National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

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

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

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

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

AVIS

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

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

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

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

Canada

Computational*RAM Implementations of Vector Quantization for Image and Video Compression

Thinh Minh Le

A thesis
submitted to the School of Graduate Studies and Research
in partial fulfillment of
the requirement for the degree of
Master of Applied Science
in
Electrical Engineering

Ottawa-Carleton Institute of Electrical Engineering
Department of Electrical Engineering
Faculty of Engineering
University of Ottawa
Ottawa, Ontario, K1N 6N5

ISBN 0-612-07822-1

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

To my family

and

my loving best-friend-and-fiancee

I hereby declare that I am the sole author of this thesis.

I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholary research.

Thinh M. Le

I further authorize the University of Ottawa to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholary research.

Thinh M. Le

Abstract

In this thesis, Computational*RAM (C*RAM) implementations of Vector Quantization for image and video compression is proposed. Vector Quantization (VQ) is a powerful technique for low bit rate image and video compression. The coding performance can be further improved by employing the adaptive techniques at the expense of significant increase in computational complexity. Recently, a number of VLSI architectures for implementing VQ have been reported in the literature. These architectures are not programmable and rather complex. We proposed the mapping of VQ-based image and video compression algorithms on the C*RAM architecture. C*RAM is a SIMD-memory hybrid where the processing elements are attached to memory columns of a conventional RAM to take advantage of the on-chip high memory bandwidth for parallel computation.

Implementation of VQ based on the mean absolute error measure using the C*RAM's parallel structure is first presented. This follows with the details of the proposed Modified Scalable VQ algorithm. C*RAM implementation of Modified SVQ is a mapping of binary tree onto parallel structure which results in scalable bit stream. We then propose a novel adaptive codebook replenishment VQ and index-based motion estimation (AVQ+ME) algorithm for video compression. Finally, the C*RAM implementation of AVQ+ME is presented. Simulation results demonstrate a good coding performance at a significantly reduced computational complexity.

Acknowledgment

First of all, I would like to thank Dr. Panchanathan at the University of Ottawa for his continously supervision and encouragement. I am also grateful of Dr. Snelgrove at Carleton University for his advice and support.

This work was made possible by the support from the Natural Sciences and Engineering Research Council of Canada (NSERC), Microelectronics Network (MicroNet) under the Network Centers of Excellence (NCE) program of the Government of Canada, the School of Graduate Studies and Research at the University of Ottawa, and the Department of Electrical Engineering at the University of Ottawa.

My appreciation are also to members of the Visual Computing and Communications Laboratory, Ms. Lucette Lepage, Ms. Michele Roy, and other members of the Department of Electrical Engineering.

Most importantly, I would like to thank my parents, my family, and my loving fiancee for their encouragement. It would not have been possible to accomplish this work without their love and support.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ALU: Arithmetic Logic Unit

BiCMOS: Bipolar Complementary Metal-Oxide Semiconductor

BMA: Block Matching Algorithm

CAM: Content-Addressable Memory

CBR: Constant Bit Rate

CCITT: Consultative Committee on International Telephony and Telegraphy

CD: Codeword Distortion

CIF: Common Intermediate Format

CMOS: Complementary Metal-Oxide Semiconductor

CPU: Central Processing Unit

CR: Codebook Replenishment

C*RAM: Computational Random Access Memory

CVQ: Classified Vector Quantization

CW: Codeword

DCT: Discrete Cosine Transform

DPCM: Differential Pulse Code Modulation

DRAM: Dynamic Random Access Memory

DT: Data Transposer

DWT: Discrete Wavelet Transform

FCB: Full-resolution Codebook

GOP: Group Of Picture

HDTV: High Definition TeleVision

HVS: Human Visual System

ISDN: Integrated Services Digital Network

ISO: International Standards Organization

IDCT: Inverse Discrete Cosine Transform

JPEG: Join Photographic Experts Group

LBG (algorithm): Y. Linde, A. Buzo, and R. M. Gray algorithm

LR: Label Replenishment

LSB: Least Significant Bit

MAE: Mean Absolute Error

MD: Maximum Descent

ME: Motion Estimation

MIMD: Multiple-Instruction stream, Multiple-Data stream

MISD: Multiple-Instruction stream, Single-Data stream

MOS: Mean Opinion Score

MOS: Metal-Oxide Semiconductor

MPEG: Moving Picture Experts Group

M/RVQ: Mean/Residual Vector Quantization

MSB: Most Significant Bit

MSE: Mean Square Error

NMSE: Normalized Mean Square Error

NTSC: National Television Standards Committee

PC: Primary Codebook

PE: Processing Element

PD: Pixel Distortion

PIT: Progressive Image Transmission

PNN: Pairwise Nearest Neighbor

PSNR: Peak Signal-to-Noise Ratio

RAM: Random Access Memory

RCB: Reduced-resolution Codebook

SA: Search Area

SC: Secondary Codebook

SIMD: Single-Instruction stream, Multiple-Data stream

SISD: Single-Instruction stream, Single-Data stream

SNR: Signal-to-Noise Ratio

SRAM: Static Random Access Memory

SVQ: Scalable Vector Quantization

TSVQ: Tree-Structure Vector Quantization

VLC: Variable Length Code

VLSI: Very Large Scale Integrated circuit

VQ: Vector Quantization

CHAPTER 1

Introduction

Visual communications is becoming increasing important with the advent of broadband networks and compression standards (JPEG, H.261, MPEG). Typical visual communication applications includes high-definition television (HDTV), image/video databases, multimedia communications, videoconferencing, etc. [1-2].

In image processing, it is often necessary to represent the image in digital form. This is performed by first sampling a continuous image in time to obtain the frames. The individual frames are then sampled into N_1 x N_2 pixels (picture elements) as shown in Figure 1.1. We note that (dx, dy) and dt are the sampling intervals in the spatial and temporal dimension, respectively. Each pixel is quantized into one of 2^b gray levels $(0,1,2,...,2^b-1)$. The raw data stream entails a large bit rate, for example, an uncompressed color video sequence sampled at a frame rate of 30 frames per second, where each frame is of size 288 x 352 pixels in the Common Intermediate Format (CIF)[85], requires the transmission of 73 Mbits per second (bps). The large channel bandwidth and memory requirements for the transmission and storage of images necessitate the more efficient image/ video compression techniques.

The goal of image/video compression is to reduce the number of bits required to represent an image while maintaining an acceptable image fidelity. Efficient compression is achieved by exploiting the spatial and temporal redundancies in the video sequence. The spatial redundancies are removed by using intraframe techniques, while the temporal correlations are usually exploited using interframe techniques.

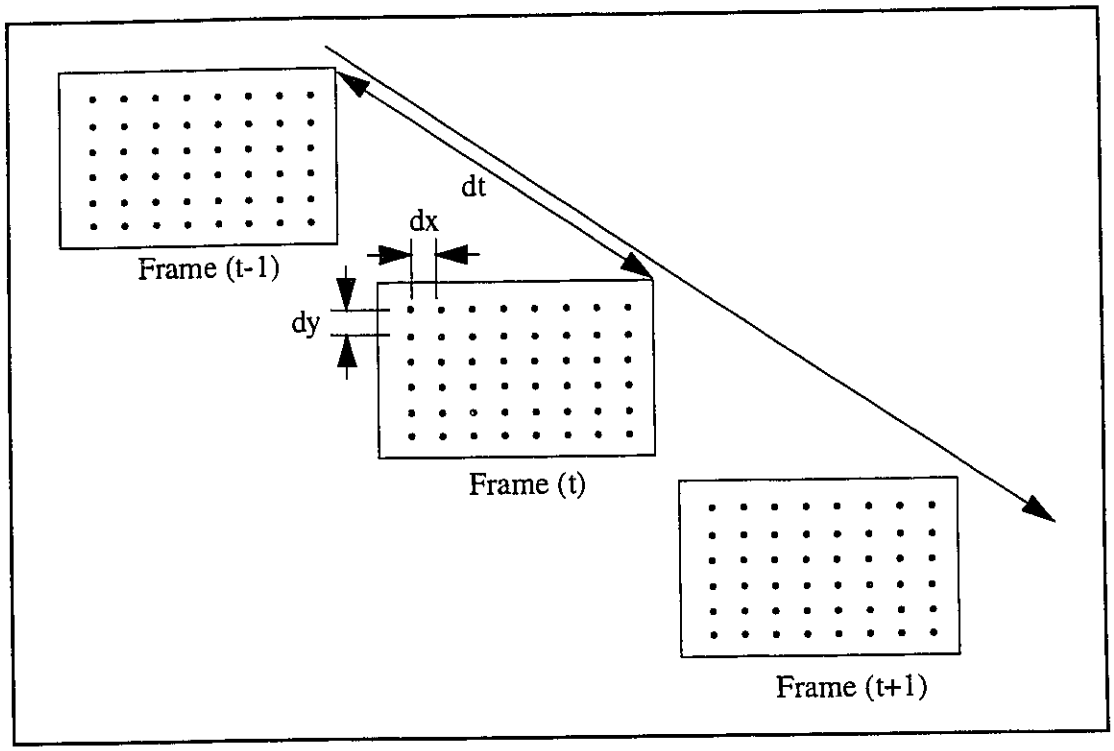

FIGURE 1.1 Sampling of a video sequence

Several intraframe and interframe image/video coding techniques based on scalar quantization have been reported in the literature [1]. Examples include: differential pulse code modulation (DPCM), transform coding, and hybrid coding. According to Shannon's rate distortion theory [3,4], a better performance can be achieved by coding vectors instead of scalars, even when the source is memoryless. Several vector quantization (VQ) techniques image coding [2] applications have recently been reported in the literature. Nasrabadi *et al.*[2] have presented a review of the basic vector quantization techniques and many of their variations for image/video compression.

In VQ[5], a set of representative images (training set) is decomposed into L-dimensional vectors. An iterative clustering algorithm such as the LBG algorithm [6] is used to generate a codebook W = $\{w_1, w_2, ..., w_N\}$, where N is the size of the codebook. This codebook is then made available at both the transmitter and the receiver. The two basic steps in VQ are encoding and decoding. In the encoding process, the image to be coded is decomposed into L-dimensional vectors. For each input vector $V_i=\{v_{i1}, v_{i2}, ..., v_{iL}\}$, the codebook W is searched using the nearest neighbour rule to find the closest codeword w_j. Compression is achieved by transmitting the label (index) j corresponding to codeword w_j. Reconstruction of images can be implemented by simple table lookup techniques where the label j is used as an address to a table containing the codeword w_j.

In conventional VQ techniques, the computational complexity is O(KLN) where K is the number of input vectors in a frame of dimensional L, and N is the number of codewords in the codebook. This high computational complexity of VQ is an impediment to real-time implementation. Tree-structure VQ [53-57] reduces the complexity of VQ to O(KLlogN) by rearranging the codewords in a tree fashion. If the input vectors are searched in parallel, the complexity can be reduced by a factor of K. On the other hand, if the codewords are searched in parallel, the complexity can be reduced by a factor of N. Example implementations which exploits the parallel in the dimensions of codeword includes Content Addressable Memory (CAM)'s and Single-Instruction Multiple-Data (SIMD) machines, etc. The recent advances in VLSI technology with decreasing circuit sizes and increasing clock rates make real-time implementations of VQ-based algorithms feasible.

In this thesis, VQ-based image/video compression algorithms are implemented on the parallel structure of a Computational*RAM (C*RAM) [100,101]. C*RAM is a conventional RAM with SIMD processing elements added to the memory array. The main advantages of this computing structure are the fast on-chip computation and minimal power dissipation in processor-memory data transport. Implementation of VQ for image compression using C*RAM's parallel structure is proposed. In addition, implementation of spatial scalability based on VQ using the C*RAM structure has also been detailed. This includes a binary to parallel mapping algorithm which enables

binary-split codewords to be arranged on the C*RAM's parallel structure. This follows with the VQ implementation for video compression. A new adaptive codebook replenishment VQ technique combined with index-based motion estimation (AVQ+ME) algorithm is then proposed. The AVQ+ME algorithm results in significant reductions in computational complexity and a sufficient coding performance.

1.1 Thesis Outlines:

The rest of the thesis is organized as follows:

Chapter 2 presents a review of image/video compression. The first section discusses the different data compression techniques in terms of lossless and lossy coding. The evaluation criteria are also presented. The distortion is specified either by an objective measure such as the peak signal-to-noise ratio (PSNR), or by a subjective measure such as the *mean opinion score* (MOS). The second and third sections include a review on the image and video compression techniques , respectively, with special emphasis on VQ techniques. The fourth section describes the image and video compression standards. Finally, VLSI implementations of VQ are reviewed in the fifth section, followed by a summary in section six.

Chapter 3 presents an overview of the Computational*RAM. The chapter includes sections on architecture, processing element model, and other C*RAM design variations.

Chapter 4 introduces the C*RAM implementation of VQ for image compression. In this chapter, a algorithm based on an Mean Absolute Error (MAE) measure for implementing VQ on the C*RAM's parallel structure is presented.

Chapter 5 presents the C*RAM implementation of the proposed Modified Scalable VQ. The proposed technique is then implemented on the C*RAM structure. This chapter also includes the binary-tree to parallel mapping algorithm to facilitate C*RAM implementation.

Chapter 6 introduces the C*RAM implementation of VQ for video compression. In this chapter, an adaptive codebook replenishment VQ using index-based motion estimation (AVQ+ME) algorithm is proposed. This algorithm exploits the spatial as well as temporal redundancies in order to reduce the bit rate. In addition, the proposed AVQ+ME technique has been implemented on the parallel structure of the C*RAM.

Chapter 7 concludes the thesis with future work and recommendations.

The thesis also provides supplements on:

- Appendix A: Computer Arithmetics for C*RAM; and
- Appendix B: The Design of the Data Transposer;
- Appendix C: Bit Rate Analysis.

1.2 Thesis Contributions

- Sections 4.2-4.3: The C*RAM implementation of VQ-MAE.
- Sections 5.3-5.5: The Modified SVQ algorithm and C*RAM implementation.
- Sections 6.2- 6.3: The AVQ+ME algorithm and C*RAM implementation.
- Section A.2-A.3: C*RAM macros and algorithms.
- Section B.3-B.6: The design of the Data Transposer.

CHAPTER 2

Review of Image and Video Compression

In this chapter, the image and video compression techniques are reviewed. The different data compression techniques classified as lossless and lossy coding are discussed first. A review of the image and video compression techniques with special emphasis on vector quantization-based techniques are then presented. This follows with the details of the image and video compression standards. Finally, VLSI implementations of VQ-based algorithms are reviewed, followed by the summary.

2.1 Data Compression Techniques

The large memory capacity and channel bandwidth requirements for storage and transmission of visual data necessitates the use of efficient image/video compression techniques. The goal of image/video data compression is to reduce the number of bits required to represent an image/ video signal while maintaining an acceptable fidelity [1].

Image and video compression is essentially a redundancy reduction process. Efficient compression is achieved by exploiting the spatial, temporal, structure and psychovisual redundancies.

The neighbouring pixels in an image/video frame are usually highly correlated. For example, there is a strong dependancy between the pixel values in a local region and hence a pixel value can be inferred from the neighbouring pixel values. This correlation is called the *spatial redundancy*, and is typically removed by employing intraframe coding techniques such as predictive coding, transform coding, etc.

Temporal redundancy refers to the correlation between the successive frames in an image/video sequence. The differences between successive frames are due to object motion and camera motion. Temporal redundancy is usually removed by employing interframe coding techniques such as motion estimation/compensation, frame replenishment, etc.

Structure redundancy refers to the fact that an image is the projection of 3-dimensional objects onto a 2-dimensional plane [7]. Therefore, if the image is encoded based on the structure of the enclosed elements, a high compression ratio could be achieved. Such example is the scene containing the pingpong ball moving in a plain background.

Most image/video frames contain *psychovisual redundancy*. That is, some information may be removed without sacrificing the subjective image quality. Therefore, if the image/video signal is encoded based on the properties of the *human visual system* (HVS), a higher compression ratio may be achieved [8,9]. Some of the properties of the human visual system that may be exploited are:

- Greater sensitivity to small signal changes in dark areas.
- Lower sensitivity to random noise compared to systematic noise.
- Lower sensitivity to faster moving objects.
- Lesser perception of distortion at higher spatial frequencies.

Image and video compression techniques can be generally classified into: "distortionless" or "lossless" and "minimum distortion" or "lossy" schemes.

Lossless coding

Lossless coding minimizes the average number of bits per pixel without any loss in image quality. That is, the decoder reconstructs the exact input image from the encoded image. Information theory states that the source can be exactly encoded with H bits/pixels, where H is the source entropy. For a source with 2^b possible independent symbols with probabilities p_i, i=1, 2, ..., 2^b, the entropy is given by:

$$H = -\sum_{i=1}^{2^b} p_i \log_2 p_i \tag{2.1}$$

This results in a variable length code (VLC) where shorter codewords are assigned to more probable pixel values and longer codewords are assigned to less probable pixel values. Huffman [10] and Arithmetic coding [11] are the most popular approaches for lossless coding. Arithmetic coding achieves higher compression ratios than Huffman coding, but it is more difficult to implement. Another technique for lossless coding is run length coding [12] in which a reduction in the bit rate is achieved by sequentially transmitting or storing the pixel values (run) followed by the number of its repetitions (length). Significant compression is possible if the input image is characterized by long runs.

Lossy coding

The purpose of lossy coding or minimum distortion is to minimize the bit rate for a given average distortion or equivalently, to minimize the average distortion for a given bit rate. For an image source X, the average distortion D is defined as:

$$D = E\{d(X, \hat{X})\} \tag{2.2}$$

where $d(X, \hat{X})$ is a distortion measure between the source X and its reproduction $\hat{X}$. Clearly, the design of such an encoding scheme depends on the statistics of the source X and the characteristics of the distortion function d. Rate distortion theory [3] provides the theoretical lower bound on

the bit rate of any quantizer. For the source X with probability function $p_X(X)$, the rate distortion function is defined as:

$$R(\tilde{D}) = min\{I(X,\hat{X})\} \tag{2.3}$$

where the minimum is taken over all the encoding schemes which result in average distortion D less than some value $\tilde{D}$, and $I(X,\hat{X})$ is the average mutual information between X and $\hat{X}$ and is defined as:

$$I(X,\hat{X}) = \sum_{x}\sum_{\hat{x}} p_X(x)\, p_{\hat{X}/X}(\hat{x}/x) \log \frac{p_{\hat{X}/X}(\hat{x}/x)}{p_{\hat{X}}(\hat{x})} \tag{2.4}$$

where $p_{\hat{X}/X}(\hat{x}/x$ is the conditional probability for $\hat{X}$ given X.

This definition of the rate distortion function indicates that for a given average distortion D, the minimum transmission rate is R(D). Shannon's coding theorem states that it is possible to design a coding system which achieves the average distortion D at a transmission rate arbitrarily close to R(D).

Evaluation Methods

The image quality of different coding schemes can be evaluated using subjective and objective measures. Subjective measures refer to the judgement of human viewer using quality rating or impairment rating. In subjective quality rating, the reconstructed image can be judged according to scales such as: Excellent, Good, Fair, Poor, and Bad. On the other hand, impairment rating scale includes: Imperceptible, Perceptible but Annoying, Slightly Annoying, Annoying, and Very Annoying. The average of the ratings for a given picture is the mean opinion score (MOS) and is used as a measure of the quality of the reconstructed image.

Objective measures are sometimes referred to as distortion measures. The most commonly used distortion measures are the followings:

- Mean Square Error (MSE): For an image of dimensions X and Y, MSE is defined as:

$$MSE = \frac{1}{XY}\sum_{k=1}^{X}\sum_{l=1}^{Y}(x_{kl}-\hat{x}_{kl})^2 \tag{2.5}$$

where x_k and $\hat{x}_k$ denote the original image pixel and encoded image pixel, respectively.

- Mean Absolute Error (MAE):

$$MAE = \frac{1}{XY}\sum_{k=1}^{X}\sum_{l=1}^{Y}|x_{kl}-\hat{x}_{kl}| \tag{2.6}$$

- Normalized Mean Square Error (NMSE):

$$NMSE = \left(\sum_{k=1}^{X}\sum_{l=1}^{Y}(x_{kl}-\hat{x}_{kl})^2\right) / \left(\sum_{k=1}^{X}\sum_{l=1}^{Y}x_{kl}^2\right) \tag{2.7}$$

- Signal to Noise Ratio (SNR):

$$SNR = 10\cdot\log\frac{1}{NMSE} \tag{2.8}$$

- Peak Signal to Noise Ratio (PSNR): The coding performance is commonly evaluated using the PSNR, which is defined as:

$$PSNR = 10\cdot\log\frac{(Peak-signal-value)^2}{MSE} \tag{2.9}$$

2.2 Image Compression Techniques

Recall that from section 2.1, in image compression, the spatial redundancies are typically removed by employing intraframe coding techniques. These techniques include predictive coding, transform coding, vector quantization, and hybrid techniques. In this section, the various intraframe image coding techniques are reviewed, with an emphasis on vector quantization.

2.2.1 Predictive Coding

Predictive coding exploits the mutual redundancy between neighbouring pixels. Rather than encoding the pixel intensity, its value is first predicted from the previous encoded pixels. The predicted pixel value is then subtracted from the actual pixel value and the difference (prediction error) is quantized and coded for transmission. The quantized prediction error is used at the receiver to reconstruct the image.

Differential Pulse Code Modulation

Differential Pulse Code Modulation (DPCM) is the most commonly used predictive coding technique. A block diagram of the DPCM system is shown in Figure 2.1. The transmitter consists of two major components, the quantizer and the predictor. The predictor estimates the value of the pixel to be encoded using the values of the previously transmitted pixels. The difference e_k, known as the predictor error, between the actual pixel value x_k and the predicted value $\hat{y}_k$ is quantized, coded and transmitted.

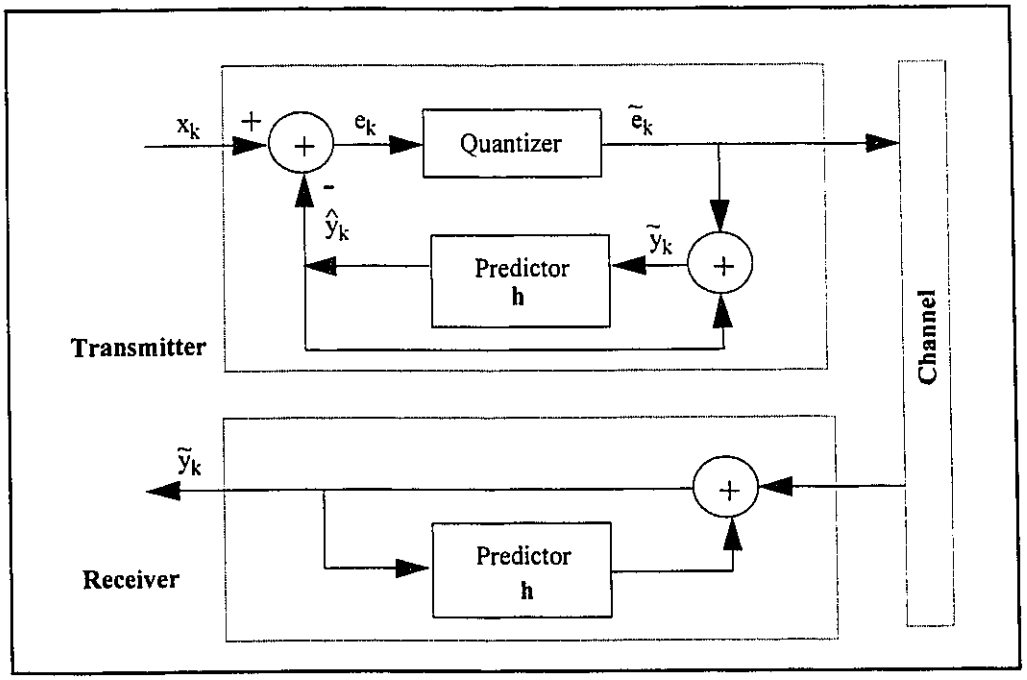

FIGURE 2.1 Block diagram of a DPCM coding system

The predictor can be linear or nonlinear. A nonlinear predictor which involves a nonlinear combination of the previously encoded pixels is difficult to implement; and hence in most DPCM systems, a linear predictor is used. A linear prediction of x_k can be determined using the following equation:

$$\hat{y}_k = \sum_i a_i \tilde{y}_{k-i} \tag{2.10}$$

where h_i are the predictor coefficients. The optimal predictor in terms of MSE is obtained by selecting the coefficients h_i so that the variance of the predictor error is minimized. The optimal coefficients can be determined by solving the following set of equations:

$$R(j) = h_{i,\,opt} R(j-1) \tag{2.11}$$

where R(j) is the correlation function.

Adaptive DPCM

The coding performance of a DPCM system can be improved by employing adaptive techniques which tracks the local statistics of the input image. Adaptivity can be achieved by either fixing the characteristics of the quantizer and varying the predictor parameters, or by varying the characteristics of the quantizer and fixing the parameters of the predictor. For example, the weighting coefficients in equation 2.10 can be periodically modified, or the quantizer levels can be changed according to the input image statistics [13].

2.2.2 Transform Coding

Unlike the conventional types of coding where the compression is achieved in the spatial domain, tranform coding compresses data in the transform domain. For still images, the input image is first divided into non-overlapping blocks $X_{m1,\,m2}$ of M_1 x M_2 pixels. In order to decorrelate the image data, a two-dimensional transform is then applied to $X_{m1,\,m2}$ as shown in Figure 2.2. The transformation maps $X_{m1,\,m2}$ into a two dimensional array $Q_{u,v}$ of transform coefficients with the same dimension. This operation is mathematically given by:

$$Q_{u,v} = \sum_{m1=0}^{M1-1} \sum_{m2=0}^{M2-1} X_{m1,m2} A_{u,v,m1,m2} \tag{2.12}$$

where $A_{u,v,m1\ m2}$ is the transform kernel. The resulting coefficients $Q_{u,v}$, u=1, 2, ... ,M1, v=1, 2, ..., M2 are then quantized, coded and transmitted.

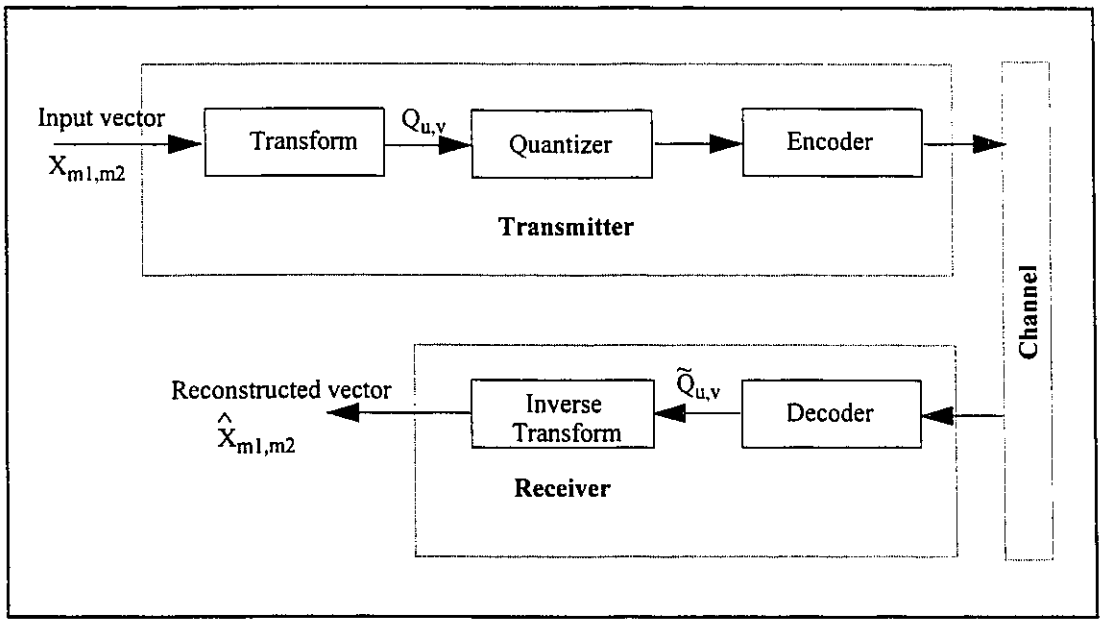

FIGURE 2.2 Block diagram of a transform coding system

At the receiver, an inverse transform operation is applied to the quantized coefficients $\tilde{Q}_{u,v}$ to reconstruct the image:

$$\hat{X}_{m1,m2} = \sum_{u=0}^{M1-1} \sum_{v=0}^{M2-1} \tilde{Q}_{u,v} A^{-1}_{u,v,m1,m2} \tag{2.13}$$

where $A^{-1}_{u,v,m1,m2}$ is the inverse transform kernel.

