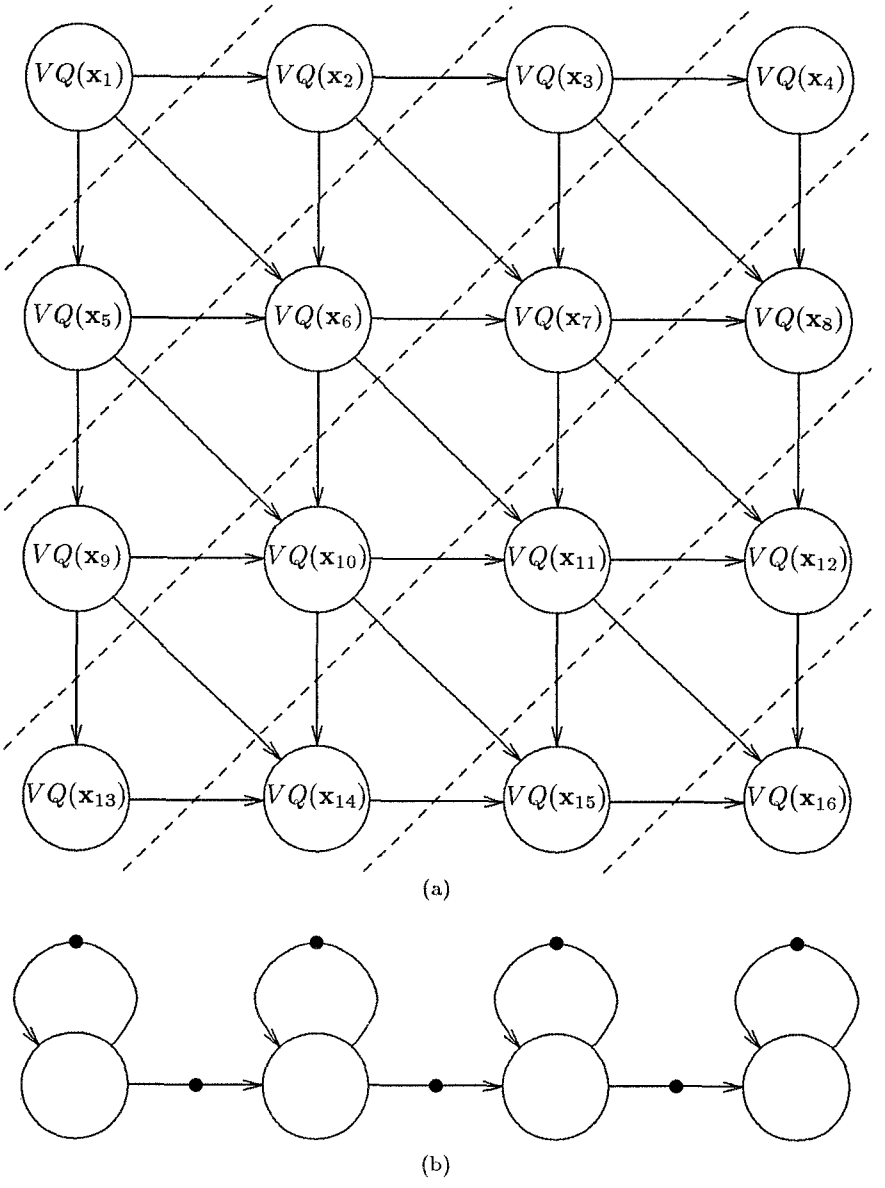


Figure 8: (a) Dependency graph of the FSTSVQ for image coding for an image of size $N = M = 4k$. Each circle represents the quantization of one input vector. (b) Projection in the vertical direction. Solid circles are unit delay elements.