An optimum transform should result in statistically independent coefficients [14]. Karhunen Loeve transform is an optimum transform in terms of both MSE and subjective quality. However, it requires a large number of operations to compute; and is hence usually replaced by suboptimal transforms [13], such as Fourier transform, cosine transform, or Hadamard transform. It has been

shown that the performance of the cosine transform [15] is close to the optimal Karhunen Loeve transform.

Adaptive Transform Coding

In adaptive transform coding [13, 16], one or more of the coder parameters are changed in a way such that the coder better matches the local image statistics. For example, each block is classified into one of several categories according to some activity index. Adaptivity is achieved by allocating a large number of bits to blocks with high activity index and fewer bits to blocks with lower activity index.

2.2.3 Hybrid Predictive / Transform Coding

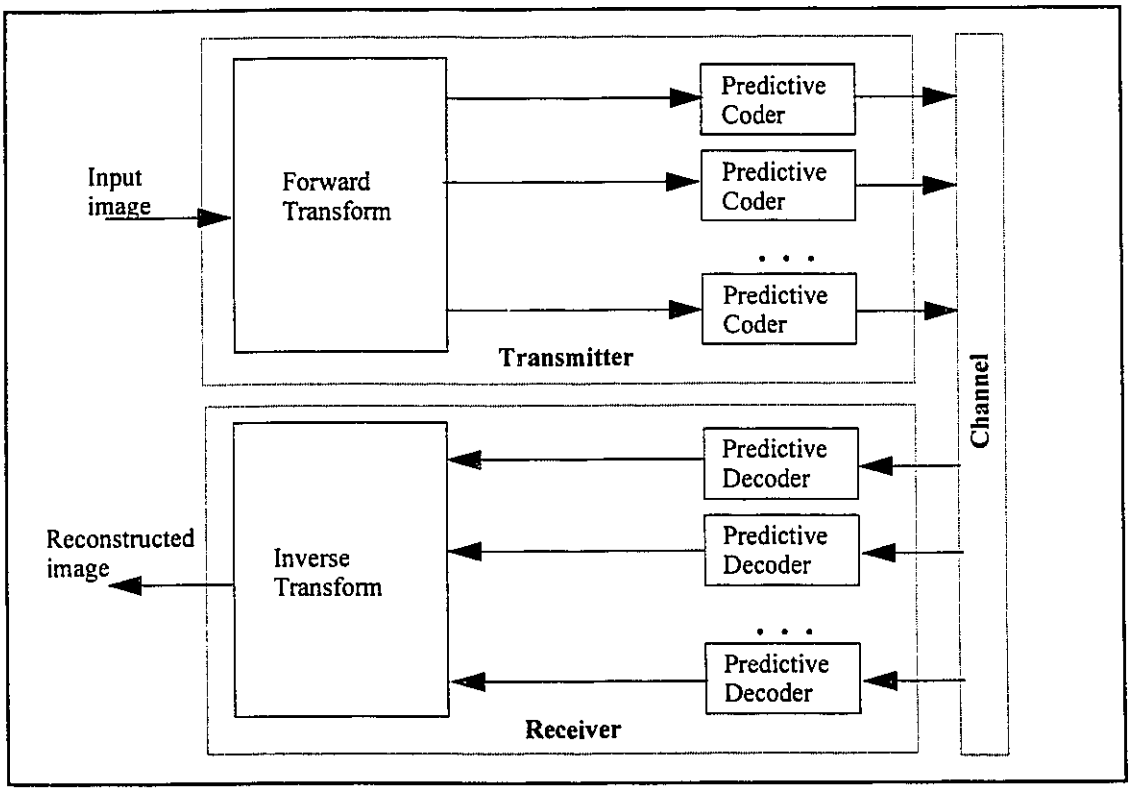

FIGURE 2.3 Block diagram of a hybrid transform / predictive coding system

Hybrid predictive/transform coding [1] is a combination of predictive and transform coding. As shown in Figure 2.3, the basic configuration is a transform operation followed by a predictive

coder for each transform coefficient. Typically, one or two-dimensional transform is applied on subimages, the coefficients are then coded using the previous coefficients as a prediction estimate.

2.2.4 Vector Quantization

Principle of Vector Quantization

Vector Quantization (VQ) is an efficient technique for low-bit rate image coding. An N-level vector quantizer is a mapping from a L-dimensional vector set {V} into a finite codebook W = {w_1, w_2, ..., w_N}.

$$Q| V \rightarrow W \tag{2.14}$$

In other words, it assigns to an input vector v a representative vector (codeword) w. The vector quantizer Q is completely described by the codebook W = {w_1, w_2, ..., w_N}, together with the disjoint partition R = {R_1, R_2, ... , R_N}, where

$$R_i = \{ v:Q(v)=w_i \} \tag{2.15}$$

and w and v are L-dimensional vectors. The partition should ideally minimize the quantization error. The block diagram of VQ is showned in Figure 2.4.

The steps involved in VQ as applied to image and video compression are vector formation, codebook generation, and quantization.

Vector Formation

The first step in VQ is to decompose the input image into vectors. The image is partitioned into two-dimensional blocks of equal or variable sizes. The features or values are extracted from the blocks and then arranged into vectors. Various vector formation schemes have been proposed in the literature. For example, the vectors can be formed from the color components of a pixel [17]; the original pixel values of the block [18]; the transform coefficients of a block of pixels [19-21]; the prediction error of a block [22,23]; the pixel values of a block normalized by the average [24]; and the intensity normalized by the mean and standard deviation [25].

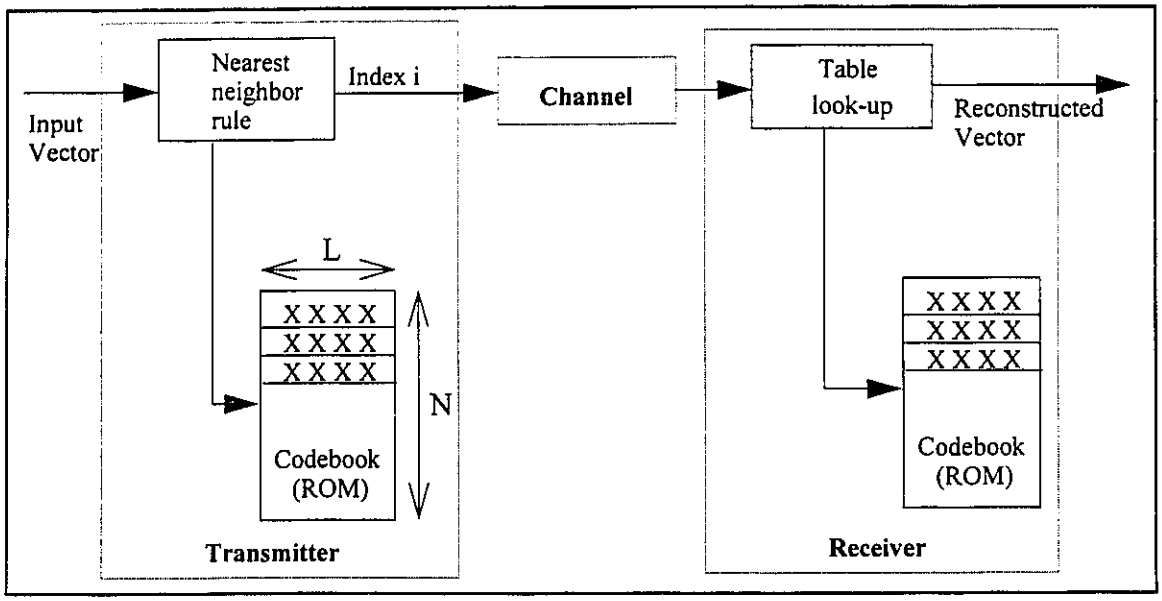

FIGURE 2.4 Principle of Vector Quantization

Codebook Generation

The key step in VQ is the design of a good codebook. An optimal codebook, using the MSE criterion, must simultaneously satisfy two necessary conditions for optimality [26].

- The input vector source V is partitioned into N closed sets or Voronoi regions $\{R_i \mid (1 \le i \le N)\}$, determined by the nearest distance rule:

$$R_i = \{v, |v - w_i| \le |v - w_m|, for \quad i \ne m\} \tag{2.16}$$

- The corresponding codewords are defined by:

$$w_i = E\{v \mid (v \in R_i)\} \tag{2.17}$$

where i = 1, 2, ... N and v is the vector in the training set.

In other words, the codewords of the optimal codebook are the mean values of the corresponding Voronoi regions under the MSE measurement criterion. The K-mean [27] or the closely related generalized Lloyd clustering algorithm proposed by Linde, Buzo, and Gray [6] (LBG) are typically used to generate the codebook. Both these clustering algorithms are iterative processes, minimizing a performance index calculated from the distances between the sample vectors and their

cluster centers. These algorithms have the advantages of not requiring any knowledge of the underlying statistics of the input source and additional minimizing a criterion related to the quantization error. However, these clustering algorithms can only assure a local optimum, which depends upon the initial codebook or cluster seeds.

A number of approaches have been developed to obtain the initial codebook. In the first [28], the centroid of the training set is calculated and split into two codewords. The LBG algorithm is applied to yield a codebook of two codewords. Each codeword is then split into two codewords to yield a codebook of four codewords. This process is repeated until the required number of codewords are generated. This approach is referred to as *binary splitting*.

The second approach starts with initial seeds for the required number of codewords, these seeds being generated by preprocessing the training sequence. Example of this approach is mode-seeking seeding in which the seeds lie at the modes of the histogram [29].

The third apprach referred to as Simulated Annealing [30,31] belongs to a family of optimization techniques called *stochastic relaxation* (SR). This is a technique in which a randomly generated perturbation to the codebook at each iteration is accepted or rejected probabilistically, where the probability depends on the change in value of the cost function resulting from such a pertubation. Although this technique generates codebooks of better performance, the computation time is drastically increased.

Recently, techniques based on neural networks [32,33], and the Pairwise Nearest Neighbour (PNN) [34] have been reported as alternatives for codebook design. In the neural network approach, the weights (which represent the codewords) between the neurons are adaptively adjusted after the presentation of each vector. The main advantage of neural network is that it converges to an asymtotic value faster than the LBG algorithm [33].

In the PNN approach [34], each of the K training vectors is considered as a separate cluster. We start with the K clusters and merge together the two clusters which result in the minimum increase

in average distortion. This merging process yields K-1 clusters. The merging process continues until only N clusters remain. PNN is faster than the LBG algorithm, however, the LBG algorithm results in a codebook that satisfies the two conditions for optimality.

Finally, the Maximum Descent (MD) approach [35] is another technique for VQ codebook generation. It produces codebooks with significant improvement on both the computation time and the coding performance compared to the LBG algorithm. The MD algorithm starts by placing all the training vectors in a global cluster. The cluster is then partitioned into two new clusters by using an optimal partitioning hyperplane. The two newly formed clusters are further partitioned into three new clusters following the maximum distortion reduction criterion. The process is repeated until the desired number of codevectors is obtained. To form N codevectors, N+1 partitioning operations are required. The MD algorithm is generally faster and produces a better image quality compared to the LBG algorithm.

Vector Quantization using Universal Codebook

VQ techniques can be broadly classified, with respect to training and codebook generation, as universal and adaptive. In this section, image coding using universal VQ is reviewed, and adaptive VQ is discussed in the next section.

Universal VQ employs a fixed codebook generated using a large set of training vectors selected from different types of images. To ensure a good image quality, the codebook must be large, which in turn increases both the bit rate and coding complexity. Moreover, if the input images have different statistics, good coding performance may not be achieved for a variety of applications. The codebook size can be reduced using techniques which exploit the local image statistics. Universal VQ includes: classified VQ, subband VQ, predictive VQ, finite state VQ, mean/residual VQ, multi-stage VQ, address VQ, tree-structured VQ, and hierarchical VQ.

Classified VQ (CVQ): In CVQ [36,37], the training vectors are classified into a finite set of classes based upon some perceptually important features, such as, the edge content and a separate

codebook is generated for each class. In the quantization process, a classifier determines the class of the input vector, which is then encoded using the appropriate codebook. The CVQ technique has been extended to include different codebooks within a class and different coding methods for each class. For example, a CVQ in the discrete wavelet transform domain (DWT-CVQ)[20] or discrete cosine transform domain (DCT-CVQ) have been reported [38,39] where the input blocks are first transformed using DWT (or DCT) and assigned to a class. The DWT (or DCT) blocks belonging to a class are then partitioned into several vectors. The subvectors are quantized separately using an appropriate codebook.

Subband VQ: Subband VQ has normally been used in speech coding. In this technique, the speech samples are separated to different frequency bands using filter banks, and each band is vector quantized separately. Quackenbush [40] has used an equal-bandwidth subband analysis/systhesis filterbank to simplify the VQ problem by coding the decimated and filtered signal, rather than the signal itself. Since an N-band subband filterbank takes a block of M input samples and produces N relatively uncorrelated blocks of M/N datum each, VQ can be used to encode the data from each filter independently. Westerink *et al.*[41] used the subband coder to filter the image into 16 subbands, and used a 16-D vector quantizer where each vector contained the same pixel location from all 16 subbands.

Predictive VQ (PVQ): Memoryless VQ can efficiently exploit the intra-vector correlation, but cannot utilize the inter-vector redundancy. PVQ schemes belong to a class of VQ with memory and exploit the inter-vector correlation. The PVQ coding system employs a predictor followed by vector quantizer [22,23]. The input vector is predicted from the previous transmitted vectors. The error vector is formed by subtracting the predicted vector from the current input vector which is then vector quantized.

Finite state VQ: Another VQ approach with memory which reduces the bit rate is the finite state VQ [24,42]. A Finite state VQ encoder consists of a state transition function f and a set of finite states. A small subcodebook is associated with each state. The function f uses the previous

encoded vectors to determine the current state of the encoder. The input vector is quantized using the subcodebook corresponding to the current state. For example, Kim [42] has proposed a Finite state VQ technique where the state transition function f is based upon the gray level transition across the boundaries of the neighbouring blocks.

Mean/Residual VQ (M/RVQ): In this coding system [43,44], the sample mean is first scalar quantized and then subtracted from the vector so that quantization error can be incorporated into the error vector. This technique is derived from the Mean/Shape (M/S) VQ, where the sample mean is scalar quantized and the resulting error vector, obtained by subtracting the sample mean from the input vector, is vector quantized [43]. The M/RVQ system reduces the blocking distortion that is caused by coarse quantization of the sample mean in the M/SVQ system. The block diagram of M/RVQ is shown in Figure 2.5.

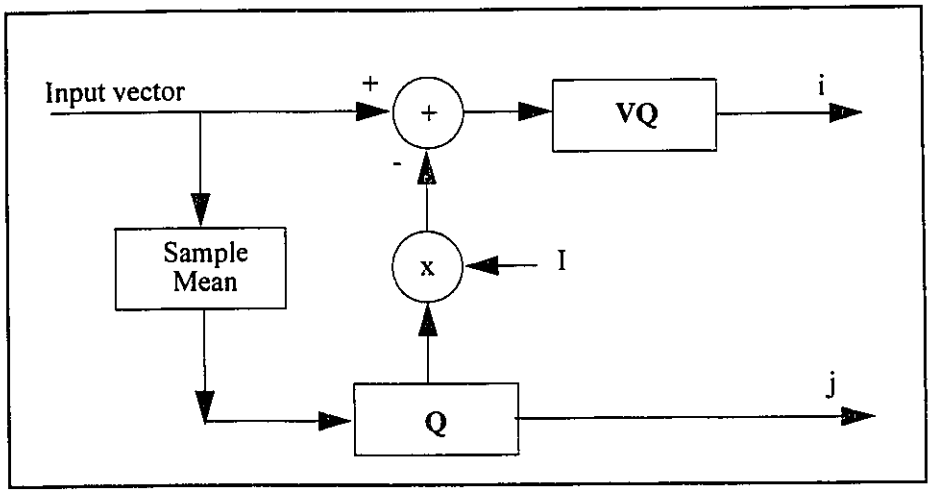

FIGURE 2.5 Block diagram of Mean / Residual VQ

Multi-stage VQ [45-49]: This coding technique is an distinctive extension of Mean/ Residual VQ. The multi-stage VQ has stage-by-stage design procedure. The codebook of each stage is optimized for that particular stage distortion and does not consider the distortion of the subsequent stage.

The input vector V_i is first vector quantized using the N_1-level quantizer and the error vector is formed. The error vector is then quantized using the N_2-level quantizer. The process is repeated

by finding the error vector from the second stage and feeding it to the third stage and so on. The final quantizer version of V_i is the sum of all reproduction vectors. The computational and storage costs are $O(KL\sum N_i)$ and $O(L\sum N_i)$, respectively.

Several variations of multi-stage VQ have been reported in the literature. For example, Wang *et al.*[46] have proposed a multi-stage VQ technique applied to the transform coefficients with optimal bit allocation strategy. Koessentini *et al.*[47] have presented a variable rate multi-stage VQ technique. Chan *et al.*[48] have proposed a joint design algorithm of the stage codebook to optimize the performance of multi-stage VQ. Rizvi *et al.*[49] have presented a codebook design procedure based on a multi-layer competitive neural network where each layer of the network represents one stage of multi-stage VQ.

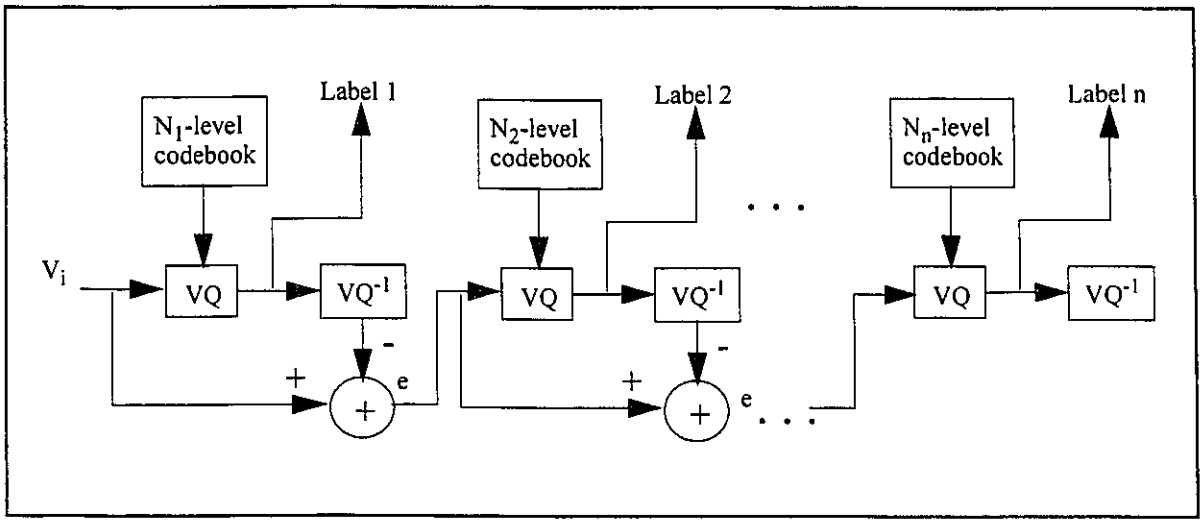

FIGURE 2.6 Block diagram of Multi-stage VQ

Address VQ: The address codebook consists of a set of address-codevectors [50]. Each address - codevector constitutes the addresses (labels) corresponding to four codewords in the LBG codebook. In the encoding process, the four neighbouring blocks are coded using the LBG codebook to form the address-codevector. The address codebooks at both the transmitter and the receiver are re-organized, using the block transition probability matrices, in order to maintain the most probable address-codevectors in the active address-codebook. The active address-codebook is then searched for a matched address-codevector. If a match is obtained, the index corresponding

matched address-codevector is transmitted. Otherwise, four labels are transmitted independently to the receiver.

An extension of address VQ is the multilayer address VQ [51], where the input image is encoded at the first layer using LBG codebook. The resulting labels from the first layer are merged together in the second layer to form a new address-codevector. This process of merging is repeated recursively in a hierarchically bottom-up fashion. At each layer the address-codebook consists of the most probable address combinations. Each address-codevector is encoded starting from the top layer, using the corresponding address-codebook similar to address VQ.

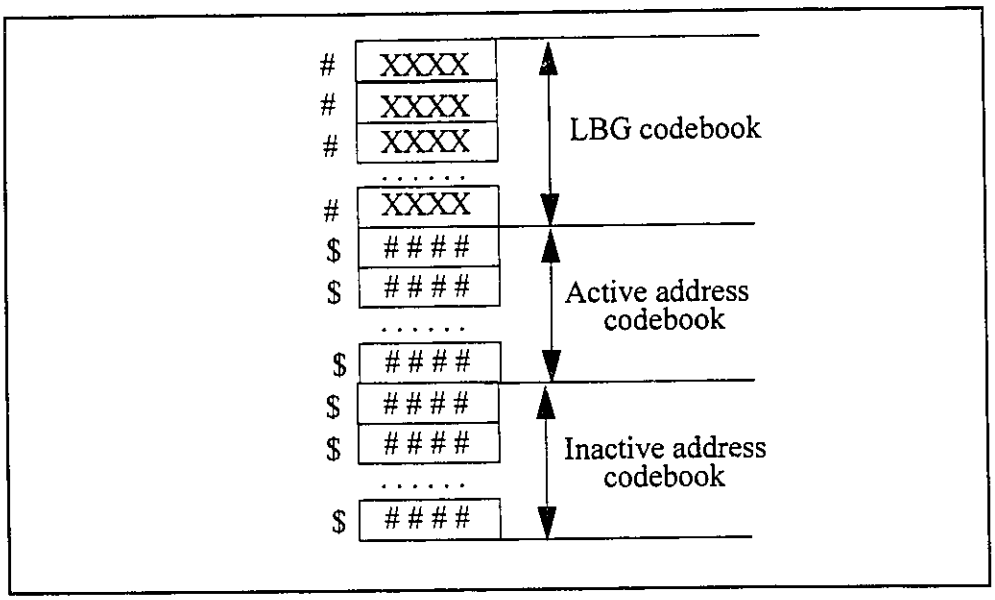

FIGURE 2.7 Block diagram of Address codebook VQ

Tree-structured VQ (TSVQ): The computations necessary for finding the matched codeword can be reduced by structuring the codebook as a binary tree [53]. The tree is constructed with a code-word at each node, such that the first layer of the tree divides the L-dimensional space into two regions, the second layer divides the L-dimensional space into four regions, and so on. Only the codewords at the leaves are used for encoding. In quantizing each vector, a decision is made between one of the two branches at each layer, and the matched codeword is found at the last

level. In comparison with the full search VQ, the computational complexity is reduced to O(KL-logN) at the expense of almost doubling the storage cost.

The K-dimensional (K-d) tree is a generalization of the binary tree and it provides a data structure which allows for multi-dimensional search. Recently, Equitz [54] and Ramasubramanian *et al.* [55] have proposed fast search algorithms for VQ encoding using K-d tree structure.

The performance of TSVQ will in general have some degradations compared to full search VQ. To improve the coding performance of TSVQ, Riskin *et al.*[56] have presented an unbalanced TSVQ where the tree is designed one node at a time rather than a layer at a time.

Hierarchical VQ: In this coding technique, a quad-tree algorithm [58] is first used to partion the image into blocks of size 2x2, 4x4, 8x8, and 16x16. This information about partitioning the image is represented by a quad-tree and is counted as the side information to be transmitted. In the coding process, the small blocks, 2x2 and 4x4, are coded by using full search VQ. The larger blocks representing constant patterns are coded by applying VQ in the transform domain. Many of the high frequency coefficients can be discarded and thus the effective dimension of the blocks is reduced for computational purposes.

Vector Quantization using Adaptive Codebook

The universal codebook has two shortcomings which may degrade performance. First, in order to ensure a good fit for all images, a large codebook is needed and this increases the bit rate. Secondly, images outside the training set may not always be well represented. Therefore, codebook adaptation to the local image statistics may require in order to improve image quality. For example, a new codebook can be generated using the vectors of the input image as a training set. The new codebook is transmitted, followed by the labels corresponding to the vectors of the image. Goldberg *et al.*[60] have presented an adaptive VQ scheme for coding monochrome and color images. In this scheme, the image is partitioned into non-overlapping subimages, and for each subimage a separate 16 -64 level codebook is created. These codebooks better match the local

image statistics, however, the improvement in coding performance is achieved at the expense of the overhead incurred for transmitting the new codewords. Zeger *et al.* [61] have proposed a technique where a new codebook is designed for each input image. The new codebook is vector quantized using a large universal codebook; hence, reducing the overhead for transmitting the new codebook. An alternative scheme, which reduces the overhead, is to update or replenish part of the codebook. For example, Gersho *et al.* [62] have proposed a technique where the distortion of each input vector is monitored, and if it is larger than a predetermined threshold, the codeword with the "largest time since use" is replaced by the input vector. Yeh [63] introduced a technique where the codewords that cause excess quantization error are replaced. Note that in the above techniques, the major drawback is that the improvement in image quality is achieved at the expense of increasing the computational complexity.

Panchanathan *et al.* [64] have presented a reduced codebook design algorithm for adaptive VQ. Here, a large universal codebook of size $\bar{N}$ is employed to encode the input vectors. A reduced codebook of size N consisting of only the used codewords in the universal codebook is then formed. Since $N << \bar{N}$, this technique results in lower bit rate for label coding. In another variation of this scheme, the input vectors are used as a training set and one iteration of LBG algorithm, using the reduced codebook as the initial codebook, is performed. The changes to the codewords are then transmitted. Panchanathan *et al.*[65] have also presented a dynamic VQ scheme where the codebook is generated on the fly from the input vectors to be coded. In this scheme, a small primary codebook is used to store the most frequently used codewords, and a larger secondary codebook is used to store the less frequently used codewords. Both the transmitter and the receiver keep track of identical codebooks without any overhead.

Bit Rate Calculation

Let X, Y be the dimensions of the image, K the number of vectors in the image, N the number of codewords in the codebook, and L the number of pixels in each codeword (or the length of the

codeword vector). In the universial VQ, only the labels are transmitted, hence, the bit rate in bits per pixel (bpp) is:

$$BR = \frac{\log_2 N}{L} \tag{2.17}$$

In the case of image adaptive VQ [60-65], the codebook and the labels are transmitted. The total bit rate now includes the number of transmitted codewords N_c, with each vector having B bits. Therefore:

$$BR = \frac{\log_2 N}{L} + \frac{N_c \times B}{K} \tag{2.18}$$

2.3 Video Compression Techniques

We recall from section 2.1 that compression of the video sequence is achieved by removing the spatial and temporal correlations using intraframe and interframe coding techniques, respectively. In this section, a review of video compression techniques is presented.

2.3.1 Motion Estimation / Compensation

Motion estimation/compensation is a widely used technique to exploit the temporal correlation in a video sequence. Motion estimation techniques attempt to obtain the motion information for the various objects in the scene. In principle, if the motion information of the object is known, then high compression could be achieved by coding the first frame together with the motion information. There are two classes of motion estimation/compensation algorithms: pel-recursive [66] and block matching (BMA's) [67]. Pel-recursive algorithms evaluate the displacement of each pixel individually. These algorithms do not require the transmission of motion information but recursively use the luminance change to find the motion information. On the other hand, block matching algorithms assume that all pixels within a block have the same motion. The motion vectors for a group of pixels are calculated and transmitted to the receiver.

A popular method for block matching is the full search block matching algorithm (FBMA). FBMA searches all possible displaced locations within a search area (SA) to find the best match. Several matching criteria have been proposed to find the best match block [68]. These include: the MSE, the mean absolute difference (MAD), etc. The MAD criterion is preferred because of its lower computational complexity. MAD is given by:

$$MAD_{(k,l)}(X,Y) = \frac{1}{n \times n} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \left| X_{i,j} - Y_{i+k,j+l} \right| \qquad -p \le k, l \le p \tag{2.20}$$

where $X_{i,j}$ is the reference block data at coordinates (i,j) and $Y_{i+k,j+l}$ is the candidate block data at coordinates (i+k, j+l).

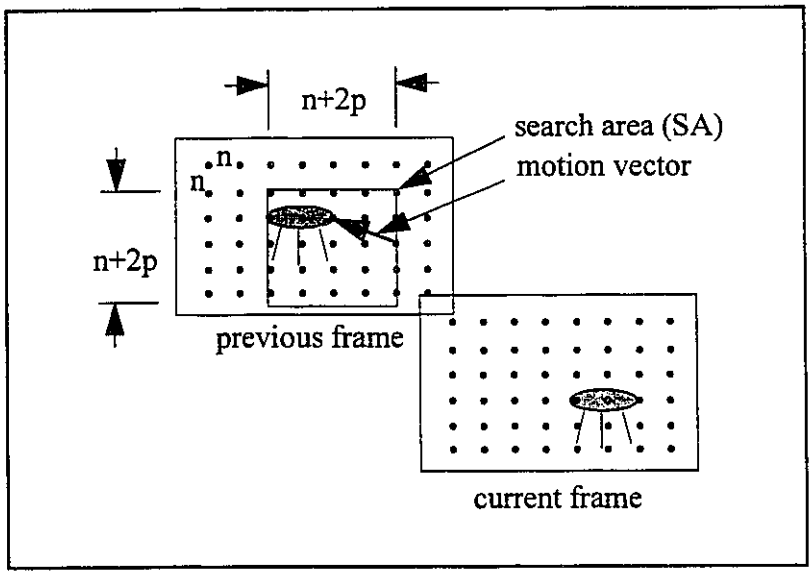

FIGURE 2.8 Block matching process

Figure 2.8 illustrates the block matching process between adjacent frames. The current frame (F) is divided into non-overlapping reference blocks of size n x n. Each reference block in (F), is compared with the candidate blocks within a SA in the previous frame (F-1) in order to obtain the best match. The size of SA is $(n+2p)^2$, where p is the maximum displacement, and is centered around the candidate block with identical coordinates as the reference block. The offset between the reference block and the best match block specifies the motion (displacement) vector.

For each of the $(2p+1)^2$ locations to be searched, the calculation of MAD requires $3n^2$ operations (that is subtraction, absolute operation, and addition). Hence for K input blocks, the computational complexity of FBMA is $O(3Kn^2(2p+1)^2)$. For example, a 512 x 512 image with a block size of 4 x 4 and a maximum displacement of 4 pixels requires approximately 63 million operations. The corresponding values for a displacement of 6 and 8 pixels are 132 and 225 million operations, respectively. In order to reduce the computational complexity, a number of fast search algorithms such as 2-dimensional logarithmic search, three-step hierarchical search [67,69] have been proprosed. These algorithms reduce the number of computations by limiting the number of candidate blocks. However, for hardware realization, the FBMA offers the advantage of low control overhead since it has regular data flow and fixed number of operation setps. We note that motion estimation techniques are typically combined with image compression techniques to remove spatial correlations.

2.3.2 Predictive Coding with Motion Compensation

Motion compensated predictive coders are a class of adaptive predictive coding systems that take into account the motion of the objects in the scene in order to reduce the prediction error. For example, if an object in a moving scene has a spatial translation of (Δi, Δj), then one expects that a pixel $x_{i,j,F}$ (belonging to the moving object) is maximally correlated with the pixel $\hat{x}_{i+\Delta i, j+\Delta j, F-1}$ rather than the pixel $\hat{x}_{i,j,F-1}$. A motion compensated adaptive coding scheme is shown in Figure 2.9. The motion vector (Δi, Δj) of each pixel $x_{i,j,F}$ is first estimated using a matching procedure, as described in section 2.3.1, which determines the displacement of each pixel such that the difference between the current pixel and the predicted pixel is minimized. The prediction error $x_{i,j,F} - \hat{x}_{i+\Delta, j+\Delta, F-1}$ and the displacement vector are then coded separately and transmitted to the receiver.

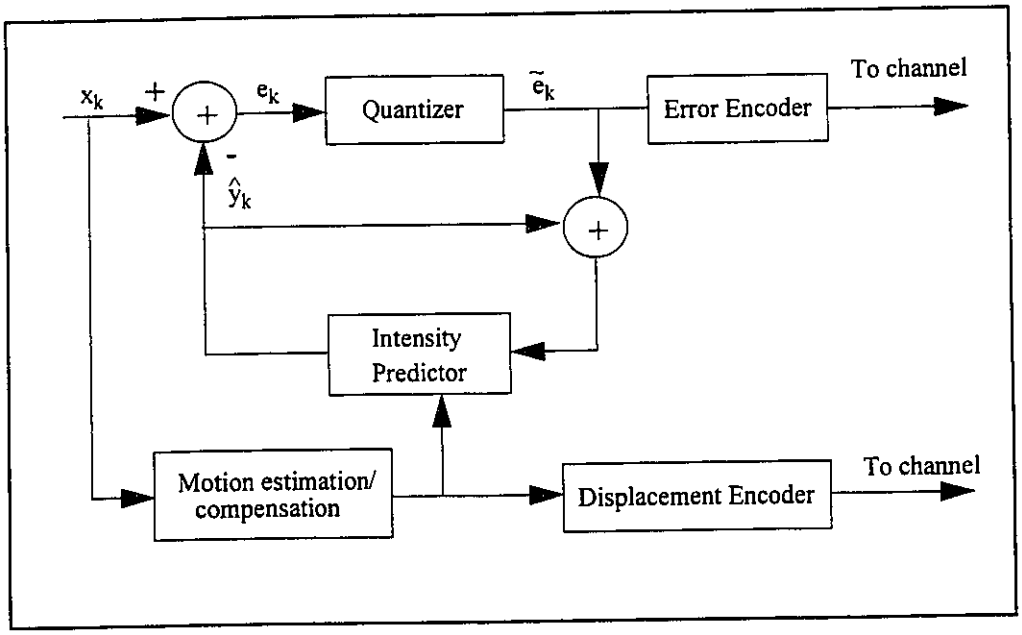

FIGURE 2.9 Block diagram of Motion compensated DPCM coder

2.3.3 Conditional Replenishment

Conditional replenishment technique [70] is based on dividing each frame into stationary and non-stationary parts. Only the changed parts are coded. If $x_{i,j,F}$ is the pixel at location (i,j) in the current frame (F), then the predicted value $x_{i,j,F}$ is the reconstructed value $\hat{x}_{i,j,F-1}$ at the same spatial location in the previous frame (F-1). The prediction error in this case is calculated using:

$$e_{i,j,F} = x_{i,j,F} - \hat{x}_{i,j,F-1} \tag{2.20}$$

If the magnitude of $e_{i,j,F}$ is greater than a prespecified threshold, then it is quantized, coded, and transmitted along with the location (i,j) of $x_{i,j,F}$.

A number of algorithms based on the basic conditional replenishment technique have been reported in the literature. For example, instead of coding the address of each changed pixel separately, the addresses and the quantized errors are coded in clusters along a line. Another variation is to change the temporal resolution in stationary parts and the spatial resolution in the moving parts in order to obtain further reductions in bit rate without affecting the subjective quality.

2.3.4 Interframe Transform Coding

Interframe transform coding exploits the temporal correlations between succesive frames as well as the spatial correlations within the individual frames. A sequence of frames is partitioned into three dimensional blocks $X_{m1,m2,t}$ of M1 x M2 x T pixels. Each block then undergoes a three dimensional transform according to the following general formula:

$$Q_{u,v,w} = \sum_{m1=0}^{M1-1} \sum_{m2=0}^{M2-1} \sum_{t=0}^{T-1} X_{m1,m2,t} A_{u,v,w;m1,m2,t} \tag{2.21}$$

to generate the transform coefficients $Q_{u,v,w}$, where $A_{u,v,w;m1,m2,t}$ is the transform kernel and M1, M2, and T denoted the row, column, and temporal indices, respectively. The transform coefficients are then quantized, coded and transmitted to the receiver. To reconstruct the encoded image sequence, the receiver applies an inverse three dimensional transform on the quantized coefficients $\tilde{Q}_{u,v,w}$:

$$\hat{X}_{m1,m2,t} = \sum_{u=0}^{M1} \sum_{v=0}^{M2} \sum_{t=0}^{T} \tilde{Q}_{u,v,w} A^{-1}_{u,v,w,m1,m2,t} \tag{2.22}$$

where $A^{-1}_{u,v,w,m1,m2,t}$ is the inverse three dimensional transform kernel.

In image/video sequences the statistics along the temporal dimension may vary significantly, therefore, adaptive techniques can substantially improve the coding performance [13]. For example, the effective number of bits assigned to each coefficient can be made proportional to the coefficient variance [71].

2.3.5 Hybrid Interframe Transform / Predictive Image Coding

Interframe hybrid coding combines transform and interframe predictive coding. Typically, each frame is partitioned into two dimensional blocks, a transform is executed on the resulting block, the coefficients are then predicted from the quantized coefficients along the temporal direction.

2.3.6 Interframe VQ

Several image/video sequence coding algorithms based on VQ have been reported in the literature. The first class of algorithms uses a fixed VQ codebook. For example, Murakami *et al.*[25] have presented a vector quantizer for video signals where a fixed codebook is generated using a long training sequence of normalized (by mean and standard deviation) vectors. The mean and standard deviation values are transmitted as side information along with the label. Guo *et al.*[72] have proposed a technique in which the difference between the input frame and the predicted frame is vector quantized. The difference frame is divided into 16-dimensional vectors. For each vector, the directional conditional vector probability matrices are used to select a sub-codebook from a larger codebook. The input vector is then encoded using the sub-codebook or larger codebook based on which codeword results in a lower distortion. Corte-real *et al.* [73] have proposed an image sequence vector quantization scheme where difference frame is divided into blocks of different shapes and sizes. Several vector quantizers are then used to encode the resulting blocks. Nasrabadi *et al.* [74] have proposed an interframe hierarchical address - vector quantizer (IHA-VQ) using quad-tree decomposition, where each frame is partitioned into 7x7 blocks. Block matching [67] is used to estimate the motion of each block. To indicate the status (motion/no motion) of the blocks, a bit map followed by the motion vectors of the moving blocks are transmitted. The difference between the current frame and the motion estimated frame is then segmented hierarchically into 32x32, 16x16, 8x8, 4x4, and 2x2 blocks. The 32x32, 16x16, and 8x8 blocks are replenished from the previous frame whereas a set of codebooks is used to encode the smaller blocks. Recently, Li *et al.*[75] have reported a fast quad-tree motion segmentation algorithm for VQ coding of video sequence coding. The frame is partioned into blocks of different size according to its activity. Small blocks containing high motion activities are intraframe vector quantized, whereas large blocks representing smooth areas are first decimated and then interframe vector quantized. There are two problems with this class of algorithms. First, a limited size codebook is not sufficient to represent the different types of image sequences. Therefore, a large codebook is required which increases the bit rate and entails practical difficulties in codebook

generation and encoding process. Secondly, the mismatch between the codebook and image sequences outside the training set may degrade the coding performance.

The second class of algorithms is based on the codebook updating technique during the encoding process so that the local statistics of the frame can be tracked. For example, Goldberg *el al.*[76,77] have presented several interframe coding techniques where changes in successive frames are tracked by incorporating *label replenishment* (LR) and *codebook replenishment* (CR). Here, the vectors of the first frame are used to generate the initial codebook. The frame is vector quantized, and the corresponding labels are stored in the label memory. The vectors in the subsequent frames are vector quantized, and the labels are compared with the corresponding labels in the label memory of the previous frame. If they differ, the label memory is replenished. We note that, in the LR technique, the label in the current frame is only compared to the corresponding label in the previous frame, without considering the neighbouring labels. Therefore, it does not fully exploit the temporal redundancy of the sequence.

In order to adapt the codebook to the local frame statistics, codebook replenishment is used where the mean of the new clusters is determined using the vectors in the current frame (MS-VQ). The new means are then compared with the corresponding codewords. If the difference is larger than a pre-specified threshold, the codeword is replaced by the corresponding new mean. Another approach to replenish the codebook is to design a new codebook for each frame using the vectors of the current frame as a training set and the codewords of the previous codebook as the initial seeds. The resulting centroids are then compared to the codewords in the initial codebook. If the difference is greater than a threshold, the codeword is replenished. Moreover, the nearly empty clusters are replaced by the more abundant ones. We use the notation SCR-VQ to refer to this technique. Monet *et al.*[78] have proposed the application of codebook replenishment technique for classified pruned tree structured VQ. This technique uses the average distortion resulting from the assignment of a set of input vectors to a particular codeword as a measure to determine whether to delete, split or modify the codeword. Yeh [17] has presented an adaptive VQ technique

for color image sequence coding. In this technique, a mean/residual binary tree VQ is employed where the quantization error is used to determine whether to replenish the codeword or not. Braccini *et al.* [57] have proposed a TSVQ technique where codebook is organized in a tree-structure and capable of modifying its topology and contents in order to adaptively track the input statistics. The resulting tree structure is unbalanced depending on the input vector distribution. Idris *et al.* [59] have presented a frame adaptive VQ which uses two modes of operations: interframe and intraframe. In the interframe mode, the input vector is compared with the corresponding vector at the same spatial location in the previous frame, or with a SA for motion estimation. If no match is found, the controller switches to intraframe mode. In the intraframe mode, two codebooks are used: primary codebook (PC) and secondary codebook (SC) which dynamically replenish using the LRU criteria. This implementation requires the synchronization at both the transmitter and receiver to update the two codebooks. Generally, frame adaptive techniques result in further increase in computational complexity.

Recently, Le *et al.*[80] have proposed a combined adaptive VQ and index-based motion estimation technique (AVQ+ME) for intraframe and interframe coding, respectively. In the AVQ method, the codewords of the current frame are used as seeds to generate the codewords for the next frame. In the ME method, the label of the current frame is compared to a search area (SA) of the corresponding labels in the previous frame and the resulting motion vector is transmitted. Huffman coded flags are required to differentiate among the different components. The AVQ+ME technique has been implemented on SIMD architecture. Details of the technique and the corresponding implementation is presented in chapter 6.

2.4 Image / Video Compression Standards

Recently, the International Standards Organization (ISO) has proposed standards for still image and video compression known as the JPEG (Joint Photographic Experts Group) [81-82,86] and the MPEG (Moving Picture Experts Group) [83-84,86]. In addition, the Consultative Committee

on International Telephony and Telegraphy (CCITT) has recommended a standard for videotelephony called the H.261 [85-86] at p x 64Kbits/s.

2.4.1 JPEG Standard for Image Compression

JPEG is jointly developed by ISO and ITU-TS and became standard in 1992. The JPEG standard can compress still images by a factor of 10 to 50 without visibly affecting image quality. The JPEG standards supports one of the following four modes:

- **Baseline sequential mode**: provides the basic feature of (DCT-based) compression for full quality and full size image reconstruction;
- **Lossless mode**: is targeted towards applications, such as medical imaging, which require perfect image reconstruction;
- **Progressive mode**: Image encoded in multiple scans, from coarse to finer, progressively improved by transmission over a low bandwidth transmission channel. As a result, this mode is used for implementing SNR scalability;
- **Hierarchical mode**: In this mode, the image is encoded at multiple resolutions, so that lower-resolution images can be obtained without having to decompress the entire compressed data. As a result, this mode is used for implementing spatial scalability.

The baseline sequential mode is chosen for review. In the baseline sequential mode, samples are encoded using 8 bits with values ranging from 0 to 255. The compression is performed in 3 steps: DCT computation, quantization, and variable length coding (Figure 2.10). The original image is first partitioned into non-overlapping blocks of 8x8 pixels. The pixel values are shifted into the range from -128 to 127 with zero as center. The DCT of the block is then computed and quantized using a quantization table suggested by JPEG. The DCT computation results in 64 coefficients. The first zig-zag scanned coefficients is the DC coefficient, the rest are AC coefficients. The DC coefficient is DPCM coded and the AC coefficients are coded using a combination of Huffman and run-length coding. The decoding scheme is just the inverse of the encoding scheme and thus is not shown here.

JPEG applications do not have to include both an encoder and a decoder. The encoded data stream has a fixed interchange format that includes the encoded image data as well as the chosen parameters and the tables of the coding process. If the compression and decompression process agree on a common set of coding tables to be used, then they need not be included in the data stream. This convention is similar to Universal VQ where the codebook is the same at both the transmitter and receiver and therefore is excluded in the data stream.

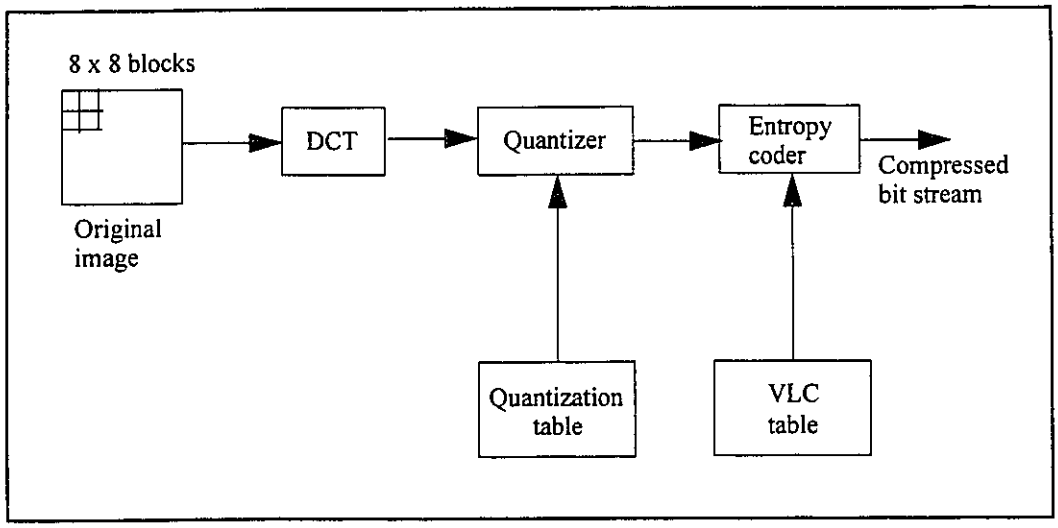

FIGURE 2.10 JPEG baseline sequential mode

2.4.2 H.261 standard for Video Compression

CCITT H.261 [85] was designed for conferencing systems and video telephony using the existing ISDN network. Due to the real-time requirement, the maximum delay of both compression and decompression must not exceed 150ms. The standard encodes video at a *constant bit rate* (CBR) of p x 64Kbps, where p=1, 2, ..., 30 by using DCT and motion compensation techniques. The coder output rate is kept constant for transmission over a CBR channel. This is accomplished by using quantization step control based on output buffer fullness.

Unlike JPEG, H.261 defines a very precise image format. The image refresh frequency at the input must be 29.97 frames/s. During encoding it is possible to generate a compressed image sequence with a lower frame rate of 10 or 15 still images per second. Only non-interlace images

are allowed at the input of the coder. The image is encoded as luminance signal (Y) and chrominance difference signals C_b, C_r according to CCIR 601 subsampling scheme (2:1:1). In H.261 data units of the size 8 x 8 pixels are used for the presentation of the Y as well as the C_b and C_r components. A macro block is the result of combining four blocks of the Y matrix each with one block of the C_b and the C_r component.

Two resolution formats each with an aspect ratio 4:3 are specified, the so-called Common Intermediate Format (CIF) defines a luminance component of 288 lines, each with 352 pixels. The chrominance components have a resolution with a rate of 144 lines and 176 pixels per line to fullfil the 2:1:1 requirement. Quater CIF (QCIF) has exactly half the CIF resolution, i.e., 176 x 144 pixels for the luminance and 88 x 72 pixels for the other components. All H.261 implementations must be able to encode and decode QCIF; CIF is optional.

The H.261 standard uses two different ways of coding: intraframe and interframe. In the case of intraframe coding, no advantage is taken from the redundancy between frames. This coding technique corresponds to the I frame coding of MPEG. For interframe coding, information from previous or subsequent frames is used; this corresponds to the P frame encoding of MPEG.

Similar to JPEG, for intraframe coding, each block of 8 x 8 pixels is transformed into 64 coefficients by a DCT. The quantization of the DC coefficients differs from the quantization of the AC coefficients. The next step is to apply entropy coding to the AC and DC parameters, resulting in a variable length encoded word. Interframe coding is based on a prediction for each macro block of an image. This is determined by a "comparison" of macro blocks from previous frames with the current frame. The motion vector is defined by the relative position of the previous macro block with respect to the current macro block. Note that according to H.261, the coder need not be able to detemine a motion vector. Therefore, a simple H.261 implementation considers only the differences between macro blocks located at the same position of consecutive images. In such cases, the motion vector is always a zero vector.

Subsequently the motion vector and the DPCM coded macro block are processed. The DPCM coded macro block is transformed by a DCT if and only if its value exceeds a certain threshold. If the difference is less than this threshold, the corresponding macro block is not encoded any further; only the relevant motion vector are entropy coded with a variable length coding that is lossless. All of the transformed coefficients are quantized linearly and variable length coded.

For H.261 the quantization is a linear funtion and the step size is dependent on the amount of data in the transform buffer. This mechanism enforces a constant data rate at the output of the coder. Therefore, the quality of the encoded video data depends on the contents of individual images as well as on the motion within the respective video scene.

2.4.3 MPEG Standards for Video Compression

Many applications have been proposed based on the assumption that an acceptable quality of video can be obtained data bandwidth of about 1.5Mbits/second (including audio). In the MPEG video compression standard, a block-based motion estimation/compensation is employed to remove the interframe correlation and DCT for the reduction of the intraframe correlation. Block diagrams of the MPEG encoder and decoder are shown in Figures 2.11 and 2.12, respectively.

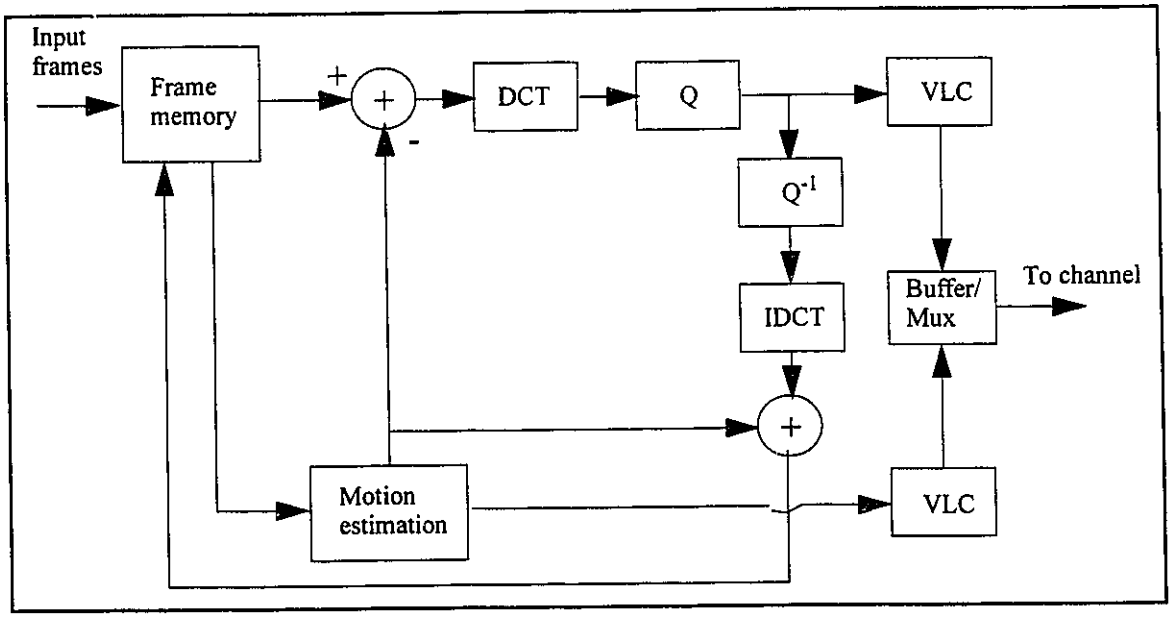

FIGURE 2.11 Block diagram of MPEG encoder

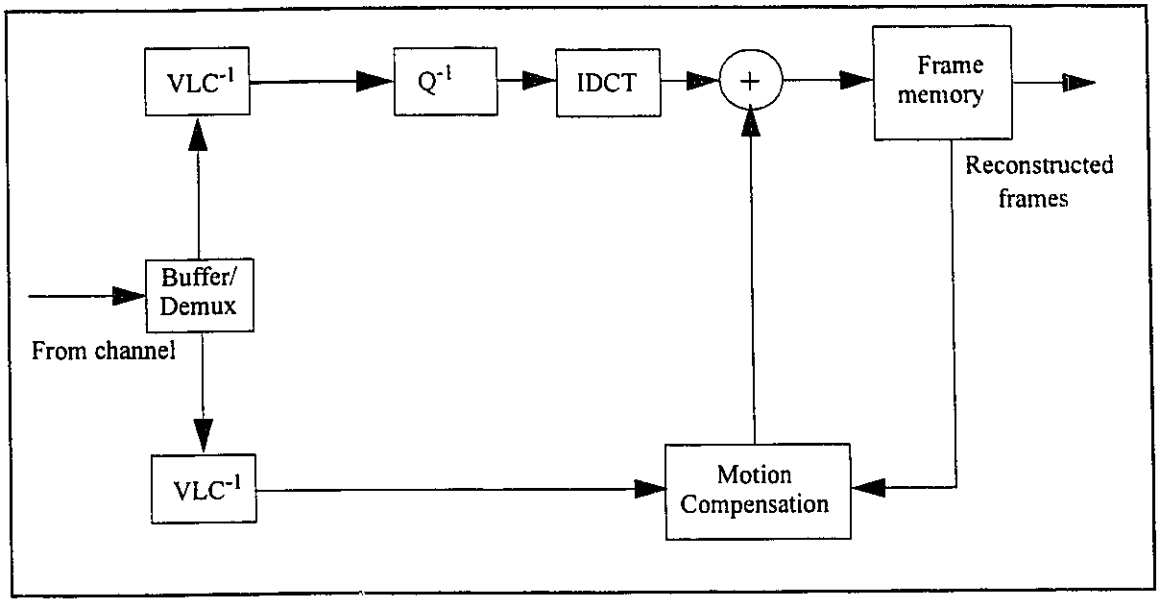

FIGURE 2.12 Block diagram of MPEG decoder

Here, a *group of pictures* (GOP) approach is used instead of frame by frame coding. A GOP is typically a combination of one or two *intra-picture* (I), *predicted picture* (P), and the rest of *bidirectional pictures* (B) as shown in Figure 2.13. The I frames (picture) provide random access points and are also used as a reference for P frames. The I frames are coded using DCT on 8x8 blocks. The DC coefficient is differentially encoded using variable length codes (VLC). The AC coefficients are zig-zag scanned and ordered into {RUNLENGTH, AMPLITUDE} pairs. A variable length coder is used to encode each pair.