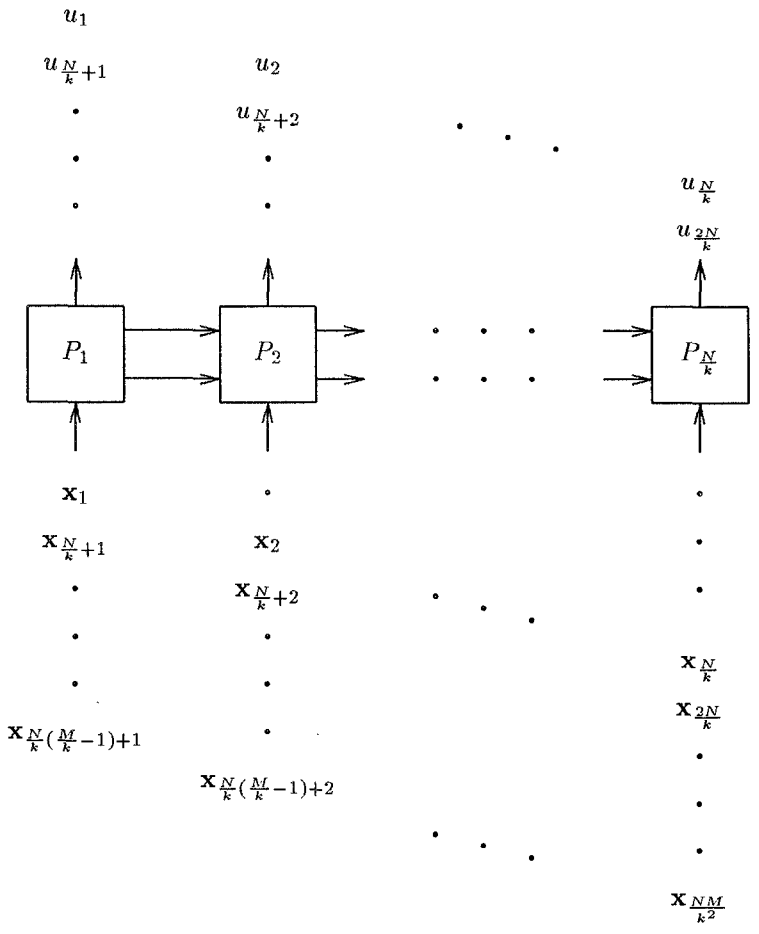


Figure 9: Systolic architecture using $\frac{N}{k}$ processors arranged as a linear array. Input data blocks are applied to this array in a skewed fashion.

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.

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.

5 Data rearrangement hardware

Input data is usually available in line-scan mode. Images are scanned pixel by pixel from left to right and top to bottom. However, the FSTSVQ systolic architecture for images presented above expects data in block-scan mode. In this section we present a simple architecture for converting line-scan mode data into the format required by the FSTSVQ systolic architectures.

Figure 11 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. This is the minimum necessary for converting line-scanned data into block-scanned mode. 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 Figure 11(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 Figure 11(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

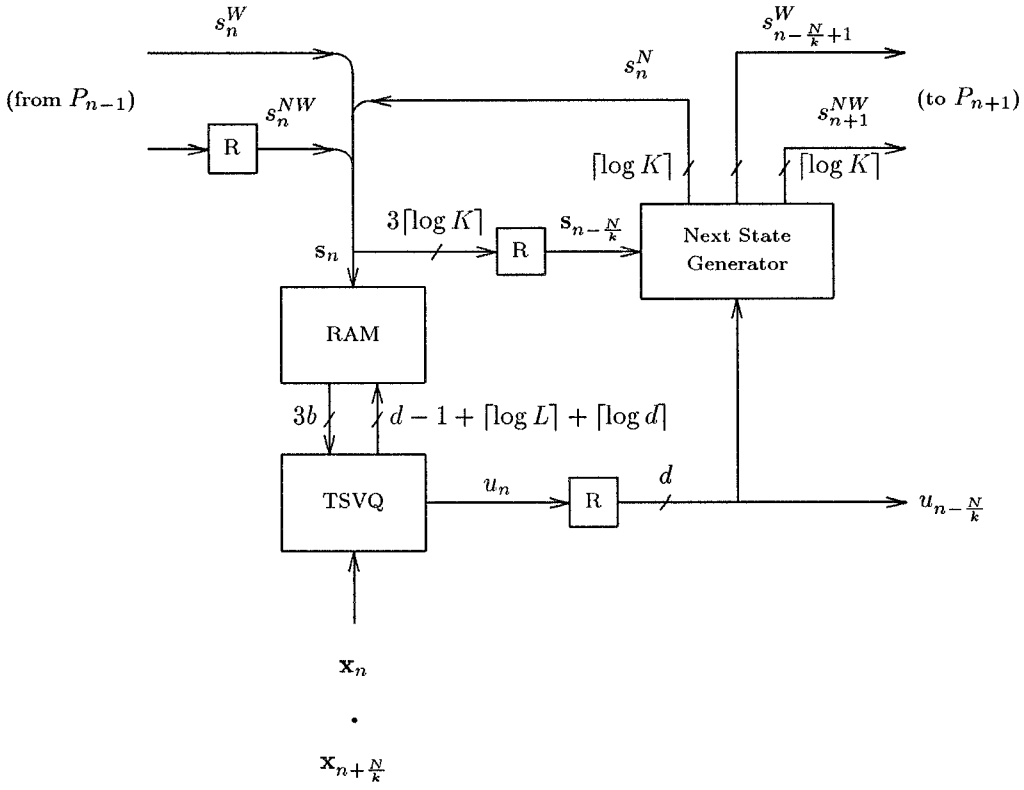


Figure 10: Detailed block diagram of processor P_n. The current state $\mathbf{s}_n = (s_n^W, s_n^{NW}, s_n^N)$ is composed of partial substates from the west and northwest (from the previous processor), and the north (from previous computations in this processor).

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.

6 VLSI Implementation

We have implemented a Single Node Processor using MOSIS' $2\mu m$ N-well process on a $7.9mm \times 9.2mm$ die [1]. 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. The next state generator is a simple table look up and will be implemented using ROMs. We are in the process of assembling these SNPs into an FSTSVQ board.

7 Conclusions

We have presented 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 1024×1024 images is 31.5 Mpixels/sec. An implementation at 31.5 MHz can quantize 1024×1024 size images in real-time. HDTV images have a typical pixel rate of 70 Mpixels/sec. Using present day VLSI technology, this speed can be achieved by using two quantizers in parallel. Since there is no interframe dependency, these two quantizers will operate on alternate frames.

Acknowledgments

We had many beneficial discussions with Xiaonong Ran, Yunus Hussain, Nariman Farvardin, and KuoJuey Ray Liu. We are grateful for their contri-