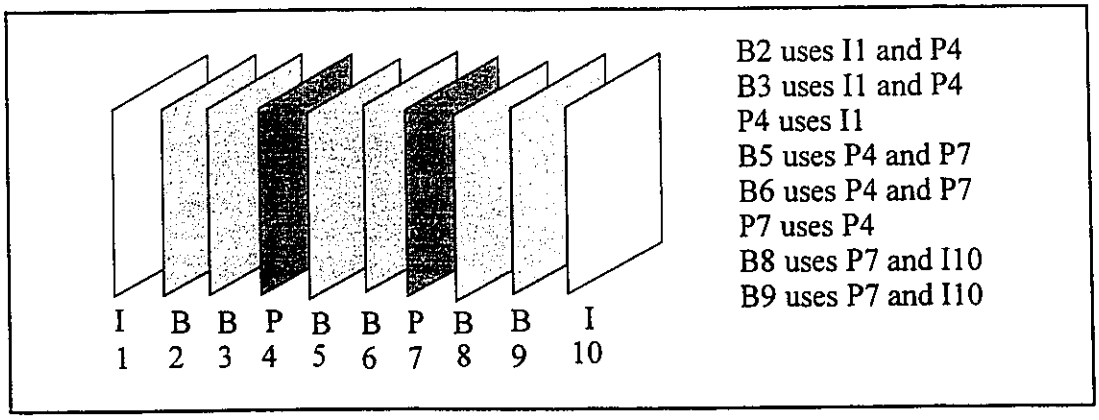

FIGURE 2.13 An example of GOP used in MPEG

The P and B frames are decomposed into 16x16 blocks and the motion vector for each block is calculated. The motion compensated difference frame is partitioned into 8x8 blocks which then undergo a 2-dimensional DCT. The DC and AC coefficients are quantized, ordered along the zig-zag scan line into {RUNLENGTH, AMPLITUDE} pairs and coded using a VLC. The motion vectors are also variable length coded and transmitted.

The features of the still image and video compression standards are summarized in Table 2.1

TABLE 2.1 Summary of the image/video compression standards

Standard	Applications	Intraframe coding	Interframe coding	Compression Ratio
JPEG	still images	DCT	n/a	1:10 to 1:50
H.261	videophone / teleconference	DCT / I frame	P frame	1:10 to 1:25
MPEG	video / games	DCT / I frame	GOP(I,P,B)	1:100

2.5 VLSI Architectures for Image / Video Compression

2.5.1 Why Architectures ?

Image processing algorithms can be classified according to the complexity into low, medium, and high level algorithms [87-88]. Low level algorithms usually scan their numeric input data regularly. Processing is periodically defined over an image segment. Segments may overlap, or may be arranged in non-overlapping order, as this is the case with most transform algorithms. Usually, the segments have the shape of a square or a vector of pixels, the smallest segment is a single pixel. An example for a low level algorithm including decisions is the binary thresholding of gray level image, setting all pixels below a specific gray threshold as "black" and all pixels above as "white". Input and output data are square images, segmented in pixels and a simple, data dependent decision is carried out.

Medium level algorithms also scan their input data regularly. They frequently process the output data of a preceeding low level algorithm. Size and complexity of these data spaces strongly depend on the algorithm, its implementation, and its parameters. A typical medium level algo-

rithm is the image histogram operation. The input data are scanned pixel by pixel, output data are a list of gray level counts. The counter array is accessed irregularly depending on the gray level of each processed pixel. For an image histogram, the size of the array depends on the number of possible gray levels and is usually significantly smaller than the amount of input data. For other algorithms, such as the Hough transform, intermediate data space may exceed the amount of input data significantly.

High level algorithms process symbolic data structures, such as lists, arrays or trees for control and analysis purposes. The size of structures is usually small, but decision processes performed on it are rather complex.

It can be calculated that if the uncompressed color video sequence is sampled at a frame rate of 30 frames per second, where each frame is of size 288 x 352 pixels in the Common Intermediate Format (CIF). If each pixel is represented using 24 bits, the data rate is 73 Mbits/second. In order to handle the high computational overload, different types of computer architectures have been applied to image/video processing applications. However, no single architecture can solve all possible problems. For instance, array processors are well suited for low level algorithms, however, they are not efficient for high level algorithms because of their limited control. Multiprocessor systems are best fit for high level algorithms, but they require expensive support for input/output, programing, and interprocessor communications. In general, it has been realized that programmable architectures are best suited for image processing applications.

2.5.2 Review of Architectures for VQ-based Algorithms

In classifying the architectures for VQ-based algorithms, we adopt the Flynn's convention [89] according to the instruction and data streams. Basically, by employing P processors which can be operated concurrently, a speed up factor as high as P can be achieved. There are four main types:

- SISD (Single-Instruction stream, Single-Data stream) structure: This is a von Neumann computer, in which a single instruction and datum are fetched from memory, processed, and the

result written back. In other words, this is the uniprocessor model where one processor executes all the tasks sequentially under the control of a single control unit;

- SIMD (Single-Instruction stream, Multiple-Data stream) structure: In this structure, there are multiple processing elements under the supervision of the same control unit. All PE's receive the same instruction broadcast from the control unit but operate on different data sets from their own data streams, examples include vector processors, array processors.
- MISD (Multiple-Instruction stream, Single-Data stream) structure: This category consists of machines in which a set of processors operates on a single stream of data. A pipeline processor is a good example of this type of system; and
- MIMD (Multiple-Instruction stream, Multiple-Data stream) structure: MIMD is characterized by multiple PE's executing different instructions on their data streams.

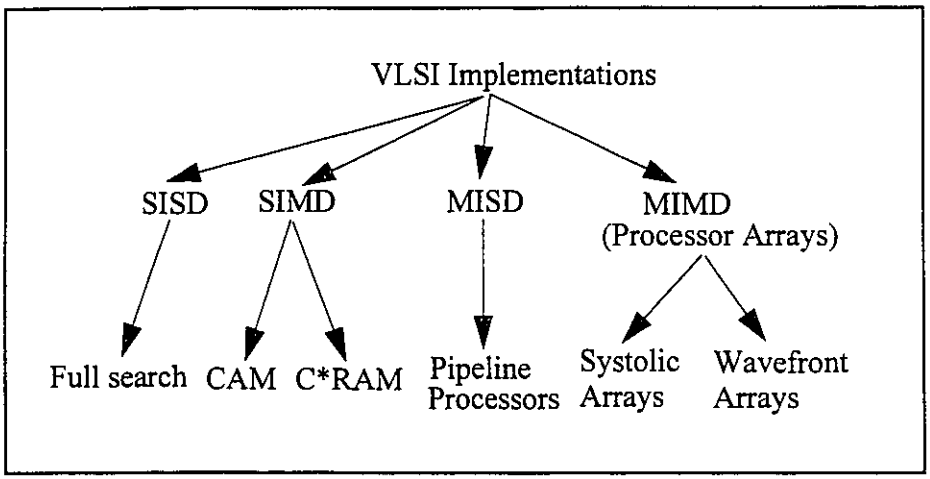

FIGURE 2.14 VLSI Implementations of VQ techniques

Based on this convention, the VLSI architectures for implementing VQ-algorithms can be classified as shown in Figure 2.13.

Full search method: The first approach is to use a fast processor with a high clock rate to execute the elemental operation. This yield a speed up of:

$$S_p = T_{SISD}/T_e \tag{2.24}$$

where T_{SISD} and T_e are the time to compute the elemental operation using an SISD machine and a fast processor, respectively.

Tao *et al.*[90] have presented the VQ implementation using a fast processor. In this implementation, there is no task optimization and the encoding process is done sequentially. This architecture uses high speed memories (CMOS), low power Schottky logic (LSTTL) and a table look-up for the squaring operation. The elemental operation is executed in 500ns. However, the speed up achieved is not large enough for image processing applications.

Content-Addressable Memory: Panchanathan *et al.*[91] have proposed a content-addressable memory (CAM) for adaptive image coding. The codebook is stored in the CAM array and the search process is carried out in parallel from the perspective of the codewords, hence exploiting parallelism in the directions of N and L. The speed up factor is given by:

$$S_p = \frac{T_{SISD}}{T_e} \times N \times L \tag{2.24}$$

Abut *et al.*[93] have reported on a fuzzy associative memory (FAM) which provides an increase in speed up by exploiting parallelism in the direction of L and processing multiple input vectors in parallel.

Computational*RAM: C*RAM is a SIMD-memory hybrid struture where a PE is attached to each memory column for bit-serial and word-parallel computation. Le *et al.*[79-80] have proposed a C*RAM implementation of adaptive/non-adaptive VQ algorithms in which the codewords are stored in C*RAM and the search is carried out parallel in the direction of N. Due to the bit serial computation of the C*RAM, the speed up is:

$$S_p = \frac{T_{SISD}}{T_e} \times N \tag{2.25}$$

Pipeline Processor: Here, each segment of the pipeline executes a portion of the elemental operation. This results in a speed up factor of:

$$S_p = \frac{T_{SISD}}{T_e} \times P \tag{2.27}$$

where P is the number of segments in the pipeline.

Davidson *et al.*[95] have reported on the design of a high speed codebook search processor (CSP) organized in a semi-custom VLSI pattern matching chip (PMC). The elemental operation is executed in the PMC using a 4-segment pipeline. The PMC computes the elemental operation in 250ns. Although this architecture results in sufficient speed up for real time speech coding, it is not suitable for image coding applications as the throughput rate for an image is three times greater than that of speech.

Pipeline processors are also used for the implementation of TSVQ as proposed by Braccini *et al.* [57]. In this approach, the codewords (and dummy ones) are arranged in the binary structure. The first two levels of the tree are stored in the local memory of the first pipeline processor. Each additional level is stored in the local memory of subsequent pipeline processors. The n-level tree will therefore occupies n-1 pipeline processors. During the encoding process, each pipeline processor performs the assigned number of operations routing the incoming input vector from the root to the terminal leave of the tree. We note that only the terminal leave of the tree has the index of the codeword. The speedup factor then becomes:

$$S_p = \frac{T_{SISD}}{T_e} \times \frac{\log_2 M}{N} \tag{2.28}$$

where N is the number of codewords, and M is the number of nodes and leaves in the tree.

Systolic Arrays: Another approach to speed up the execution of the elemental operation is to use systolic arrays. Davidson *et al.*[96] have presented several systolic architectures for speech and

image coding which provide a tradeoff between speed and VLSI chip area. These include: sub-linear array, linear array, superlinear array, and two-dimensional array architectures. The number of cycles required for a full search ranges from O(NL) cycles to O(1) cycles.

A higher speed up is possible by exploiting parallelism in the directions of the codebook size N and the vector dimension L. This yields a speed up of:

$$S_p = \frac{T_{SISD}}{T_e} \times N \times L \tag{2.28}$$

Panchanathan *et al.*[92] have proposed a systolic array for adaptive VQ where the encoding process and the codebook generation are overlapped in the same array. The basic PE operates in two modes: forward and reverse. In the forward mode, the PE executes the elemental operation, while in the reverse mode, the PE computes the new centroids. The speed up achieved is not only sufficient for image encoding, but also for sub-optimal codebook generation in real time.

Systolic arrays are also used to implement tree-structured VQ. Yan *et al.*[97] have presented a bit level systolic array for tree structure VQ. The architecture consists of an inner product processor to execute the elemental operation and addressable memories to determine the label of the matched codeword. The bit level systolic array can be expanded to more general tree operations. Abut *et al.*[93] have also proposed a systolic array for speech coding using tree-structured VQ. Here, the elemental operation is executed in 100ns.

Wavefront Arrays: Sun *et al.*[98] have reported a VLSI wavefront array for full search VQ. The wavefront array is configured in a rectangular array of (L+2)xN basic processing cells with regular and local interconnections. Here L is the dimension of the vector, and N is the size of the codebook. The codebook is stored in the upper part of the array. The vectors enter the array in parallel by pipelining the successive components with one unit time delay. The distortion computation rate is 100ns and the latency is N+L+1 time units.

2.6 Summary

In this chapter, the lossless and lossy compression techniques were reviewed along with image evaluation methods. The different image compression techniques including predictive, transform, and hybrid codings were also reviewed. Special emphasis was placed on VQ-based techniques. This followed with a review of various interframe coding techniques The details of the image and video compression standards were then presented. Finally, the VLSI architectures for implementing VQ-based image and video compression algorithms were presented.

CHAPTER 3

Overview of Computational RAM

3.1 Introduction

Computational RAM (C*RAM) [100,101] is a SIMD-memory hybrid architecture where each column of memory has an associated processing element (PE). C*RAM was primarily designed with the goal of augmenting conventional RAM's (used in computer main memory and video frame buffers) with computation capability. The first memory-SIMD hybrid project was experimented by Snelgrove *et al.*, with a machine called VASTOR [102]. Efforts to integrate VASTOR were hampered by the crudeness of tools/technology available in the academic environment. A 64 PEs, 8Kb SRAM-based C*RAM using 1.2 um CMOS technology was prototyped by Elliott *et al.* in 1989, using a simplified PE [101], while a DRAM-based chip with 2048 PEs and 4Mb of memory is intended for the full version C*RAM.

The objective of C*RAM is to enhance the memory subsystem of a computer with low-cost local computing power, while conserving its conventional storage function (Figure 3.1). The main advantages of this computing structure are:

- The utilization of the wide bandwidth inherent in high-density memories, currently wasted or sparely used;
- The significant reduction in the power dissipated in processor-memory data transport by keeping applicable computation inside memory; and
- The incremental change of the well-established conventional computer memory.

There are two major functions in a C*RAM: memory and computation. When functioning as memory, C*RAM is read or written as part of the host processor address space. When functioning as a computing engine, all PE's execute operations (on its own local memory) in parallel, sequenced by a controller.

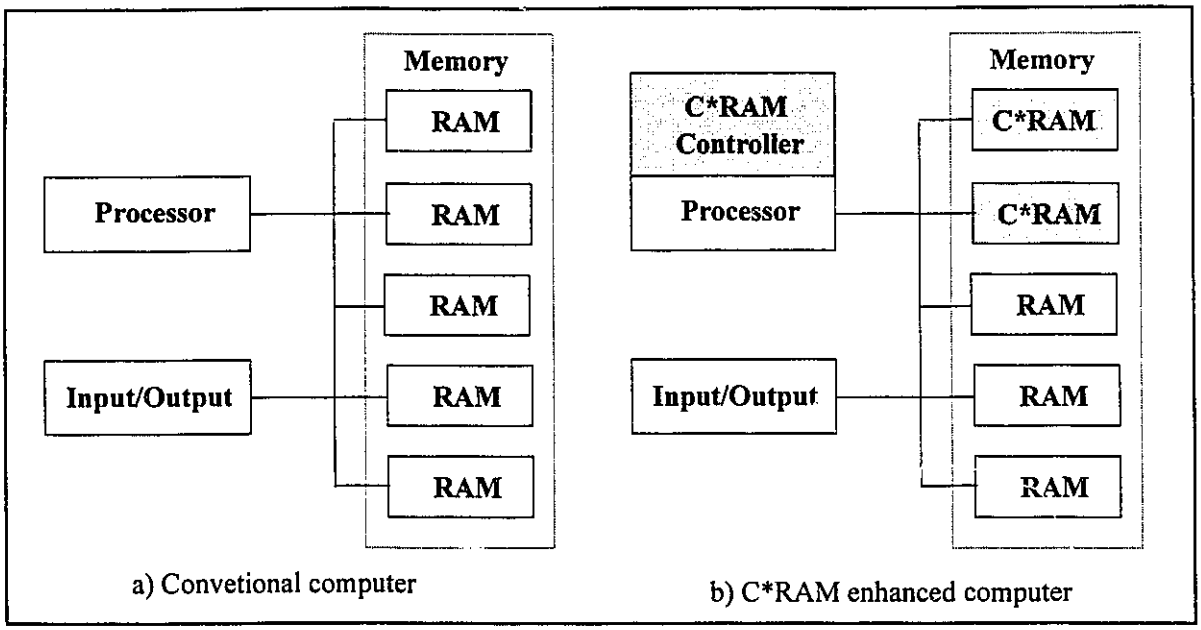

FIGURE 3.1 Conventional Computer Memory and the C*RAM Concepts

The processing elements are organized in a SIMD linear array. This memory localized computing structure has a data parallel nature and maps well to applications in the area of image and video processing, recently gaining more attention in the workstation and personal computer market. A large number of applications can benefit from the fine grain parallelism of C*RAM: scientific (numerical) computation, database search, data compression, neural and genetic algorithms.

3.2 C*RAM Architecture

Recently, C*RAM has been fabricated in a 0.8μm BiCMOS chip having an SRAM core[103]. This chip is referred to as Cs64p1k [104]. For C*RAM, the main point of interest in memory is the number of PE's on chip, a limit attained if the PE layout width (pitch) can be made as fine as the sense amplifier pitch. The maximum number of sense amplifiers in the mentioned SRAM is 64. This means that a memory block can supply at most 64 PE's. In order to support a larger number of PE's, the memory module would have to be redesigned which imposes an unecessary amount of work and increases the overall risk. There are thus 64 processing elements (PE) attached to the 64Kbit memory columns which may effectively be considered as 64 independent computing units having its own PE and 1Kbits of local memory (Figure 3.2).

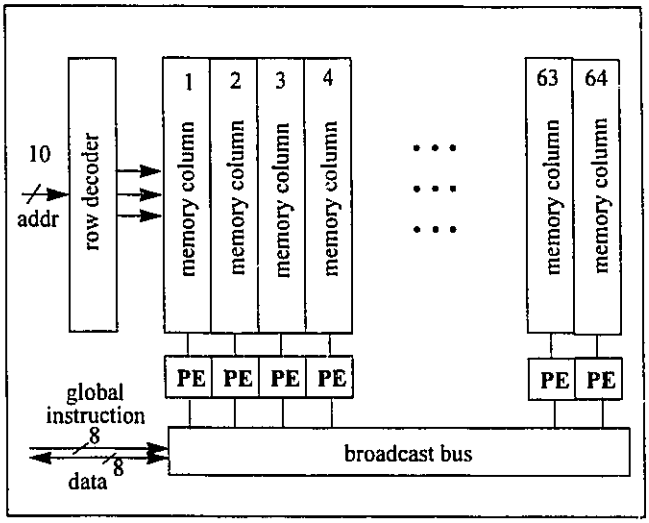

FIGURE 3.2 C*RAM architecture

In conventional memory, each memory cell is addressed by the row and column decoders. However, in this architecture, the entire row is addressed at the same time. Each PE is addressed by the address stored in its local memory. Communication in a C*RAM is done via the PE of each computing unit. Currently, only left communication and right communication are possible.

The setup time, hold time, and clock period are typically 5ns, 0ns, and 20ns, respectively. Different C*RAM cycles are required to operate upon the data. Such cycles are memory-read, memory-

write, read-operate, read-operate-write, operate-write ranging from 20ns to 45ns [104]. The average value of 35ns is chosen for all simulations in chapters 4, 5, and 6.

3.3 Processing Element Model

A PE (Figure 3.3) has two 1-bit registers, X and Y, and a 1-bit ALU which functions as a multiplexer. Inputs to the bit-arithmetic ALU can be contents of registers X, Y and a memory bit M. An 8-bit global instruction from the C*RAM controller chooses between the different functions of the ALU. After each computation, the results are written back into the memory or the registers. Each X, Y register has a left or right connection enabling data shifting and neighborhood communication between PE's.

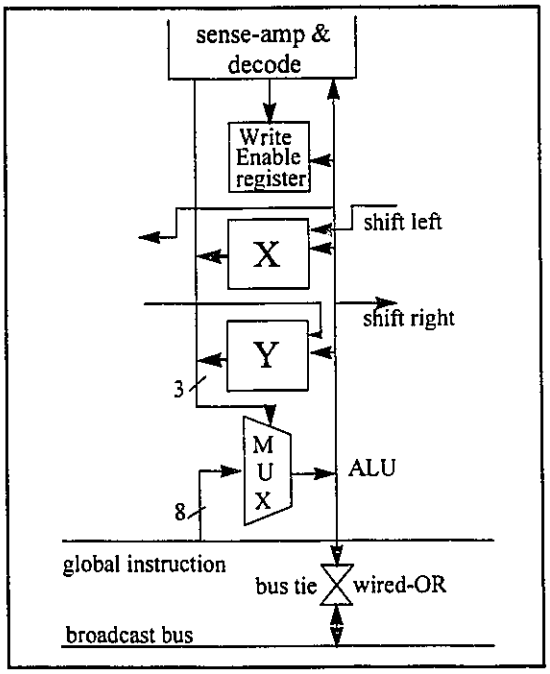

FIGURE 3.3 Processing Element model

In addition to the X,Y registers and the ALU, the Write Enable (WE) register permits conditional operations. WE register functions as a mask, blocking undesired memory columns in various operations. In other words, the WE register is set only when there is a Write to the corresponding memory column, or reset otherwise. We note that different masks may be stored in the scratch

memory for different PE operations. This enables all the PE's in the C*RAM to participate in the global operations without modifying the masked memory contents.

A bidirectional bus-tie circuit performs the wired-OR logic of all PE's local results. This global wired-OR is fed back to all the PE's. The same line may also be used by the C*RAM controller to broadcast a 1-bit constant value to all the PE's.

3.4 Variations in C*RAM designs

In this section, the C*RAM design variations are discussed. The first version of C*RAM [101] was designed with 64 PE's with 8Kbit of memory. Cojocaru *et al.*[104] then proposed a 64PE C*RAM (Cs64p1k) which was designed and implemented using the 0.8um BiCMOS with each PE having 1Kbit of local memory. The special features of this versions are: the design of area-efficient PE and the compactness design criterion for PE circuitry safe-guard against potential technology hazards. The first two versions of C*RAM were designed based on the baseline PE introduced in section 3.3. The third version of C*RAM integrated the programmable segment bus-tie (PSB) to enhance the functionality of baseline PE in the previous C*RAM versions. This chip still has 64 PE's but a smaller local memory of 512bits (Cs64p512). Finally, the bit-parallel oriented PE array was introduced. This chip has 512 PE's and 480 bits of local memory (C512mp512). In addition to programmable segment bus-tie, the new PE has programmable word boundaries and a hard-wired ripple-carry chain, resulting in flexible word length bit-parallel operation. In order to facilitate comparison, the notation X1, X2, X3, and X4 are used for the four C*RAM versions, respectively.

Technology: X1 was laidout using 1.2um CMOS, the subsequent ones however, were designed on the 0.8um BiCMOS technology. The shrink in circuit size surely increases the circuit speed by as much as 30%.

DRAM vs. SRAM: Due to the difference in the basic memory cell, DRAM offers more storage capacity than SRAM, but is four to six time slower. Because DRAM prices are four to six times

lower than the same size SRAM, the main memory in the workstation and personal computers is built with DRAM. In view of DRAM higher density and lower cost, DRAM is assumed to be the choice for building C*RAM. Large capacity SRAM's could be used for more specialized or high end machines. We note that all four C*RAM versions used SRAM.

Single-port vs. dual port: This version of C*RAM uses a single-port SRAM which results in a smaller memory core compared to its dual-port counterpart. However, dual-port SRAM maps well with the dual nature of C*RAM: computation and storage. In case of image and video applications, the two ports can simultaneously be used for frame encoding (on one port) and label transmission (on the other).

Programmable segment bus-tie: This feature enables the bus-tie to be segmented to sections in stead of over the entire 64 computing units thus saving a few cycles. The same function could be performed without this hardware by having masks in the memory.

Bit-serial vs. bit-parallel: Bit-parallel approach uses the word-parallel load in the conventional memory system. This structure loads the word directly without the need of a data transposer as required in the bit-serial structure.

Single-bit PE vs Multiple-bit PE: The latest version of C*RAM reflects the new functionality of the extended PE compared to the baseline PE. From a implementation point of view, the baseline PE conforms to an already fairly classical view of the massively parallel single-bit processor SIMD machine. This type of machine exhibits a weakness in floating point intensive applications, because of the bit-serial nature of multi-bit arithmetic. Floating point multiplies are especially time consuming in bit-serial machines. For example, in the case of integer multiply, multiple redundant memory cycles are needed to retrieve the multiplicand and to store the partial product after every partial addition. There is, therefore, a need to go beyond the single-bit PE.

Programmable word boundaries: With the new feature of programmable word boundaries, flexible word length bit-parallel operation can be performed. This is an interesting feature for a pilot project so that the needs for implementing an application could be easily fine tuned.

Hard-wired ripple-carry chain: In X1, X2, and X3, the full-add operation requires 2N cycles for adding N-bit numbers. In fact, if both numbers are in the memory, one extra cycle is needed to load one of the registers with the bit value (of one operand) in the memory. This results in a total of 2 + 3N cycles (please see Appendix A for details). The hard-wired ripple-carry chain in X4 has reduced the total full-add operation to N + 1 cycles. This is equivalent to approximately 66% reduction in arithmetic computing time.

3.5 Conclusion

In this chapter, an overview of C*RAM was discussed, including C*RAM general architecture, baseline PE (used in the first two versions of C*RAM), and other C*RAM design variations.

CHAPTER 4

C*RAM Implementation of VQ for Image Compression

4.1 Introduction

We recall from section 2.2 that, in VQ, there are two processes: quantization and decoding. In the quantization process, an input vector is compared with a set of codewords (sometimes referred to as templates). The label (index) of the best match codeword, assigned to this input vector, is sent to the receiver. At the receiver, the image is reconstructed by simple table lookup techniques where the label is used as an address to a table containing the codeword.

The abovementioned quantization process can be performed using pattern-matching technique or distortion computation technique. The pattern-matching technique involves the matching process between the input vector with a set of codewords using a preset threshold. If a codeword falls within this threshold, a match is found and the label of the corresponding codeword is sent to the receiver. Otherwise, the threshold is increased and the matching process repeats. In cases of multiple matches, any codeword can be chosen as the best match. It can be seen that if the threshold step size is too large, a large number of codewords match a given input vector resulting in the deg-

radation in image quality. If the threshold step size is too small, it would take more iterations to find the best match codeword. On the other hand, the distortion computation technique always results in a constant execution time. In this technique, the distortion between the input vector and a codeword is, however, calculated. The codeword which results in a minimum distortion is chosen as the best match. The label of the best match codeword is then sent to the receiver.

In this section, a distortion computation technique, called VQ-MAE, is implemented on the SIMD architecture of the C*RAM. Recall from chapter 3 that C*RAM is a SIMD-memory hybrid where a processing element (PE) is attached to each memory columns of the conventional RAM. Typically, a C*RAM with 64 memory columns and 64 PE's would have the power of 64 computing units. These computing units execute in parallel under the supervision of the same controller.