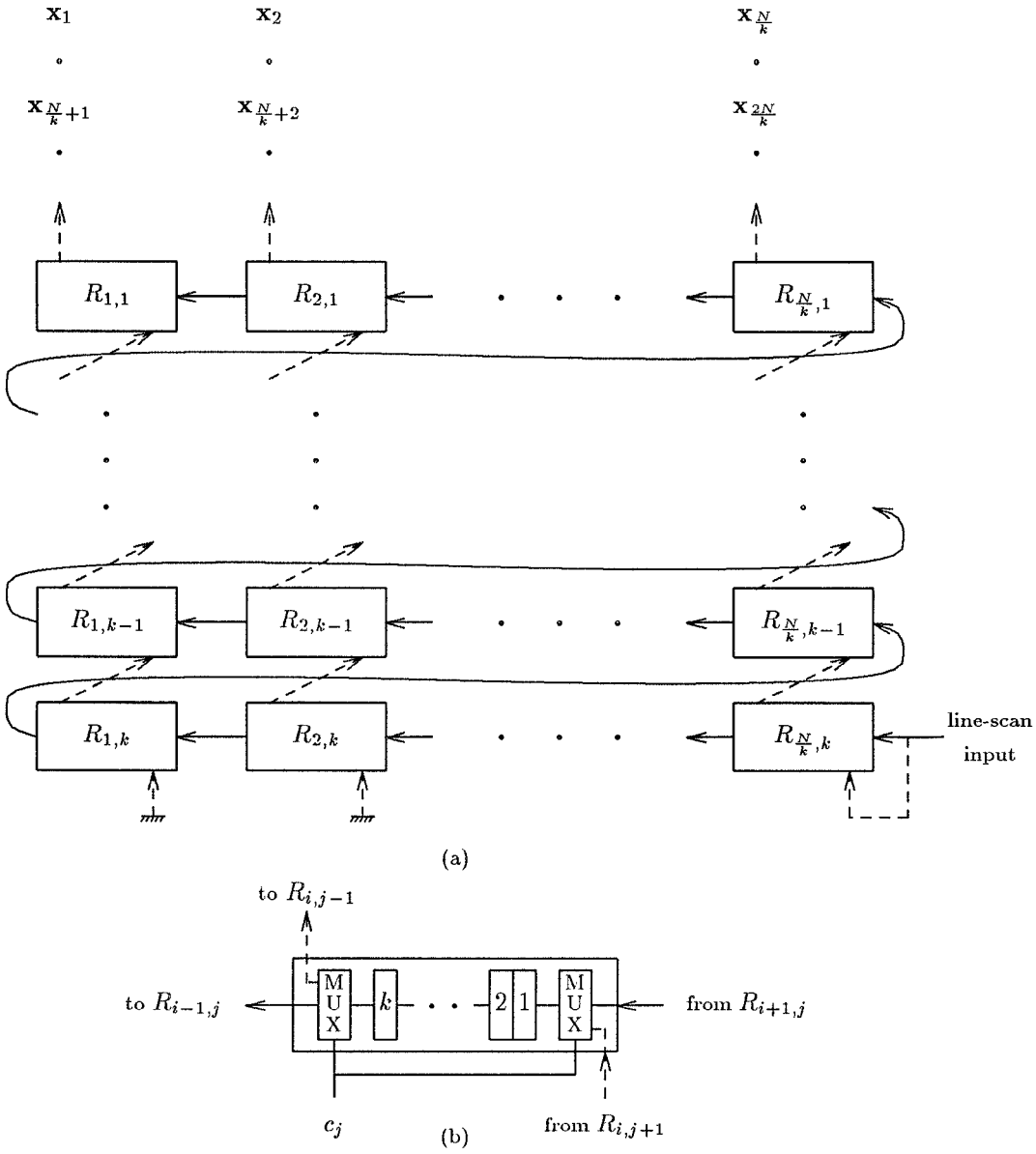


Figure 11: Architecture to convert line-scan image data into block-scan mode. (a) 2-D array of cells. (b) Detail of cell $R_{i,j}$ with k latches and two multiplexers. Control signal c_j switches the cell between horizontal and vertical modes of operation.

bution to this work.

References

[1] R. Kolagolta, S.-S. Yu, and J. F. JáJá, "VLSI implementation of a tree searched vector quantizer," Tech. Rep. SRC TR 90-74, University of Maryland, Oct. 1990.

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

[3] W. C. Fang, C. Y. Chang, and B. J. Sheu, "Systolic tree-structured vector quantizer for real-time image compression." Private communication, Oct. 1990.

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

[5] R. M. Gray, "Vector quantization," *IEEE ASSP Mag.*, vol. 1, pp. 4–29, 1984.

[6] 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.

[7] Y. Hussain and N. Farvardin, "Variable-rate finite-state vector quantization and applications to speech and image coding," *IEEE Trans. Signal Processing*, submitted for publication.

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

[9] 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.

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

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