The distortion, in VQ, is usually determined by employing the MSE measure. However, this measure cannot be readily implemented in the C*RAM architecture and hence, is replaced by the mean absolute error (MAE) measure. The MAE measure results in little degradation in performance compared to the conventional MSE measure. Unless otherwise stated, the absolute value of the pixel difference between the pixel of the input vector and the corresponding pixel of a codeword is referred to as pixel difference (PD) throughout the thesis. The VQ-MAE technique is summarized as follow:

> Image is first subdivided into blocks of p x p pixels to form L-dimension vectors. From the training set formed by the input vectors of the standard images, the codebook is generated using the LGB algorithm. The codewords are pre-loaded into the C*RAM's. For each input vector, the PD of each pixel is calculated. This results in L PD's for each codeword. The sum of the PD's is search for the minimum and the label of the corresponding codeword is sent to the receiver.

4.2 Codeword arrangement in C*RAM

For simplicity purpose, a 1-D codeword and a 1-D input vector are considered. In a computing unit, some memory locations (SM) are reserved for PE address, masks, counter, etc. A 1-D codeword (CW) takes 8 localtions (assuming 2^b=256 is the number of gray levels required). Since, the

1-D input vector is, bit by bit, broadcasted and subtracted from the 1-D codeword, it does not use up any location. The 2'complement operation is then applied on the negative results of the subtraction and the PD is then stored in the next 8 memory locations. Finally, the sum of the distortion (CD) is stored in the 8 subsequent memory locations. The arrangement of all components are shown in Figure 4.1

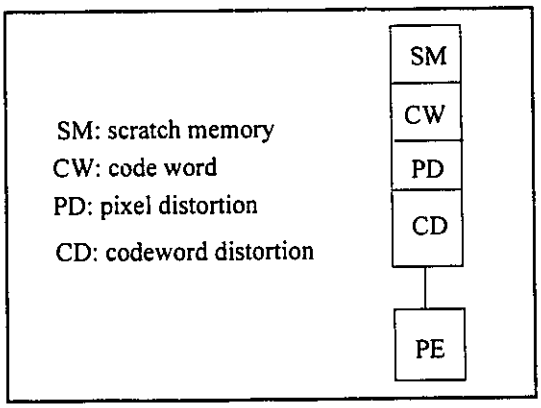

FIGURE 4.1 Arrangement in a computing unit

For 16-D codeword and 16-D input vector, the CW section (Figure 4.2) is extended to 16 x 8 memory locations of the same computing unit and so as the PD section. The CD is now 12-bit long. For 256 16-D codewords and 16-D input vectors, 256 computing units are needed. The overall arrangement is shown in Figure 4.2.

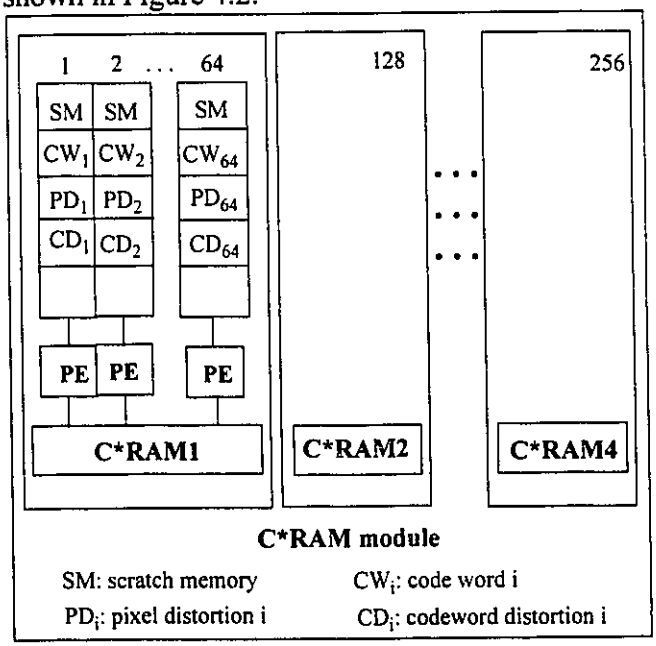

FIGURE 4.2 Codewords arrangement in C*RAM using VQ-MAE

4.3 Implementation

The quantization process of VQ-MAE can be executed in two stages: PD computation and MAE search as shown in the flowchart of Figure 4.3.

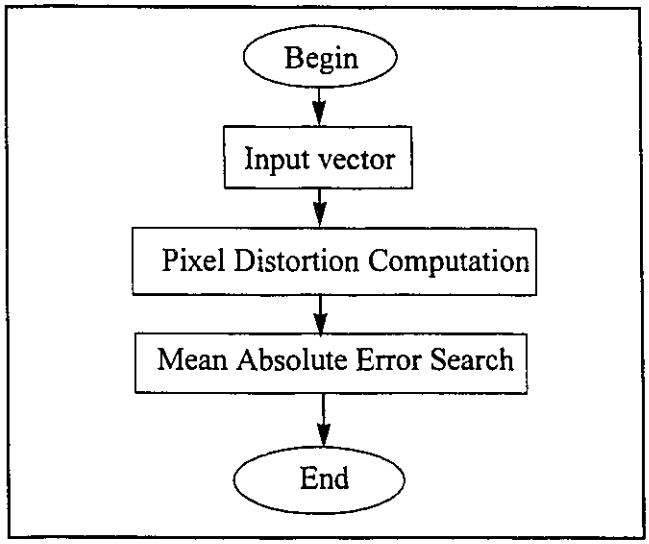

FIGURE 4.3 Flow chart for VQ-MAE

In the PD computation stage, the PD's of every codeword and the input vector are calculated in parallel. With a codebook of N codewords each of size L, the pixel distortion $PD_j(x, y_i)$, for a given input vector $x = (x_1, x_2, \ldots, x_L)$, and a codeword $y_i = (y_{i1}, y_{i2}, \ldots, y_{iL})$, is given by:

$$PD_j(x, y_j) = |x_j - y_{ij}| \quad ,i = 1 to N \quad j = 1 to L \tag{4.1}$$

The PD computing stage is implemented on the C*RAM as follow: In order to implement the absolute operation, 1's are appended next to the MSB of each codeword pixel, and 0's are appended next to the MSB each input vector pixel (Figure 4.4). The pixels of the input vector are then subtracted from those of the padded codeword and the appended bit is changed accordingly. If the appended bit of the result $PD_j(x, y_i$ is a 0, then result $PD_j(x, y_i$ is negative, and is therefore 2's complemented. The corrected PD's (without the appended bit) are then added. The sum is called codeword distortion (CD) and is given by:

$$CD_i = \sum_{j=1}^{L} PD_j(x, y_i) \qquad i = 1 to N \tag{4.2}$$

where CD_i is the 12-bit sum of all 16 pixel distortions of a particular codeword i. An alternative implementation is to use 1's complement (method B) instead of 2's complement (method A) in the PD computation.

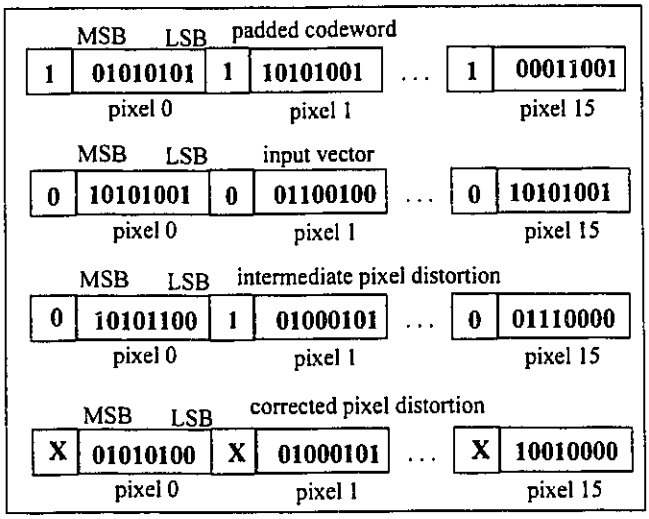

FIGURE 4.4 Pixel Distortion Computation

4.4 Simulation Results

Implementation of VQ-MAE has been executed at the functional level of the C*RAM simulator. Simulations have been carried out using Lena image of size 512 x 512 pixels with the 4 universal codebooks of size 512, 256, 128, and 64 generated from 10 standard images using LBG algorithm. The results of simulations which used 1's complement in the PD computation stage are shown in brackets. The image quality in PSNR and the execution time (in cycles) are tabulated as follows:

TABLE 4.1 PSNR comparison (dB) of VQ-MAE

Codebook size	FSVQ	VQ-MAE
512	29.70	29.46 (29.44)
256	29.05	28.83 (28.81)
128	28.23	27.98 (27.96)
64	27.45	27.21 (27.20)

TABLE 4.2 Execution Time comparison(η_1 cycles) of VQ-MAE

Codebook size	PD Computation	Quantization (Q)	Indexing (I)	Total (cycles)
512	656 (432) x K	629 x K	9 x K	1294 (1070) x K
256	656 (432) x K	629 x K	8 x K	1293 (1069) x K
128	656 (432) x K	629 x K	7 x K	1292 (1068) x K
64	656 (432) x K	629 x K	6 x K	1291 (1067) x K

where K is number of input vectors in an image.

Table 4.1 shows that the image qualities of VQ-MAE are very close to those obtained by the full search VQ. The loss of less than 0.2dB can be attributed to the fact that MAE measure is used for quantization instead of MSE. The reconstructed images of FSVQ using codebooks of sizes 512, 256, 128, and 64 are shown in Figures 4.7, 4.9, 4.11, 4.13, respectively. The reconstructed images of method B using codebooks of sizes 512, 256, 128, and 64 are shown in Figures 4.8, 4.10, 4.12, and 4.14, respectively.

In Table 4.2, the complete encoding cycles in terms of PD computation, quantization, and indexing for each simulations are shown. PD computation column shows the number of cycles required to broadcast the input vector to C*RAM and to calculate the PD's between the input vector and all codewords. Quantization column indicates the number of cycles required to find the best match codeword. And indexing column shows the number of cycles required to send the index of the best match codeword off-chip. We note that the indexing time is small and proportional to the number of bits used for labels of the codebook, hence it can be neglected. The numerical values in the total column shows the average number of cycles required to encode each input vector.

In order to calculate the execution time of an image, the numerical value in the total column is multiplied by the cycle time to obtain the average executing time for each input vector. This average is then multiplied by the number of input vectors in the image to obtain the image coding time. For example, for an image of size 512 x 512 with a codebook of size 512, it takes

(512*512)/(16) * (1070) * 35 ns = 614ms to encode if method B is used. For smaller size images, such as 288 x 360, it only takes (288*360)/(16) * (1070) * 35 ns = 243ms to encode.

Also shown in Tables 4.1 and 4.2, method B outperforms method A by more than 200 C*RAM cycles without significantly degrading the image quality.

4.5 Summary

In this chapter, C*RAM implementation of VQ-based algorithms using MAE measure has been presented. VQ-MAE shows a good coding performance at less than 1100 cycles per input vector which corresponds to a speedup of 20%. VQ-MAE using 1's complement demonstrates a strong coding performance over it's 2's complement implementation with little image degradation. It is, therefore, used in subsequent implementations. Moreover, real-time image compression is feasible if a number of C*RAM modules are used.

FIGURE 4.5 Original Lena image of size 512 x 512 pixels.

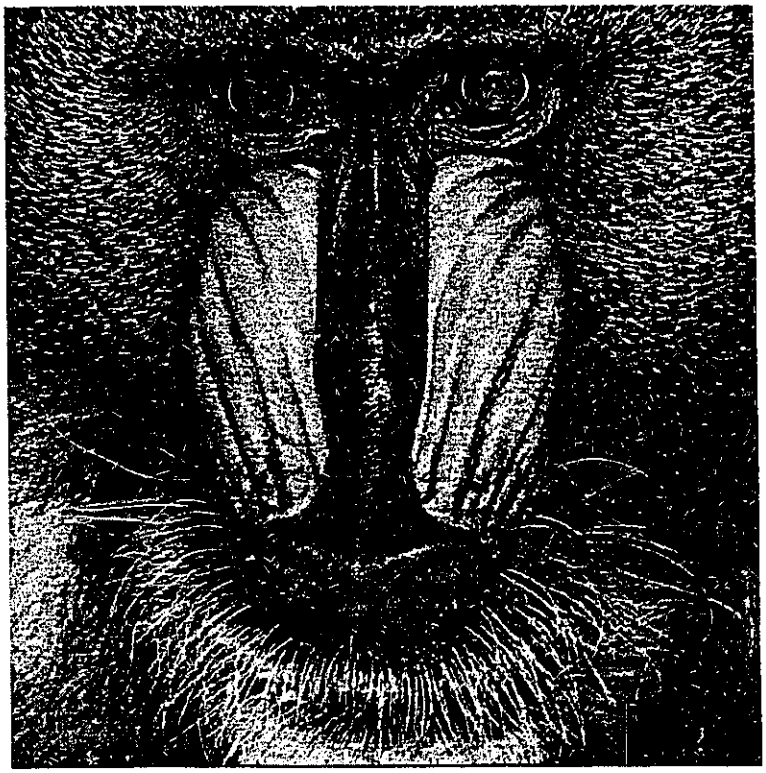

FIGURE 4.6 Original Baboon image of size 512 x 512 pixels

FIGURE 4.7 FSVQ using universal CB (N=512; PSNR=29.70dB; BR=0.563bpp)

FIGURE 4.8 VQ-MAE using universal CB, 1's comp. (N=512; PSNR=29.44dB; BR=0.563bpp)

FIGURE 4.9 FSVQ using universal CB (N=256; PSNR=29.05dB; BR=0.500bpp)

FIGURE 4.10 VQ-MAE using universal CB, 1's comp. (N=256; PSNR=28.81dB; BR=0.500bpp)

FIGURE 4.11 FSVQ using universal CB (N=128; PSNR=28.23dB; BR=0.438bpp)

FIGURE 4.12 VQ-MAE using universal CB, 1's comp. (N=128; PSNR=27.96dB; BR=0.438bpp)

FIGURE 4.13 FSVQ using universal CB (N=64; PSNR=27.45dB; BR=0.375bpp)

FIGURE 4.14 VQ-MAE using universal CB, 1's comp. (N=64; PSNR=27.20dB; BR=0.375bpp)

CHAPTER 5

C*RAM Implementation of Scalable VQ for Image Compression

5.1 Introduction

In general, each image and video compression application has specific functional requirements and quality contraints. For example, an image browsing application generally requires the capability of displaying images of different sizes and qualities. Similarly, simultaneous transmission of NTSC and HDTV also requires the capability of displaying video signals in multiple spatial resolutions. Although the two applications require similar features, their quality requirements can be different. The transmission application requires high subjective quality at all spatial resolutions, while the browsing application may require a high quality only at the full spatial resolution. In addition, some of the features of an application may be optional and may depend on the availability of resources such as, CPU processing power, transmission bandwidth and display resolution. These resources normally vary from one system to another, and in some cases may impose constraints even on the basic features required by the applications. Therefore, the encoding and

decoding algorithms should be flexible enough to provide a satisfactory performance, by efficiently using the available resources. This extra flexibility is referred to as scalability. In other words, scalability is a property of the image/video data representation that allows applications to provide certain features [105]. The types of scalabilities are now briefly reviewed.

Temporal scalability refers to a hierarchial representation of video such that each level of the hierarchy has a different frame rate. Temporal scalability is only applicable to video and thus is not discussed in detail in this thesis.

SNR (quality) scalability is a generic feature referring to the representation of images in different qualities, or Signal-to-Noise ratio (SNR), at a fixed size. SNR scalability makes possible an approximate reconstruction of the image using only a small portion of the compressed bit stream for quick image recognition. In addition, SNR scalability is also useful in transmitting images over low bandwidth channels, such as ISDN. SNR scalability, also referred to as *progressive image transmission* (PIT), can be achieved in two different ways [86]:

- Spectral selection: The low frequency (of the transformed) coefficients are processed first, then come the additional higher frequency coefficients.
- Successive approximation: The MSBs are processed first, then the less significant ones.

Spatial scalability, however, refers to the feature of image representation in different sizes or spatial resolutions. This features provides the capability of reconstructing images in lower spatial resolution using a small portion of the compressed bit stream. Like SNR scalability, spatial scalability is also very useful in image browsing applications. Altenatively, spatial scalability can be used to simultaneously display an image at different spatial resolutions as in videophone applications. In this chapter, a review of SVQ is first presented. Then, the Modified Scalable VQ (MSVQ) technique is proposed. This follows with a Binary tree to Parallel structure mapping algorithm. Finally, the C*RAM implementation of the Modified SVQ is detailed.

5.2 Review of Scalable Vector Quantization [112]

VQ has been known to provide a good coding performance at low bit rates. Residual VQ [107,108] and MSVQ [47] achieve SNR scalability by iterative vector quantization of the residual error images. M/RVQ requires multiple codebooks, one for each residual error image. Since there is little correlation between the residual error images, there is no correlation between their labels. In other words, the label bitstream is not scalable. Unlike M/RVQ and MSVQ, tree-structured VQ (TSVQ) provides scalable labels for achieving SNR scalability. TSVQ also uses multiple codebooks of different sizes which are organized in a binary tree structure. TSVQ may result in a sub-optimal compression performance for the full quality image. This is due to the generation of codebooks in a tree-structured fashion, where each child codeword (at level i+1) is trained using only those image vectors which were represented by its parent codeword (at level i). We note that neither M/RVQ nor TSVQ can be used for providing spatial scalability.

A VQ-based technique for achieving spatial scalability should not only ensure partial decodability of VQ labels for reconstructing the images at smaller sizes, but should also provide a good quality image at the full spatial resolution (512 x 512 pixels). The technique should be able to provide good to excellent image quality for low resolution images of sizes 256 x 256, 128 x 128. Scalable Vector Quantization (SVQ) proposed by Panchanathan *et al.*[106] requires separate codebooks, one for each spatial resolution. For example, in Figure 5.1, the image is first reconstructed at a resolution of 128 x 128 pixels (I_3) by decoding only the first 6 MSBs of the VQ labels which serve as an index for a table lookup operation in a small size codebook, U_3. The spatial resolution of image I_3 can be further enhanced by using the next MSB of the labels for a table lookup operation in codebook U_2 to reconstruct the image at a size of 256 x 256 pixels (I_2). The full size image (I_1) of resolution 512 x 512 pixels is obtained by using all the 8 bits of each label as an address to codebook U_1. Note that codewords in codebooks U_3, U_2, and U_1 are codewords of dimensions 1, 4, and 16, respectively.

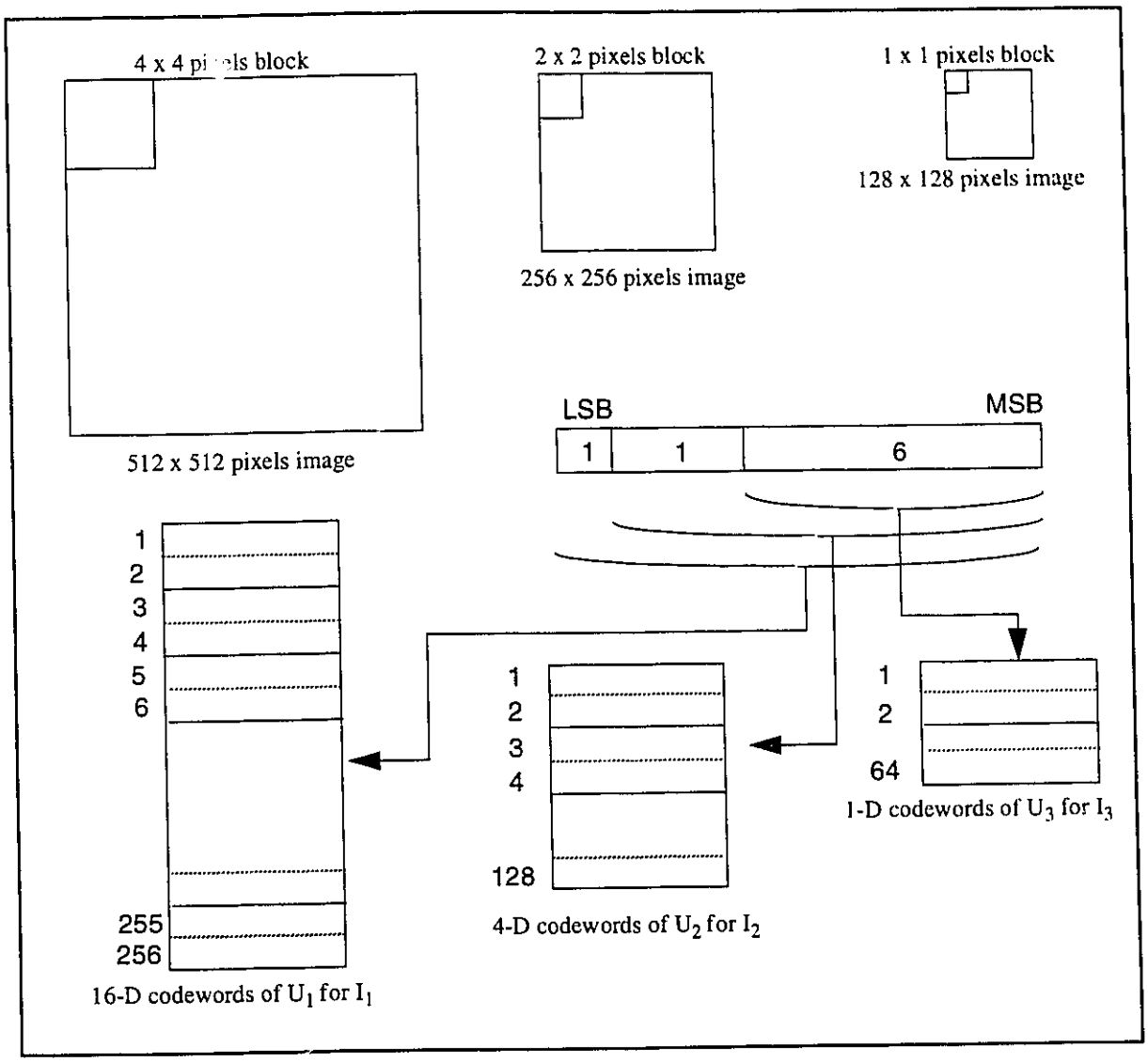

FIGURE 5.1 Scalable Vector Quantization technique

In SVQ, there are 2 types of codebooks: full resolution codebook (FCB - U_1) and reduced resolution codebook (RCB- U_i, i =2, 3, etc). To start with, a universal codebook U_1 of size N_1 (N_1 is usually a power of 2) is generated from a large number of test images $\{T_1\}$, having a resolution of M_1 x M_1 pixels, using the LBG algorithm with binary splitting technique. We note that $N_1 = 256$ and $M_1 = 512$ in Figure 5.1. The universal codebook generation ensures a reconstructed image of good quality at the highest resolution. In order to generate U_1, the test images are partitioned into non-overlapping blocks of size B_1 x B_1 pixels which are then reorganized as vectors of dimension

B_1^2. The LBG algorithm with binary splitting technique is then applied to generate the codebook of size 256.

We note that the binary splitting technique for codebook generation results in reasonably similar neighbor codewords (i.e codewords with adjacent labels). In other words, these vectors are likely to be represented by the same parent codeword in the previous stage codebook. Once the codebook U_1 is generated , each image vector in I_1 is vector-quantized by searching the codebook to locate the best-match codeword and the corresponding label is transmitted. Although the entire label bit stream is now available at the decoder, it is not possible to reconstruct the image of smaller sizes until the corresponding RCBs (corresponding to the codebooks of the reduced spatial resolutions) are generated. We note that good quality RCBs can be obtained for the reduced resolution images, if the codebook U_1 is generated using the binary splitting technique. The RCBs are generated by merging the codewords for full size image through a simple process as described below:

The codebook U_2 is generated by first downsampling the test images $\{T_1\}$ by a factor of 2 in each direction to obtain the lower resolution images $\{T_2\}$. A similar procedure can be used to generate U_3, or U_4, etc. In [106] wavelet filters was chosen to downsample the images. The downsampled images are partioned into blocks of size B_2 x B_2, each of which is reorganized as a B_2^2 vector such that:

- $B_2 < B_1$;
- B_2 and B_1 are power of 2; and
- $\{T_2\}$ and $\{T_1\}$ contains the same number of blocks.

Note that the reduced size image I_2 has to be reconstructed using the MSB's of each label in the bitstream, therefore it is required that $\{T_2\}$ images consists of the same number of blocks as $\{T_1\}$.

Let the size of codebook U_2 be denoted by N_2 such that $N_2 < N_1$ and N_2 is a power of 2. The codebook U_2 with the codewords of size B_2^2 is generated by merging the codewords in U_1 through the merging process as described below:

- First, all the B_1^2- dimensional blocks represented by codewords A and B in the original images $\{T_1\}$ are marked.
- Second, all the corresponding B_2^2- dimensional blocks in the downsampled images $\{T_2\}$ are marked.
- The codeword P is computed by calculating the mean of all vectors marked in the downsampled test images.

The merging process is illustrated in Figure 5.2 below:

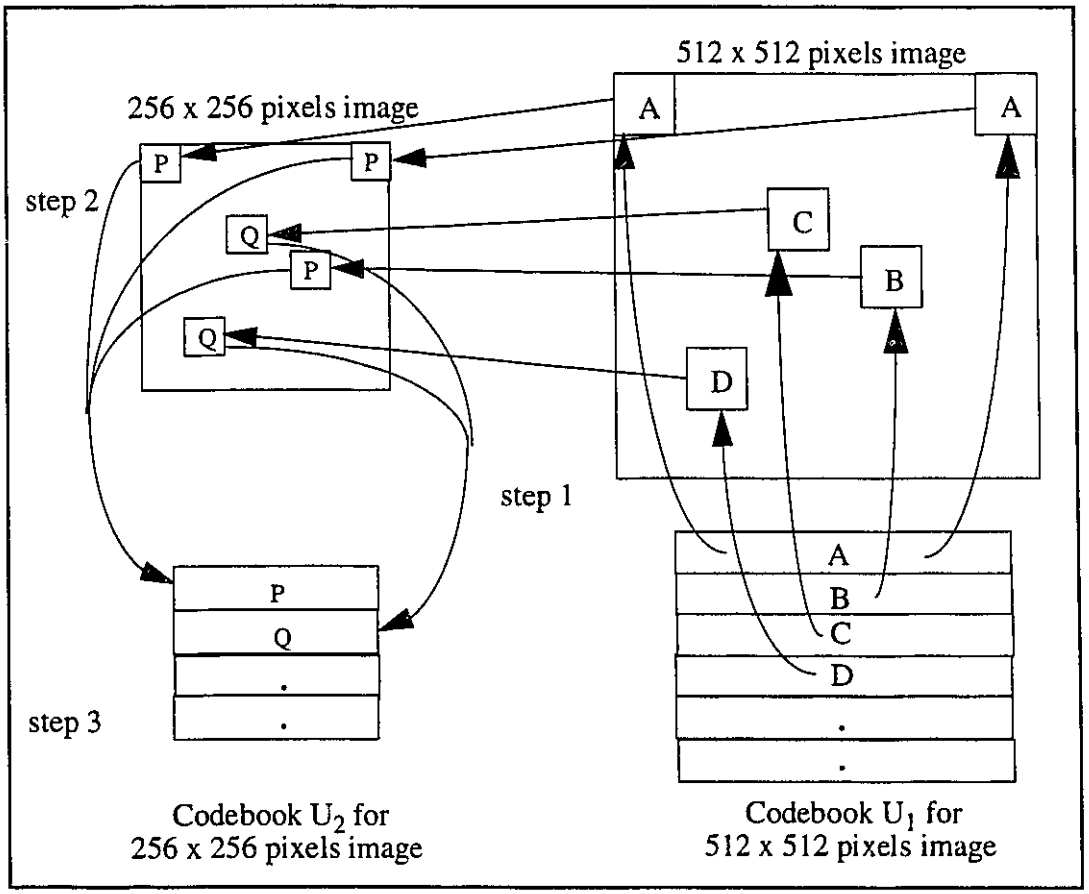

FIGURE 5.2 Codebook Generation of Reduced Size Images Using SVQ

The advantage of this technique is that the iterative LBG algorithm is not required to generate the codewords of the reduced size codebook. However, the image downsampling and vector marking processes require a great deal of computational effort and memory for storage.

5.3 The Modified SVQ

In this section, we propose a modification to SVQ technique in [106] which greatly simplifies codebook generation procedure for reduced size images and hence significantly improving the coding efficiency.

The codebook generation procedure for the lower resolution images can be summarized as follows. Contrary to the splitting algorithm, the codebooks U_2 and U_3 are generated by merging the codewords in U_1. The codewords in U_2 is generated by mean-averaging every 2 consecutive codewords of U_1 and more importantly, reducing the codeword dimension by a factor of 4. Similarly, codewords in U_3 can be generated by mean-averaging every 4 consecutive codewords in U_1 and also reducing the codeword dimension by a factor of 16. Compared to SVQ, this RCB generation technique possesses the following advantages: reduced effort for downsample the images $\{T_1\}$ and elimination of vector marking process.

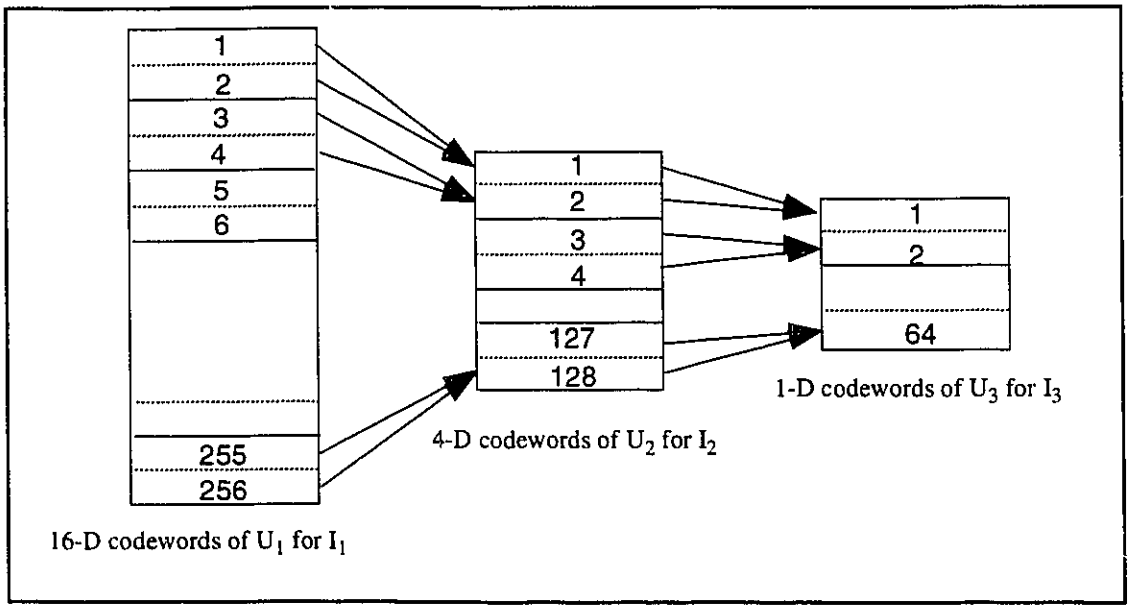

FIGURE 5.3 Codebook Generation of Reduced Size Images Using Modified SVQ

Since the codewords in U_i are formed by mean-averaging 2 consecutive codewords in U_{i-1}, this relationship can be considered as binary merging process as shown in Figure 5.3. Hence, the label of any codeword in RCBs can be directly derived from the label of codewords in FCB. In the following section, a binary-tree to parallel structure mapping algorithm is introduced. This algorithm enables the Modified SVQ to be easily implemented on the C*RAM as described in section 5.5.

5.4 Binary-Tree to Parallel Structure Mapping Algorithm

The binary-tree to parallel mapping algorithm is described as follows: Let A_0, B_{0-1}, C_{0-3}, and D_{0-7} be the codewords of the RCB's of size 1, 2, 4, and 8, respectively. Let E_{0-15} be the codewords of the FCB. We note that the dimensions of the codewords are of little importance since they do not affect the column address by which the codewords are identified. The binary merging process representing the relationship of these codewords is illustrated in Figure 5.3.

It can be seen that the codewords E_{0-15} are stored in the first level, horizontally across 16 columns of a parallel structure. On the other hand, 8 codewords D_{0-7}, 4 codewords C_{0-3}, 2 codewords B_{0-1}, and 1 codeword A_0 are stored in the second level as shown in Figure 5.3.

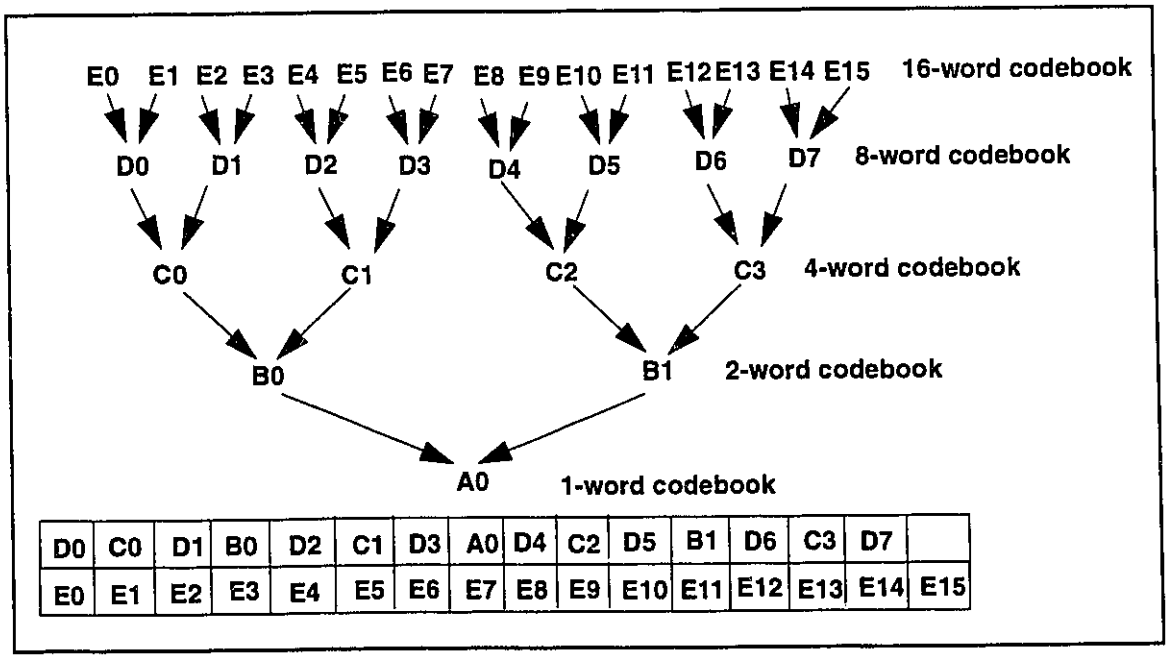

FIGURE 5.4 Binary Tree to Parallel Mapping for Codeword Arrangement

This arrangement can be derived using the following polynomial:

$$\sum_{i=0}^{N-1} 2^i = 2^N - 1 \tag{5.1}$$

By rearranging it, we obtain:

$$2^N = \sum_{i=0}^{N-1} 2^i + 1 \tag{5.2}$$

and expanding the right side:

$$2^N = 2^{N-1} + 2^{N-2} + \ldots + 2^2 + 2^1 + 2^0 + 1 \tag{5.3}$$

With the assumption that all codewords are of equal dimension, it can be seen from equation (5.3) that the number of locations required to store a codebook of size 2^N is equal to the total number of locations required to store codebooks of size 2^{N-1}, 2^{N-2},...,4, 2, 1, with an empty location.

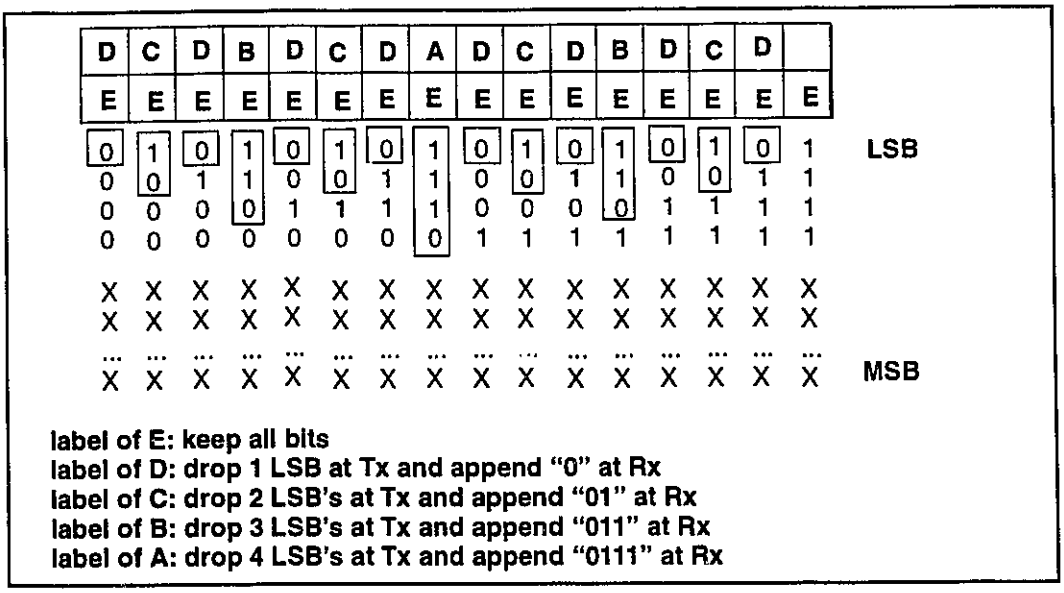

FIGURE 5.5 Index Assignment for Different Reduced-Size Codebooks

The locations of the columns can be addressed as shown in Figure 5.5. It can be seen that there is a unique bit-pattern to each type of codewords. For instance, the LSB of codeword D's are always

"0", the 2 LSBs of codeword C's are always "01", the 3 LSBs of codeword B's are always "011", and the 4 LSB of codeword A's are always "0111". Not only does this arrangement help identify the type of codewords, but it also helps reduce the bit rate. For low resolution image transmission, the bit-patterns can be dropped at the transmitter and be re-appended at the receiver. For full resolution image transmission, all label bits are used.

5.5 C*RAM Implementation of Modified SVQ

In this section, we introduce the details of the implementation of Modified SVQ. A universal codebook is generated from 10 standard images using the LBG algorithm with binary splitting technique. The Modified SVQ technique starts with the decomposition of the input image into a set of input vectors. The two basic steps are quantization and decoding (Figure 5.6).

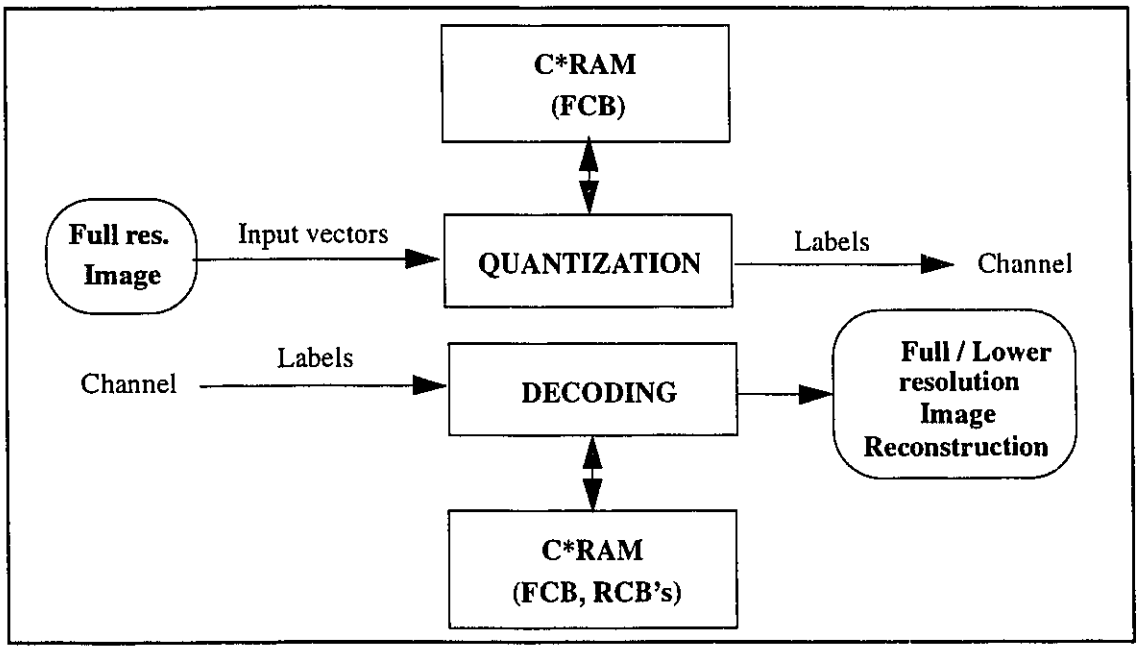

FIGURE 5.6 Scalable Image Quantization and Decoding Scheme

In the quantization process, for each input vector, the FCB is searched to obtain the closest codeword. Compression is achieved by transmitting the label of the codeword. Reconstruction of images can be implemented by simple table lookup techniques where the label is used as an address to a table containing the codewords. The full size image can be reconstructed by using all

the label bits with a table lookup operation in FCB. Similarly, the low resolution images are reconstructed by using some of the label bits (from the same VQ operation) with a table lookup operation in RCB.

C*RAM implementation of Modified SVQ has been performed at the functional level. Image is subdivided into blocks of 4 x 4 pixels to form 16-D vectors. The FCB is pre-loaded into the C*RAM where all 16 pixels of each codeword are arranged in one computing unit. Similarly, the RCBs are pre-loaded in the second level of the respective computing units as described in section 5.4. A FCB of 256 codewords will, therefore, occupy 256 units which corresponds to a module of 4 C*RAM chips. The RCBs which are of size 128 and 64 will occupy 192 computing units.

The input vectors are broadcast to all the 256 computing units. The encoding process is then executed in parallel over all the codewords of FCB.

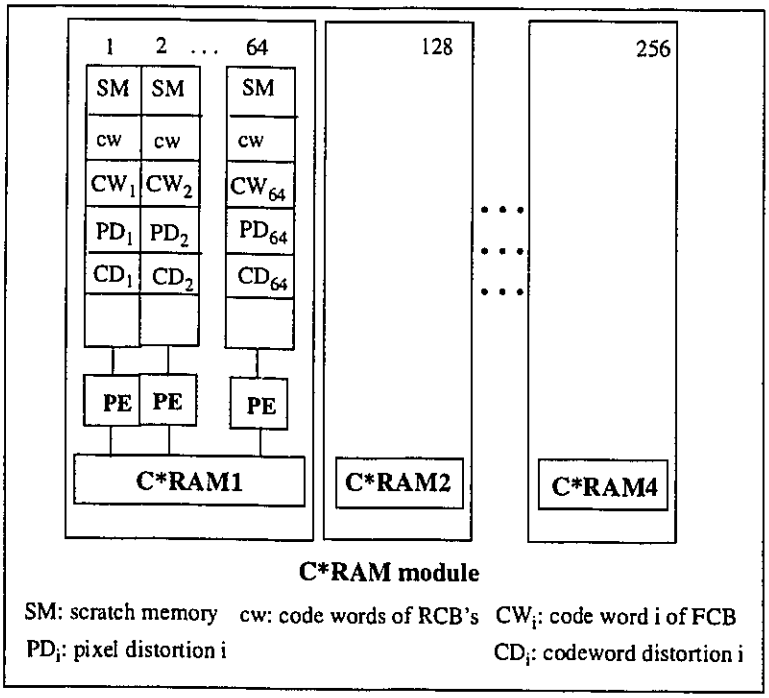

FIGURE 5.7 Codeword Arrangement in C*RAM for Modified SVQ

Similar to the VQ-MAE implementation, the first two stages of the Modified SVQ process are: PD computation and CD computation. We recall from section 4.2 that the CD computation stage,

the label of the codeword which has smallest codeword distortion will be chosen as the best match. This label can be fully or partially sent to the receiver depending on the resolution requirement. If the full size image is required, all the label bits are sent. If the low resolution image is required, some of the MSB's of the labels are sent. At the receiver, appropriate bit-pattern is concatenated (please refer to section 5.4) to this label to select the appropriate codewords in the appropriate RCB.

5.6 Simulation Results

Simulations have been carried out using standard images Lena and Baboon, Figures 4.5 and 4.6, respectively, (which are in the training set), each of size 512 x 512 pixels with a universal codebook of size 256 codewords.

In Table 5.1, the loss in PSNR for a 512 x 512 pixels image between Modified SVQ and SVQ is partially due to the fact that the MAE selection criteria chosen for Modified SVQ is used instead of MSE in SVQ.

TABLE 5.1 PSNR comparison (dB) with FCB of 256

Image size	Mod.SVQ Lena	SVQ Lena	Mod.SVQ Baboon	SVQ Baboon
128 x 128	23.56	24.35	19.79	19.41
256 x 256	27.10	27.87	21.27	21.51
512 x 512	28.83	29.05	22.87	23.10

We note that the subjective quality evaluation is imperceptible. The reconstructed Lena images of sizes 128 x 128, 256 x 256, and 512 x 512 pixels using Modified SVQ are shown in Figure 5.8, 5.9, and 5.10, respectively. The reconstructed Baboon images of sizes 128 x 128, 256 x 256, and 512 x 512 pixels using Modified SVQ are shown in Figure 5.11, 5.12, and 5.13, respectively.

Unlike JPEG hierarchical encoding where considerably large storage capacity is required for low resolution image retrieval, the Modified SVQ only requires another set of RCB codebooks, whose

total size is always one or more codewords less than that of the FCB. Since these codewords can be directly derived from the FCB, computational complexity can be considered negligible. The low resolution images as well as the full resolution image can be reconstructed from the same bit stream with no additional execution time. In addition, the Modified SVQ does not require image downsampling and vector marking processes which significantly simplifies the RCB generation compared to SVQ. This simplification makes possible real-time scalable video coding using VQ.

5.7 Summary

In this chapter, we have detailed the Modified SVQ technique which has been implemented on the C*RAM structure. The proposed technique significantly reduces the complexity of RCB generation with minimal degradation in image quality and makes possible real time implementation of scalable using VQ.

FIGURE 5.8 Lena image of size 128 x 128 reconstructed using Modified SVQ

FIGURE 5.9 Lena image of size 256 x 256 reconstructed using Modified SVQ

FIGURE 5.10 Lena image of size 512 x 512 reconstructed using Modified SVQ (N=256; BR=0.500bpp)

FIGURE 5.11 Baboon image of size 128 x 128 reconstructed using Modified SVQ

FIGURE 5.12 Baboon image of size 256 x 256 reconstructed using Modified SVQ

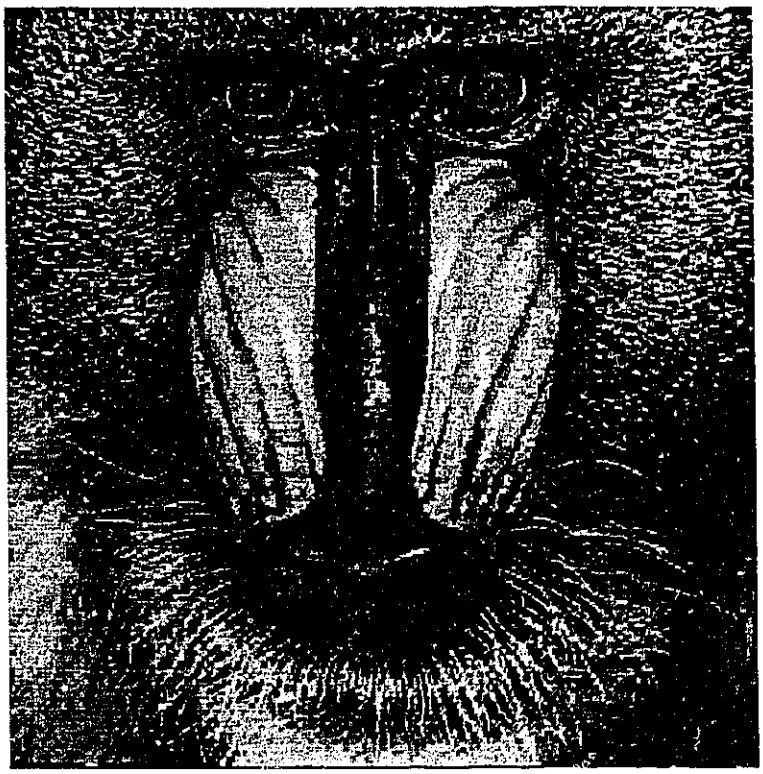

FIGURE 5.13 Baboon image of size 512 x 512 reconstructed using Modified SVQ (N=256; BR=0.500bpp)

CHAPTER 6

C*RAM Implementation of Adaptive VQ for Video Compression

6.1 Introduction

Video sequences are highly non-stationary and generally exhibit variations from frame to frame; hence using a fixed codebook to encode the difference frame cannot guarantee a good coding performance. Recall from section 2.2 that two approaches are usually used to improve the coding performance of VQ. The first is to use a large fixed codebook, while the other is to use a smaller codebook but replenish the codebook so that it matches the local statistics of the frame to be encoded. However, both approaches result in further increases in computational complexity and bit rate. In this chapter, an adaptive codebook replenishment VQ using index-based ME (AVQ+ME)[80] algorithm is proposed. This algorithm exploits the spatial as well as temporal redundancies in order to reduce the bit rate (Figure 6.1). In addition, the proposed AVQ+ME technique has been implemented on the parallel structure of C*RAM.

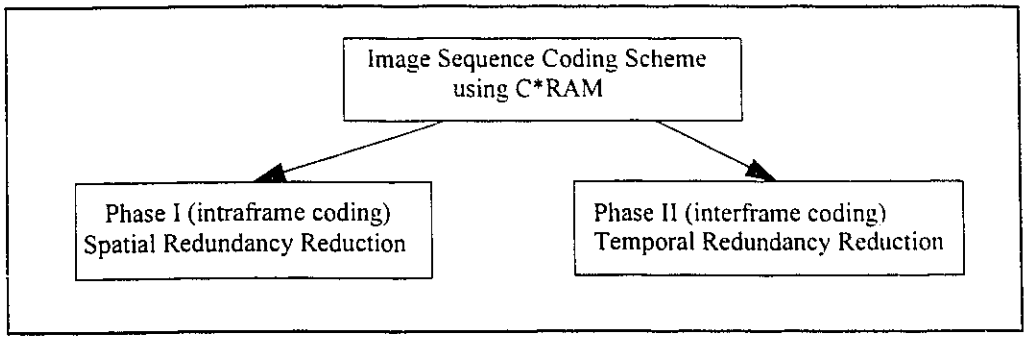

FIGURE 6.1 Image Sequence Coding Scheme

In the following section, the AVQ+ME algorithm is introduced. The C*RAM implementation of AVQ+ME algorithm is presented in section 6.3. The analysis of coding performance and execution time is given in section 6.4. Finally, the summary is presented in section 6.5.

6.2 The AVQ+ME Algorithm

In this section, the index-based motion estimation (ME) concept is first introduced followed by the adaptive VQ technique. We recall from section 2.3.6, that the label replenishment (LR) technique does not fully exploit the temporal redundancy in the labels of successive frames in the sequence. We propose a novel technique which achieves further compression by using ME on the labels. In this technique, a search area (SA) is designated to each label. Typically, the size of SA is 3x3 or 4x4 labels. The displacements in the 3x3 and 4x4 SA's can be represented using 3- and 4-bit motion vectors, respectively. A 4x4 SA has been chosen in this implementation. We note that, the label of the current frame M^j is compared not only to the corresponding labels in the previous frame M^{j-1}, but also to the 4x4 SA of M^{j-1} (Figure 6.2).

We recall from chapter 4 that the distortion between a pixel of the input vector and the corresponding pixel of the best match codeword is called pixel difference (PD). First, if the PD's of the best match codeword are less than or equal to a threshold Δ, then there are three scenarios depending on the relative position of the label:

- **block exact-match**: the label of the input vector matches exactly with the label at the same location in the label map M^{j-1}, for instance in Figure 6.2, $5M^j$ matches $5M^{j-1}$. (The exact match

concept can be loosely defined as match within a small tolerance if the codewords are ordered and closely spaced).

- **block neighbor match**: the label of the input vector matches exactly with one of the neighboring labels in the SA in M^{j-1}. The resulting label will have a non-zero motion vector. For example, $5M^j$ matches any of $\{M^{j-1}\}$ in SA except $5M^{j-1}$.
- **block mismatch**: the label of the input vector does not match any labels in the SA in M^{j-1}. In other words, a new label is required. For example, $5M^j$ does not match any of the label within the SA in M^{j-1}.

In cases where any PD of the best match codeword is greater than a threshold Δ, the block mean is updated:

- **block mean updating**: Here, the label of the best match codeword is stored in the label memory, and the input vector will be used to update the temporary mean of the codeword.

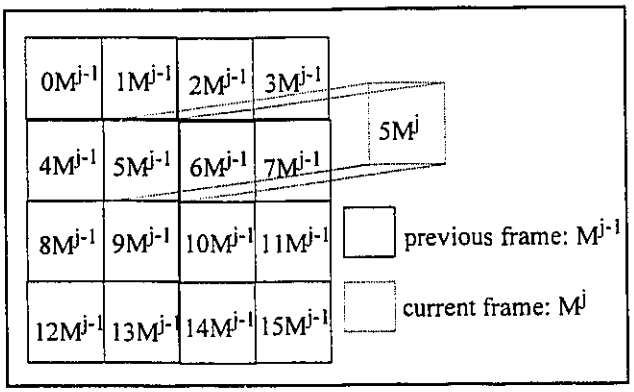

FIGURE 6.2 Motion estimation in a 4x4 label search area.

After the individual blocks have been compared, the 4 x 4 macro blocks of the label memory are then searched for identical motion vectors:

- **macro-block match**: If all blocks in the same macro block have the same motion vector (either all exact-match blocks or all neighbor match blocks), a macro block match is found, and the motion vector of any block can be used as the motion vector of the macro block.

Before introducing the AVQ+ME algorithm, it is worth illustrating the 2-D model of the subregions of the Voronoi disjoint partitions and the related concepts. Figure 6.3 shows a Voronoi partitions for generating a codebook of size 3. Let Δ be the small and constant pixel threshold. The pixel threshold Δ defines a hypersphere of radius Δ around each cluster center w_i^{j-1}. This hypersphere is referred to as $R_i^{'j}$. The region outside of $R_i^{'j}$, is referred to as O_i^j. The indices j and j-1 denotes the current frame and previous frame, respectively.

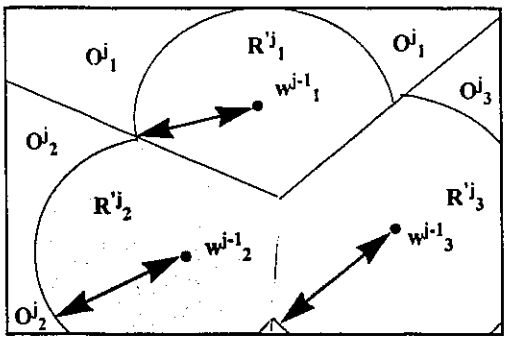

FIGURE 6.3 2-D model of subregions of the Voronoi disjoint partitions at the jth frame

The codebook updating procedure is described as follows: If the input vectors belong to the subregion $R_i^{'j}$, they can be fully represented by the existing codeword w_i^{j-1}. In cases where the input vectors belong to the subregion O_i^j, they can be reconstructed by using the temporary mean value of the input vectors z_i^j, where $z_i^j = E\{v^j, v^j \in O_i^j\}$, prior to codebook replenishment. The codewords are later updated by the weighted sum of the mean value z_i^j and the existing codeword w_i^{j-1}.

We now present AVQ+ME algorithm: The first frame is used to generate the initial codebook which is then transmitted to the receiver. For subsequent frames, in particular the jth frame, the following steps are executed:

Quantization: Each input vector is quantized using the MAE criterion (due to its pixel nature). The minimum distortion of the best match codeword is given by the following equation:

$$CD_i = \sum_{p=1}^{L} PD_p(x, y_i) \qquad i = 1 to N \tag{6.1}$$

where i and p represents the codeword and pixel position, respectively. The resulting label i is then compared to the SA in M^{j-1} using index-based ME.

Updating Temporary Mean Values: The input vector of the current frame, v^j which is in the subregion O_i^j, is used to update the temporary mean z_i^j of the representative vectors w_k^{j-1}:

$$z_i^j = E\{v^j, v^j \in O_i^j\} \tag{6.2}$$

Frame Reconstruction: In cases where there is a macro-block match, block exact match, block neighbor match, or block mismatch, the codebook of the previous frame W^{j-1} can be used for reconstruction. In cases where the blocks require updated means, the temporary mean Z^j are used for reconstruction.

Codebook Replenishment: The codeword $\{w_k^{j-1}\}$ is replaced by $\{w_k^j\}$:

$$w_k^j = (1 - a_k) w_k^{j-1} + a_k z_k^j \qquad k = 1 to N \tag{6.3}$$

$$w_k^j = (1 - a_k) \cdot E\{v^j | (v^j \in R_k^{j-1})\} + a_k \cdot E\{v^j | (v^j \in O_k^j)\} \tag{6.4}$$

where, $E\{v^j | (v^j \in R_k^{j-1})$ is the nearest codeword k in the (j-1)th frame, and $E\{v^j | (v^j \in O_k^j)\}$ is the temporary mean z_k^j of all input vectors in the subregion O_k^j. The term a_k is the Q-bit scalar quantized weight factor of z_k^j. For example, if there are 70 input vectors in the subregion $R_1^{'}$ and 30 input vectors in the sub-region O_1^j (Figure 6.3), then the weight factor of 0.3 is typically a 4-bit scalar quantized value. The resulting weight factor can be sent to the receiver along with the address for codeword 1, and the codeword updating flag. At the receiver, this 4-bit quantized value is converted back to the weight factor to update the codeword.

We note that in order to further reduce the bit rate, each initial codeword or temporary mean is Hadamard transformed prior to transmission. Therefore, only a fixed number of zig-zag scanned

non-zero coefficients are retained and transmitted to the receiver. The implementation of Hadamard transformation on the image is described in Appendix A. The bit rate of AVQ+ME is due to the following two factors. The first factor BR_1 is similar to the bit rate of MS-VQ technique, the second factor BR_2 is a result of:

- The additional motion vectors and flags for macro block and neighbor block matches;
- The temporary mean, its flag, its label, and its quantized weight factor.

$$BR_{AVQ+ME} = BR_1 + BR_2 \tag{6.5}$$

$$BR_1 = BR_{codebook} + BR_{mismatch} + BR_{exact-match} + BR_{updated-block} \tag{6.6}$$

$$BR_1 = \frac{NB_c}{N_p} + \frac{I(\log_2 N + F_i)}{N_f} + \frac{EF_e}{N_f} + \frac{C(\log_2 N + F_c)}{N_f} \tag{6.7}$$

$$BR_2 = BR_{macroblock-match} + BR_{neighbour-match} + BR_{tempo-mean} \tag{6.8}$$

$$BR_2 = \frac{B(M_b + F_b)}{N_b \times N_f} + \frac{M(M_m + F_m)}{N_f} + \frac{R(\log_2 N + F_r + B_c + Q)}{N_f} \tag{6.9}$$

where N = the number of codewords;

B_c = the average number of bits per codeword;

N_p = the number of pixels in the input source;

I, F_i = the number of mismatch blocks that required new labels, and the number of bits for its flag, respectively;

E, F_e = the number of exact match blocks and the number of bits for its flag, respectively;

C, F_c = the number of blocks that requires updated codewords, and the number of bits for its flag, respectively;

N_f = the number of pixels per frame;

R, F_r, Q = the number of temporary means transmitted, the number of bits for its flag, and the number of bit to quantized the weight factor, respectively. Note that R<C. In cases where the codeword is replaced by the incoming input vector, R=C. C can then be eliminated.

B, F_b, M_b = the number of macro blocks detected (with zero or non-zero motion vector), the number of bits for its flag, and the number of bits for its motion vector, respectively;

N_b = the size of the search area (4x4);

M, F_m, M_m= the number of blocks detected with non-zero motion vector, the number of bits for its flag, and the number of bits for its motion vector, respectively;

It can be seen that: $I+E+C+16B+M = (X*Y) / L = N_f / L$.

We note that, the number of blocks that require the updated codewords (C) is always less than the number of temporary means (R). If all the blocks, classified as C, are replaced by the incoming input vectors, it has three drawbacks: First, the codewords are replaced more often than required. Secondly, the input vector might not be the true representative of the blocks located in its cluster. And finally, the codebook structure which is very crucial for index-based motion estimation is not preserved. Analysis (in appendix C) has shown that the bit rate increases only when R is less than 88% of C.

In the following section, the implementation of AVQ+ME algorithm using the parallel architecture of C*RAM is presented.

6.3 C*RAM Implementation of the AVQ+ME Algorithm

The C*RAM implementation of AVQ+ME algorithm is organized into two stages: AVQ where the spatial redundancy is exploited, and index-based ME where the temporal redundancy is exploited. The implementation process is illustrated in Figure 6.4.

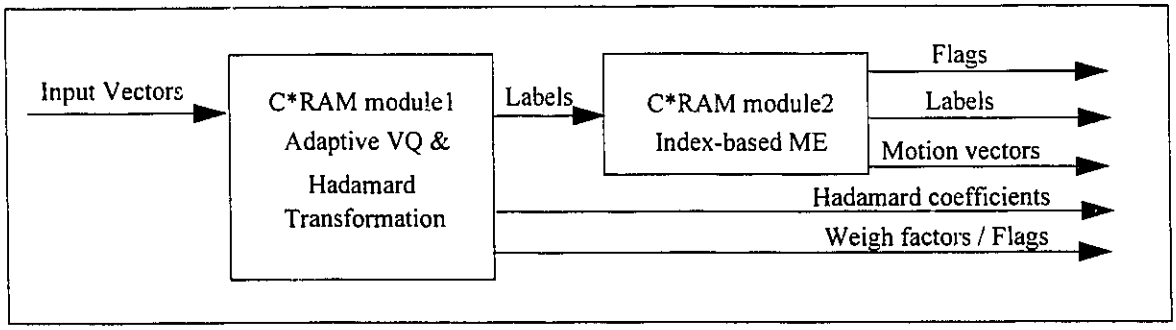

FIGURE 6.4 C*RAM Implementation of AVQ+ME

6.3.1 Adaptive Vector Quantization (AVQ) stage

In the AVQ stage, the codewords are pre-loaded into the C*RAM where all 16 pixels of each codeword are arranged in one computing unit. A codebook of 256 codewords will therefore

occupy 256 units which correspond to a module of 4 C*RAM chips. This set of 4 C*RAM chips is hereafter referred to as C*RAM module1. The encoding process is thus executed in parallel over the entire codebook. The input vectors are broadcast to all units of the C*RAM module one at a time (Figure 6.5).

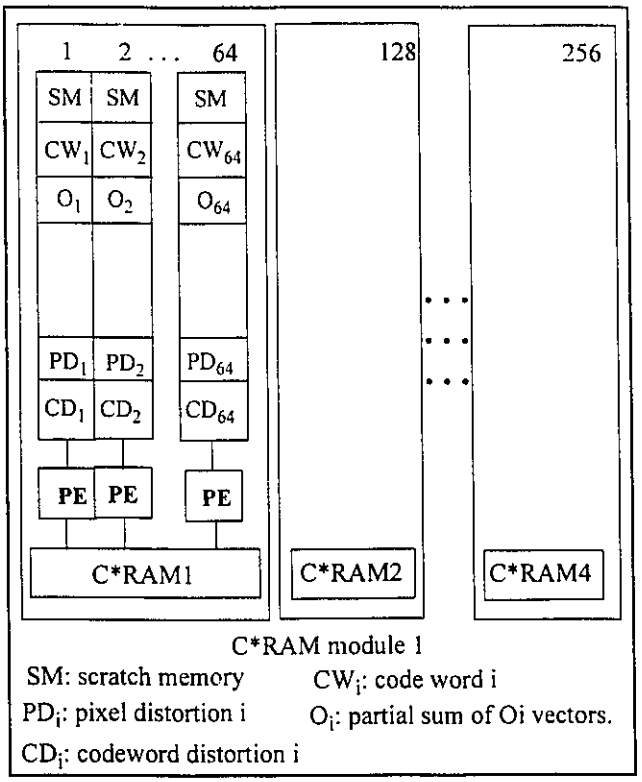

FIGURE 6.5 Codeword arrangement in C*RAM for AVQ stage

Let j, j-1 be the indices for current frame and previous frame, respectively. In each memory column of a computing unit, the memory cells are reserved for the following:

1. Scratch memory (SM): for PE address, temporary results, counters, masks for different PE operations, and the pre-computed matrix elements for Hadamard transform;
2. Codeword i(CWi): comprising 16, 8-bit pixels;
3. Partial sum of vectors lying in the subregion O_i^j.
4. Pixel distortion i (PD_i): comprising 16, 8-bit distortions between the pixels of CWi and the corresponding pixels of IV; and
5. Codeword distortion i (CD_i): comprising 12-bit sum of all 16 PD's of a particular codeword i.

The codewords, input vector, etc. are arranged similar to VQ-MAE (section 4.2), except that there are partial sums of vector lying in the subregions O_i^j and the pre-computed matrix elements for Hadamard transform. The AVQ stage is executed in two steps: frame encoding and codebook replenishment.

Frame Encoding: The PD's of every codeword and the input vectors are calculated in parallel. With a codebook of N codewords each of dimension L, the pixel distortion PD_i^j, for a given input vector v^j in the j[th] frame, and a codeword w_i^{j-1} is given by:

$$PD_p^j(x, y_p) = \left|x_p - y_{ip}\right| \qquad ,i = 1 to N \qquad p = 1 to L \tag{6.10}$$

After the PD's have been calculated, all L pixel distortions PD_{ip}^j are added to form the codeword distortion CD_i^j, and the index of the codeword which has smallest codeword distortion is chosen as the best match. The distortion of each codeword is given by:

$$CD_i^j = \sum_{p=1}^{L} PD_p^j(x, y_i) \qquad i = 1 to N \tag{6.11}$$

Codebook Replenishment: At the end of each frame, all input vectors in the O_i^j subregion are accumulated to form the temporary mean z_i^j. Codewords in computing units with no O_i^j vectors will remain unchanged, otherwise, they will be replaced by the weighted sum of z_i^j and the existing codeword. In addition, z_i^j is Hadamard transformed and the first 10 zig-zag scanned Hadamard coefficients along with the weight factor are sent to the receiver. The updated codebook will be used for encoding the next frame

6.3.2 Index-based Motion Estimation (ME) stage

In the ME stage, a search for best match label with either zero or non-zero motion vector is executed. A 360 x 288 frame will generate 90 x 72 8-bit labels and hence requires 90 x 576 memory cells. C*RAM module2 is thus used as the label memory. In general, the uncompared labels of frame M^{j-1} are always stored in the label memory. The labels of the current frame M^j overwrites

the labels in M^{j-1} after they have been compared. The implementation of ME using C*RAM is illustrated in Figure 6.6. A few implementation concepts are used to facilitate the presentation.

- Mask1 is used to mask out the label being compared.
- Mask2 is used to define the horizontal span of SA being used on the 8-bit label memory, while the vertival span of SA is indicated by the controller via row addresses.
- The dirty bit Ki is used to indicate whether there is a block mismatch or a block which requires an updated mean in the corresponding column of SA. If Ki is 0, then the macro block search can be skipped.

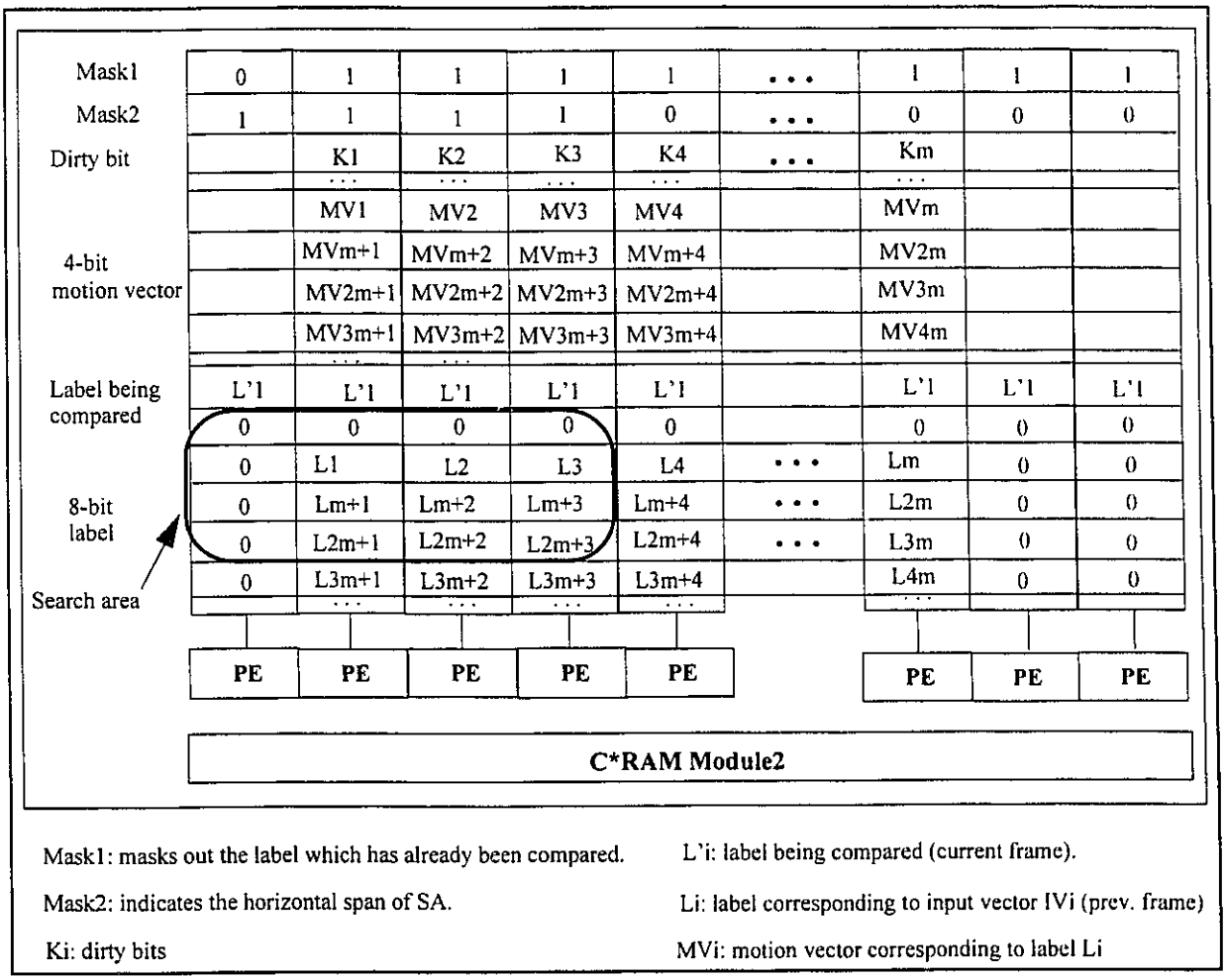

FIGURE 6.6 Label arrangement in C*RAM for ME stage

- The 4-bit motion vector section stores the motion vector of the label being compared in M^{j-1} in a predetermined order. (Note that, for each label, there exists at least one motion vector in the SA).
- The label being compared is broadcast to all columns of C*RAM module2, and only those specified by Mask2 is involved in the search process. This label is saved for the ME of next frame; and
- The 8-bit label section stores the labels of M^{j-1}.The ME stage is performed as follow:

1. Block search procedure: Let m, n, and L'1 be the numbers of labels in the 2 (horizontal and vertical) dimensions of a frame, and the first label of the current frame corresponding to the input vector IV1, respectively.

- Step 1: L'1 is first broadcast to all columns of C*RAM module2. Then, only the columns which has the 1's in Mask2 are involved in the search. As shown in Figure 6.6, the SA, for L'1, includes the following {{0, 0, 0, 0} {0, L1, L2, L3} {0, Lm+1, Lm+2, Lm+3} and {0, L2m+1, L2m+2, L2m+3}}. After the search, the resulting motion vector is stored in MV1. The value of K1 is preset at 1 and only reset if any of {MV1, MVm+1, MV2m+1, MV3m+1} indicates a block mismatch or a block which requires an updated mean. The first position in Mask1 is reset.
- Step 2: Mask2 is shifted once to the right defining the new SA. L'2 is then broadcast to all columns except the first one (being masked by Mask1). The new SA, for L'2, includes {{0, 0, 0, 0} {L1, L2, L3, L4} {Lm+1, Lm+2, Lm+3, Lm+4}, and {L2m+1, L2m+2, L2m+3, L2m+4}}. The resulting motion vector is stored in MV2, and the dirty bit K2 is updated accordingly. The second value of Mask1 is then reset. This step is repeated until the label L'm has been compared.
- Step 3: For label L'm+1: L'm+1 is broadcast to and stay at the position right below L'1. The resulting motion vector is stored in MVm+1, and the dirty bit is, however, stored in K1. This procedure is repeated until L'4m has been compared.

2. Macro-block search procedure: Once the L'4m has been compared, the macro-block search begins by first looking for the 4 consecutive 1's in Ki. If it is the case, then corresponding motion

vector is search for unique value. The macro block search is repeated until all 4m motion vectors are processed. And the block search, the macro block search are repeated until the end of frame.

We note that as the ME stage proceeds, the label memory stored in C*RAM module2 is shifting up and to the left. Currently, 2 C*RAM chips are used for simulation involving 24 frames of Miss America sequence. We also note that in this ME stage, the need for the storage of 2 label memory frames has been avoided.

We recall from section 6.2 that output of the ME stage can be classified into one of the following scenarios: block exact-match (with zero motion vector), block neighbor-match (with non-zero motion vector), block mismatch (new label required), block mean updating, and macro block match. In a block exact match, only flag **e** is sent, whereas in a block neighbor match, flag **m** and the corresponding motion vector are transmitted. When a new label is required (as in block mis-match), flag **i** and the corresponding label are sent. When a block requires an updated codeword, flag **c** and the label of the updated codeword are sent. A macro-block match will results in the transmission of flag **b** and the corresponding motion vector to the receiver. At the end of each frame, flag **r**, the retained Hadamard coefficients of the temporary mean, its corresponding weight factor, and its label are transmitted to the receiver to reconstruct the frame and update codebook. Since, the probability of the flags can be predicted, a Huffman table is used to assign bit patterns to different flags. A typical assignment is summarized in the table below:

TABLE 6.1 Bit assignment for flags

flag	pattern
e	0
m	10
i	110
c	1110
r	11110
b	11111

6.4 Performance Analysis

6.4.1 Bit rate performance

In the first experiment, the rate-distortion of AVQ+ME is compared with SCR-VQ[76]. Simulations have been performed with the first 24 frames of Miss America sequence using codebooks of sizes 512, 256, 128, and 64 codewords, and the pixel threshold Δ of values 16, 20, 24, and 24, respectively. The SCR-VQ (section 2.3.5) is modelled using thresholds of 256, 320, 384, and 384 (expressed in the MSE sense) corresponding to the 4 codebook sizes. The rate-distortion curves of the two techniques are shown in Figure 6.7. We note that the highest point of each graph corresponds to the average of bit rate and PSNR over the first 24 frames of Miss America sequence using codebook of size 512. Figure 6.7 shows that the rate-distortion curve of the AVQ+ME technique is better than SCR-VQ. For example, at a PSNR of 36.83dB, the bit rate of AVQ+ME is 0.37bpp compared to 0.61 of SCR-VQ. This is equivalent to a reduction in bit rate as high as 34%. This confirms that the AVQ+ME successfully exploits the temporal redundancy in the video sequence.

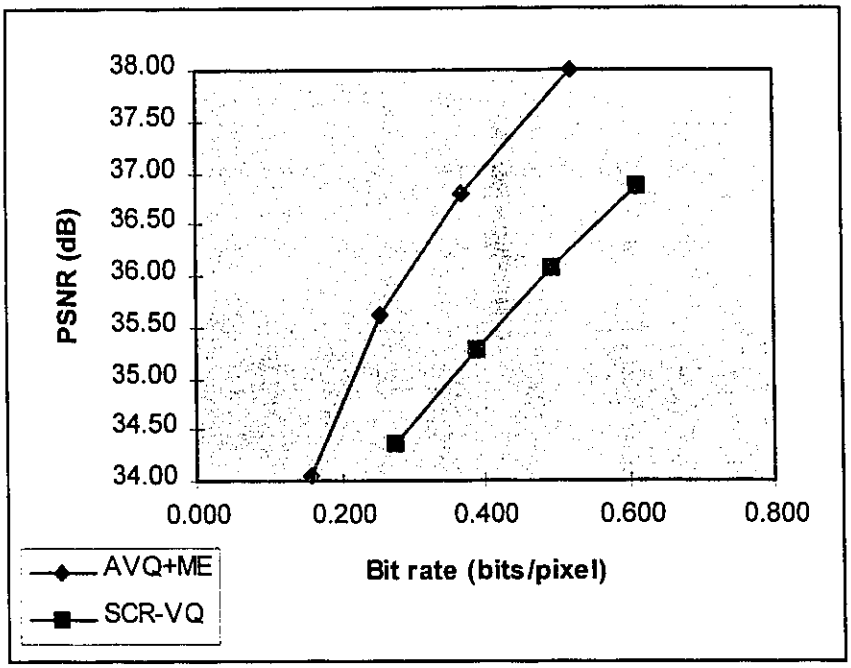

FIGURE 6.7 Rate - Distortion Curves of AVQ+ME vs. SCR-VQ

In the second experiment , the coding performance of AVQ+ME is compared with the MPEG-1 coder (Figure 6.8). The AVQ+ME has been simulated, on the first 24 frames of Miss America sequence with a codebook of 256 codewords at pixel threshold equal to 20. The resulting bit rate is 0.37bpp compared to 0.40bbp for the MPEG-1 coder. Moreover, the resulting PSNR of AVQ+ME is, on an average, 0.95dB better than the MPEG-1 coder. Once again, it can be seen that AVQ+ME outperforms the MPEG-1 coder and maintains a relatively constant quality. The original frames 0, 12, and 23 of Miss America sequence are shown in Figure 6.11. The corresponding reconstructed frames using AVQ+ME are shown in Figure 6.12.

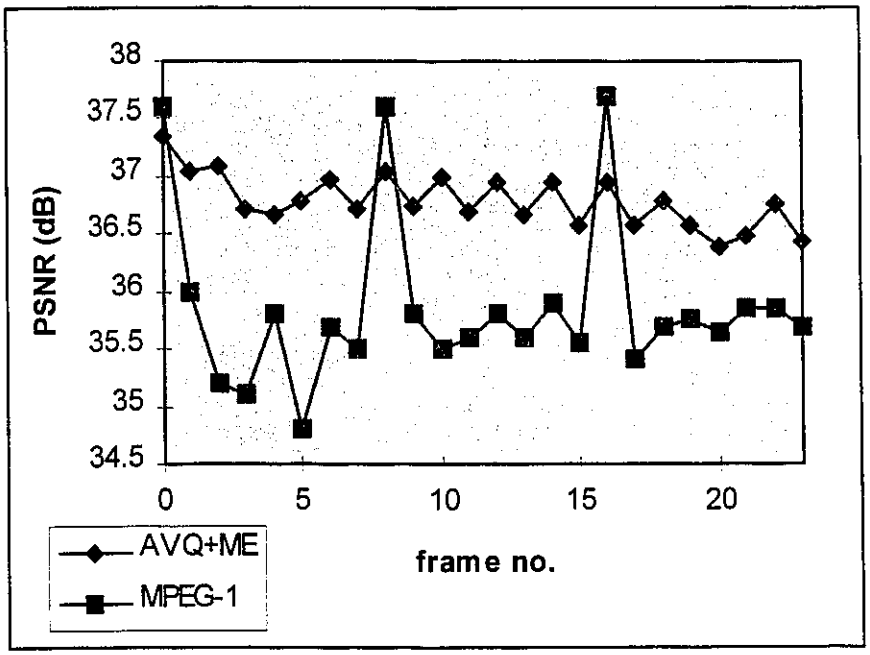

FIGURE 6.8 AVQ+ME and MPEG comparison.

In the third experiment, AVQ+ME is compared with IHA-VQ recently reported in the literature [74]. Simulations have been performed on the Miss America sequence from frame 13 to frame 97 with a step size of 3. The codebook is chosen of size 512. The simulation results are shown in Figures 6.9 and 6.10. The PSNR of AVQ+ME is marginally lower than that of IHA-VQ (Figure 6.9), but it maintains a relatively constant quality throughout the sequence. It can be seen in Figure

6.10 that both coders require an average bit rate of 0.60bbp. The AVQ+ME, however maintains at a constant bit rate compared to the increasing bit rate by IHA-VQ.

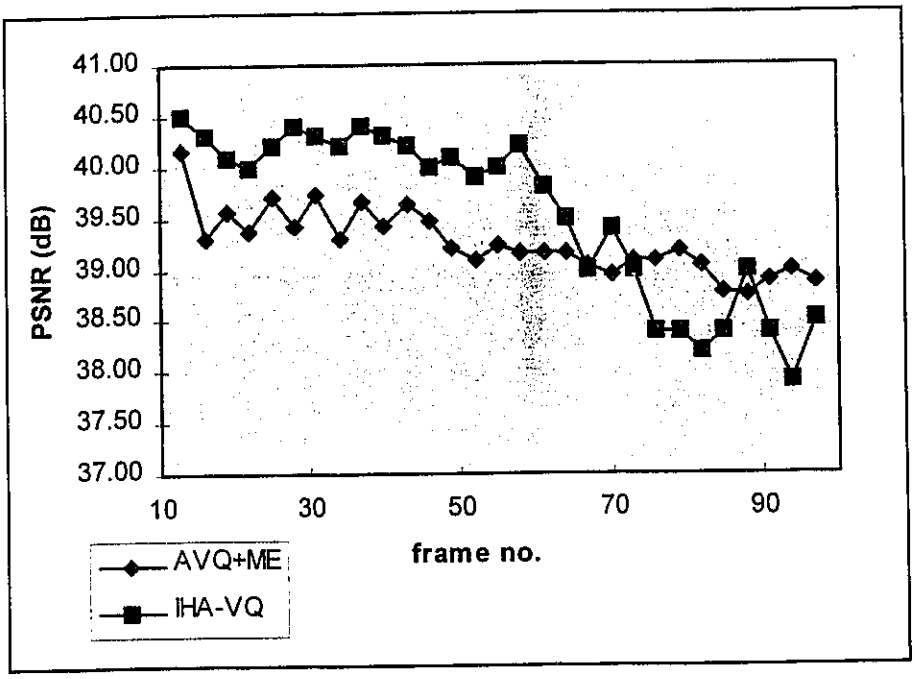

FIGURE 6.9 PSNR comparison: AVQ+ME vs. IHA-VQ

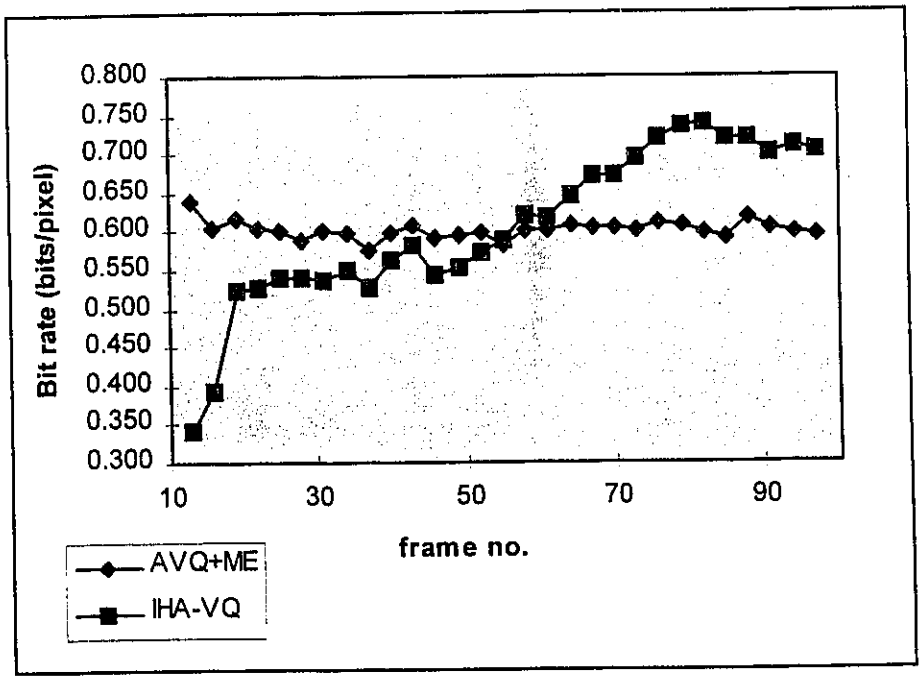

FIGURE 6.10 Bit rate comparison AVQ+ME vs. IHA-VQ

6.4.2 Execution time

Simulations have been performed with the first 24 frames of Miss America sequence using codebooks of sizes 512, 256, 128, and 64 codewords, and the pixel threshold Δ of values 16, 20, 24, and 24, respectively. The complete break-down of the execution time of the two stages in AVQ+ME are tabulated in the following tables:

TABLE 6.2 Frame execution time of AVQ stage (Miss America sequence)

Codebook size	PD Computation	Quantization (Q)	CB Updating Normalized to K	Indexing (I)	Total Time (cycles)
512	432 x K	629 x K	98 x K	9 x K	1168 x K
256	432 x K	629 x K	84 x K	8 x K	1153 x K
128	432 x K	629 x K	75 x K	7 x K	1143 x K
64	432 x K	629 x K	82 x K	6 x K	1149 x K

TABLE 6.3 Frame execution time of ME stage (Miss America sequence).

Codebook size	Motion Vector Determination	Macro Blk Matching Normalized to K	Other C*RAM control operations	Total Time (cycles)
512	112 x K	5 x K	10 x K	127 x K
256	100 x K	5 x K	10 x K	114 x K
128	88 x K	5 x K	10 x K	101 x K
64	76 x K	5 x K	10 x K	88 x K

In Table 6.2, all the columns, except column codebook (CB) updating, are similar to VQ-MAE. The codebook updating column shows the average number of cycles to update a codeword in that particular simulation. The large numerical value in the total time column shows the average number of cycles required to encode each input vector including updating the codeword in that simulation.

In Table 6.3, the motion vector determination requires the largest number of cycles compared to other columns. The numerical value in the total time column indicates the average number of cycles to process a label.

The execution time of the two stages in AVQ+ME, when using codebook of size 256, is calculated as follow: In the AVQ stage, it takes 262 ms to encode a 360 x 288 pixels frame. In the ME stage, it takes 26 ms to compare the labels of the frame. Hence, if AVQ and ME stages are organized in a pipeline fashion, the total encoding time for a 360 x 288 frame, using a codebook of size 256 codewords, is 262 ms . Therefore, real time implementation is possible with 8 C*RAM module1's arranged in parallel.

6.5 Summary

In this chaper, an adaptive VQ technique for video compression has been proposed. This technique provides a competitive coding performance compared to the other techniques reported in the literature. Most importantly, the AVQ+ME coder is best suited for constant quality and constant bit rate applications. The proposed coder has been implemented using the C*RAM structure. Real-time implementation is possible using 8 C*RAM modules.

FIGURE 6.11 Original frames 0, 12, and 23 of Miss America sequence, size 360 x 288 pixels.

FIGURE 6.12 Frames 0, 12, and 23 reconstructed using AVQ+ME with average bit rate = 0.6bbp

CHAPTER 7

Summary and Future Work

7.1 Summary

In this thesis, the VQ-based image and video compression algorithms have been mapped on the C*RAM architecture. Some new algorithms have also been proposed. First, VQ-based algorithm using MAE as selection criteria (VQ-MAE) is implemented on the C*RAM. The VQ-MAE technique eliminates the need to have the squaring operation thus computational complexity is greatly reduced. Moreover, if the 1's complement is used in the absolute difference operation, image degradation is considerably imperceptible while the execution time is noticeably reduced. VQ-MAE using 1's complement is, therefore, chosen for intraframe coding.

Secondly, the Modified SVQ algorithm is proposed and implemented on the C*RAM. The algorithm enables the FCB and RCB's to be arranged on the C*RAM's parallel structure. The main advantage of Modified SVQ is that different image resolutions can be reconstructed from the same bit stream used for the full resolution image.

Finally, the AVQ+ME algorithm is proposed and implemented. AVQ+ME achieves relatively constant quality and constant bit rate over the time. We also note that AVQ+ME can be a strong

competitor to H.261 in terms of real-time and bandwidth requirements. For instance, if 15 QCIF frames are encoded per second, the execution time and the required bandwidth are:

- 15 frames * 1584 blocks/frame * 1267 cycles/block * 35ns/cycle ~ 1.0s.
- for luminance: 15 frames/sec * (176*144)pixels/frame * 0.6 bits/pixel = 3.5 x 64Kbits/sec
- for chrominance: 2* 15 frames/sec * (88*72)pixels/frame * 0.6 bits/pixel = 1.8 x 64Kbits/sec

The bandwidth required, therefore, 6 x 64Kbits/sec which is within H.261's specification.

7.2 Future Work

C*RAM has undergone a number of changes and needs more changes to keep up with the increasing image/video compression demand.

On the architecture side, the future C*RAM should be able to:

- Accept data directly from the host without transposing;
- Increment/decrement in each memory column without slowing down the global operation; and
- Have two ports: while input vector is arriving at one port, the label of the previously encoded input vector is leaving the other port.

On the algorithm side:

- A SVQ algorithm for video compression using C*RAM is one of the future projects;
- If SVQ is not considered, the MD codebook should be used to increase frame quality; and
- The AVQ+ME should have a mechanism to detect scene change and to generate a new codebook accordingly in order to adapt well to the statistics of the sequence.

In conclusion, C*RAM can be used to encode image with spatial scalable feature in real-time. Moreover, the C*RAM implementation of VQ for video sequence also demonstrates a competitive real-time coding performances over other techniques: relatively constant quality and constant bit rate throughout the video sequence.

APPENDIX A

Computer Arithmetics for C*RAM

In this appendix, the computer arithmetics used in C*RAM (Cs64p1K) are discussed. The-C*RAM basic functions, register functions, user-defined functions are reviewed [109]. Then the macros for C*RAM are presented, such as FULL_ADDER/SUBTRACTOR, EXACT_MATCH, INEXACT_MATCH, ONE_COM, etc. The main section of the appendix is the different algorithms for implementing VQ-based algorithms using C*RAM structure, including Hadamard image transformation.

A.1 C*RAM Functions

A number of notations and assumptions have been used throughout the appendix.

- The notation A' implies the complement of A.
- The for, while, and if-statements are C*RAM controller statements which act upon all the PE's in parallel.
- In memory READ and WRITE operations, the column address is specified only when specific computing unit is of interest, otherwise they are omitted and understood as all PE's are participating in the current operation.

A.1.1 Basic functions

The following is the table of basic C*RAM functions with 3 operands X, Y, and M. These functions are assumed for 3 operands operation. The truth table for 2-operand operations can easily be derived.

TABLE A.1 Basic functions

Y	X	M	AND	NAND	OR	NOR	XOR	XNOR
0	0	0	0	1	0	1	0	1
0	0	1	0	1	1	0	1	0
0	1	0	0	1	1	0	1	0
0	1	1	0	1	1	0	0	1
1	0	0	0	1	1	0	1	0
1	0	1	0	1	1	0	0	1
1	1	0	0	1	1	0	0	1
1	1	1	1	0	1	0	1	0

A.1.2 Register functions

Beside the basic functions, a number of register functions are defined for C*RAM:

- LD_REG(X,Y,WE;M[row]): loads the registers X's, Y's, or WE's with the value in the memory location specified by the row address in M; This function is used to load the mask (stored in memory) to the registers.
- LD_MEM(M[row];X,Y): loads the memory location specified by row address in M with the content of the registers X's, or Y's; This function is used to store the temporary results to the memory.
- SET_ REG(X,Y,WE): sets the values of (all) X's, Y's, or WE's to 1.
- RST_REG(X,Y,WE): resets the values of (all) X's, Y's, or WE's to 0.
- INV_REG(X,Y,WE): negates the values of (all) X's, Y's, or WE's.
- SHIFT_RT(Y): the result in the result line from the PE's right adjacent neighbour is written into Y register.

- SHIFT_LF(X): the result in the result line from the PE's left adjacent neighbour is written into X register.
- ONES(M[row]): broadcasts a 1 to memory location specified by row address in M.
- ZERO(M[row]): broadcasts a 0 to memory location specified by row address in M.
- WIRE_OR(X,Y,M[row]): wire-OR the results in registers X, Y, or memory location specified by row address in M; This function is commonly used to check the status of a search (if result=1: a match occurs; if result=0: a mismatch occurs).

A.1.3 User-defined functions

Sometimes, user-defined functions are required as in the addition and subtraction operation. Here the previous carry (of addition) and the previous borrow (of subtraction) can be stored in register X. Assuming M+Y and M-Y.

TABLE A.2 User defined functions

M	Y	X	CARRY	BORROW
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

A.2 C*RAM Macros

In this section, pseudo-coded macros have been used for C*RAM implementation are discussed. Most of the macros have included the numbers of memory cycles required. Unless otherwise stated, N is the number of bits representing the number of levels in a monochrome image. Since there are 256 gray levels, N is 8. The bit position starts from MSB bit 7 (horizontally left-most or vertically highest) to LSB bit 0 (horizontally right-most or vertically lowest).

1. Exact_match: This macro takes **1 + 3*N cycles** for comparing two N-bit numbers A and B, which is in the memory. Note that there is no memory WRITE so that WE is not involved. X and Y registers are for final and temporary results, respectively. If one of the argument is broadcasted on-line then it only takes **1 + 2*N cycles** since the load register Y is not required.

```
EXACT_MATCH(A,B) {vector A={a_i} is compared with vector B={b_i} in the memory}
    SET_REG(X) {assuming all candidates match}
    for i = 0 to N-1 (by 1)
        LD_REG(Y, a_i)
        Y = XNOR(Y,b_i)
        X = AND(X,Y)
    endfor
```

2. Greater_than: This macro adopts the algorithm proposed by Chu [110] for CAM. Both the search argument B and vector A are in the memory. Let s, p be the number of zero's in the search argument and the average number of cycles for completing the exact_match, respectively. Then it takes **s*(2+p) + N cycles** to conduct the search. We note that the search argument has to be in memory for the exact_match process, it cannot be down-loaded on-line from the controller.

```
GREATER_THAN(A,B) {vector A={a_i} is compared with the search argument B={b_i} in the memory}
    for i = N-1 to 0 (by -1)
        if (WIRE-OR(b_i)==0)
            ONES(b_i )
            EXACT_MATCH(A_i,0,B_i,0); (p cycles)
            ZERO(b_i )
        endif
    endfor
```

3. Less_than: Similar to greater_than search, this macro was also adopted from Chu's algorithm for CAM. Both A and B are in memory. It takes **s*(2+p) + N cycles** to complete the search.

```
LESS_THAN(A,B) {vector A={a_i} is compared with the search argument B={b_i} in the memory}
    for i=N-1 to 0 (by -1)
        if (WIRE-OR(b_i)==1)
            ZERO(b_i )
            EXACT_MATCH(A_i,0,B_i,0); (p cycles)
```

```
            ONES(b_i )
        endif
    endfor
```

4. Less_than_CRAM: This macro is written solely for C*RAM word-parallel bit-serial computation, it takes **1+N cycles** to complete. The search argument P is read bit by bit on-line from the controller. This method is better than less_than (CAM) search because the search argument can be provided by the controller and no memory access for the search argument is required. The result is stored back to the Mask.

```
LESS_THAN_CRAM(P,B) {the search argument P={p_i} is compared with vector B={b_i} in the memory}
    SET_REG(X)
    LD_REG(WE,Mask)
    for i=N-1 to 0 (by -1)
        if (p_i==0)
            if(WE=1)
                Mask = b_i'
            WE,X= AND(X,Mask)
        else
            WE,X= AND(X,b_i)
        endif
    endfor
```

5. One-bit subtractor: This macro adds a value of 2^i to the number. It is never used. It takes **3 + 2*(N-1-i) cycles**

```
SUB_ONE(i,B) {subtraction of 2^i from vector B in the memory}
    WE,Y = Mask
    b_i = XOR(Y,b_i)
    SET_REG(X)
    for n = i to N-2
        WE,X = AND(X,b_n')
        b_n+1 = XOR(Y,b_n+1)
    end for
```

6. One-bit adder: This macro adds a value of 2^i to the number. It takes **3 + 3*(N-i) cycles**

```
ADD_ONE(i,B) {addition of 2^i to vector B={b_i} in the memory}
    Y = M'ask
    LD_REG(X,b_i)
    SET_REG(WE)
    for n = i to N-1 (by 1)
        b_i = XOR(X,Y)
        Y,WE = AND(X,Y)
        LD_REG(X, b_i+1)
    endfor
```

7. Maximum search [99]: This macro searches for the maximum value among the PE's. It takes **1+ (2+if)*N cycles.** The result is stored in X.

```
MAX(B) {N: the length of vector B={b_i} to be maximum-searched}
    SET_REG(X)
    for i = N-1 to 0 (by -1)
        Y = AND(X,b_i)
        if (WIRE_OR(Y) == 1)
            X = Y
        endif
    endfor
```

8. Minimum search [99]: This macro searches for the minimum value among the PE's. It takes **1+ (2+if)*N cycles.** The result is stored in X.

```
MIN(B) {N: the length of vector B={b_i} to be minimum-searched}
    SET_REG(X)
    for i = N-1 to 0 (by -1)
        Y = AND(X,b_i')
        if (WIRE_OR(Y) = 1)
            X = Y
        endif
    endfor
```

9. Two's complement: **2 + 2*(N-1) cycles**

```
TWO_COM(B) {vector B in the memory is two's complemented}
    ZERO(WE,X) /* at previous stage */
    Y = b_N' /* b_N' is the mask */
    b_0 = XOR(X,b_0) /* not needed */
    for i = 0 to N-2
        WE,X = OR(X,b_i)
        b_i+1 = XOR(Y,b_i+1)
    end for
```

10. One's complement: This macro generates the one's complement of the argument. It takes **N + 2 cycles.** The vector B is only one's complemented if and only if the bit B_N is a 1 which implies that B is negative.

```
ONE_COM(B) {vector B in the memory is one's complemented}
    ONES(WE)
    Y = b_N' /*b_N' is the mask*/
    for i = 0 to N-1 (by 1)
        b_i = XOR(b_i,Y)
    endfor
```

11. Full adder: This macro adds the input vector to the codeword and store the sum in the memory. It takes **1 + 3*N cycles**

```
FULL_ADD(Mcw,Y) {N = number of bits of Ms; X=carry; Y=input vector; Mcw=codeword; Ms = sum}
    LD_REG(X, M_previous carry or 0)
    for i = 0 to N-1
        Y = iv_i
        Ms_i = XOR(X,Y,Mcw_i)
        X = CARRY(Mcw_i,Y,X)
    endfor
```

12. Full subtractor: This macro subtracts the input vector from the codeword and store the difference in the memory. It takes **1 + 3*N cycles**

```
FULL_SUB(Mcw,Y) {N = number of bits of Md; X=borrow; Y=input vector; Mcw=cw; Md = difference}
    LD_REG(X, M_previous borrow or 0)
```

```
for i = 0 to N-1
    Y = iv_i
    Md_i = XOR(X,Y,Mcw_i)
    X = CARRY(Mcw_i',Y,X) /*or X = BORROW(Mcw_i,Y,X) */
endfor
```

A.3 C*RAM Algorithms

In this section, pseudo-coded algorithms have been used for C*RAM implementation are discussed.

A. Load codewords and extra bits into C*RAM: We note here that since data written from the Data Transposer (DT) to the C*RAM are in blocks of 8 words, they must be read in 32 times (strips) to completely fill the 256 computting units. Therefore, it takes **32*16*8 = 4096 cycles** to load all the 256 codewords into C*RAM:

```
for strip = 0..31
    for pixel = 0..15
        for bit = 0..7
            M[base0 + bit + 9*pixel, 8*strip] = DT(a,b)
        endfor
    endfor
endfor
```

The extra bits are used to determine whether the pixel difference is negative. 1 is appended next to the MSB of the codewords and 0 next to the MSB of the input vectors. It takes **2*16 = 32 cycles** to load.

```
for pixel = 0..15
    M[base0 + 8 + 9*pixel, 8*strip] = 1 /*for codewords */
    M[base1 + 8 + 9*pixel, 8*strip] = 0 /*for input vectors */
endfor
```

B. Image histogram:

1) 256 graylevels are prestored in 256 coding units.

2) Each bit of the pixels is sequentially loaded into C*RAM. The cycles can be saved by broadcasting and matching at the same time.

3) The bit of the pixels are exact-matched with the corresponding one in the coding unit.

1+2*8 = 17 cycles/pixel

4) When a coding unit detects a match, it increases it owns counters by 1. The counter should ideally have 18-bit precision.

1+3*18 = 55 cycles/pixel

Total cycles = 512 * 512 * (17 + 55) = 18,874,368 cycles

Total time = 18,874,368* 35ns = 661ms.

C. Adjacent codeword averaging: A and B are the codevectors from the right and left of the current PE, respectively. This algorithm is written but never used.

LD_REG(WE,M_{ask})

ZERO(M_{carry})

For i = 0 to N-1 (by 1) {add the two adjacent numbers}

LD_REG(Y,a_i)

SHIFT_RT(Y)

LD_REG(X,b_i)

SHIFT_LF(X)

Ms = XOR(X,Y,M_{carry})

M_{carry} = CARRY(M_{carry},Y,X)

endfor

For i = 1 to N (by 1) {Divide by 2, repeat for N-1 bits. [2*(N-1)]}

LD_REG(X,Ms_i)

LD_MEM(Ms_{i-1},X)

endfor

Total cycles = 2 + 6*N + 2*N = 66 cycles

Total time = 66 * 35ns = 2.31 um

D. Image Transformation: In this section, the mapping of image transformation on the SIMD architecture is proposed. As a C*RAM application, the Hadamard transform has been implemented. An image transformation can be expressed as:

$$V = AUB \tag{A.1}$$

Let $V = \begin{bmatrix} v_1 & v_2 \\ v_3 & v_4 \end{bmatrix}$, $U = \begin{bmatrix} u_1 & u_2 \\ u_3 & u_4 \end{bmatrix}$, $a = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix}$, and $B = \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix}$, we then have

$$\begin{bmatrix} v_1 & v_2 \\ v_3 & v_4 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \begin{bmatrix} u_1 & u_2 \\ u_3 & u_4 \end{bmatrix} \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} \tag{A.2}$$

By expanding the left hand size, we obtain:

$$\begin{bmatrix} v_1 & v_2 \\ v_3 & v_4 \end{bmatrix} = \begin{bmatrix} u_1a_1b_1 + u_2a_1b_3 + u_3a_2b_1 + u_4a_2b_3 & u_1a_1b_2 + u_2a_1b_4 + u_3a_2b_2 + u_4a_2b_4 \\ u_1a_3b_1 + u_2a_3b_3 + u_3a_4b_1 + u_4a_4b_3 & u_1a_3b_2 + u_2a_3b_4 + u_3a_4b_2 + u_4a_4b_4 \end{bmatrix} \tag{A.3}$$

This transformation is mapped onto the SIMD architecture as follows: First, the products a_ib_j are precomputed and stored in the respective computing units. Secondly, since the order of the transfrom coefficients is known in advance, they can also be ordered in the zig-zag manner. The operation can be shown in Figure A.1 below:

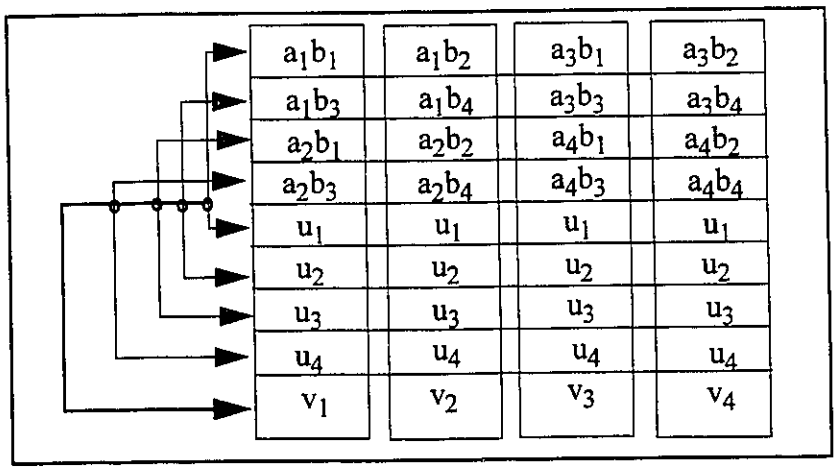

FIGURE A.1 Image Transformation implemented on a SIMD architecture

As an example, Hadamard transform is implemented on the C*RAM as follows:

The input vectors are broadcast into C*RAM, pixel by pixel.

An input vector of 4 x 4 pixels forms a 1-D array of 16 pixels.

The Hadamard coefficients are preloaded in the C*RAM with 0 representing an ADDITION and, 1 representing a SUBTRACTION.

The bits of the pixels can be broadcasted and added (or subtracted) at the same time.

The result is right-shifted once (divide by 2) and stored in the coefficients i, where i = 0..15.

The memory map for executing Hadamard transform can be shown in the following figure:

TABLE A.3

Zig-Zag order	**1**	**2**	**4**	**7**	**3**	**5**	**8**	**11**	**6**	**9**	**12**	**14**	**10**	**13**	**15**	**16**
Coefficients	s0	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15
1st op.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2nd op.	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
3rd op.	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
4th op.	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
5th op.	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
6th. op.	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0
7th op.	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
8th op.	0	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1
9th op.	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
10th op.	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0
11th op.	0	0	1	1	0	0	1	1	1	1	0	0	1	1	0	0
12th op.	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1
13th op.	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
14th op.	0	1	0	1	1	0	1	0	1	0	1	0	0	1	0	1
15th op.	0	0	1	1	1	1	0	0	1	1	0	0	0	0	1	1
16th op.	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0
pixel no. 0	p0	p0	p0	p0	p0	p0	p0	p0	p0	p0	p0	p0	p0	p0	p0	p0
pixel no.1	p1	p1	p1	p1	p1	p1	p1	p1	p1	p1	p1	p1	p1	p1	p1	p1
pixel no.2	p2	p2	p2	p2	p2	p2	p2	p2	p2	p2	p2	p2	p2	p2	p2	p2
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
pixel no. 15	p15	p15	p15	15p	p15	p15	p15	p15	p15	p15	p15	p15	p15	p15	p15	p15
	PE 00	PE 01	PE 02	PE 03	PE 04	PE 06	PE 06	PE 07	PE 08	PE 09	PE 10	PE 11	PE 12	PE 13	PE 14	PE 15

We note that the coefficients can also be zig-zag ordered in advance. The matrix multiplication of Hadamard transform can be executed as follows:

```
{N = number of bits of Ms; X=carry; Y=input vector; Ms = sum}
for pixel = 0..15
    ZERO(X)
    for i = 0 to N-1 /*i is the bit position */
        Y = iv_i /* = 0 if i>=N */
        Ms_i = XOR(X,Y,Ms_i)
        Ms_i = XOR(M[operation i],Ms_i) /* this will negate Ms_i if M[operation i] = 1 */
        X = CARRY(Ms_i,Y,X)
    endfor
endfor
```

If the sum has 12-bit long, it takes 16*(1+4*12) = 784 cycles for loading and calculating; and about 2*12 ~ 24 cycles for unloading more than 8 transform coefficients.

E. Image compression technique: **Mean Absolute Error (VQ-MAE)**

1) Broadcast and subtract the corresponding pixels of the codeword from the input vector, and the result is stored in M[base2]. Then, check if (b_8 of the pixel difference=0), if so 1's complement that pixel. It takes 16*(1+9*2+8) = 432 cycles.

```
for pixel = 0..15
    WE=1
    for bit = 0..8
        Y = bit of the pixel of the input vector from controller
        FULL_SUB2(M[base0+9*pixel],Y, M[base2+9*pixel]) /* 2 cycles */
        if (bit=8)  WE= (carryout)' /*this is performed once in parallel with FULL_SUB2 */
    endfor
    for bit = 0..7
        ONE_COM(M[base2+9*pixel])
    endfor
endfor
```

2) Add all 8-bit pixel differences (without the bit b_8). The result is stored in M[base3]. After that apply the mininum search to the sum. The total time is 16*(1+3*12) + 1+3*12 = 629 cycles.

```
for pixel = 0..15
    FULL_ADD(M[base2+9*pixel], M[base3])
endfor
MIN(M[base3], X, 10) /* result is stored in X register */
```

3) Output index: 8 cycles

```
OUT_INDEX(X)
```

Performance measures:

Total number of cycles = 432 + 629 + 8 = 1069 cycles

Total time for block encoding: 1069 cycles ~ 37.4 usec (average cycle time = 35ns)

Memory usage (each column): (16+128+16+128+12)/1024 ~ 29.3%.

F. Video compression technique: **AVQ+ME**

step 1: Frame Encoding (similar to VQ-MAE)

1) Broadcast and subtract the corresponding pixels of the codeword from the input vector, and the result is stored in M[base2]. Then, check if (b_8 of the pixel difference=0), if so 1's complement that pixel. It takes 16*(1+9*2+8) = 432 cycles.

```
for pixel = 0..15
    WE=1
    for bit = 0..8
        Y = bit of the pixel of the input vector from controller
        FULL_SUB2(M[base0+9*pixel],Y, M[base2+9*pixel]) /* 2 cycles */
        if (bit=8)  WE= (carryout)' /*this is performed once in parallel with FULL_SUB2 */
    endfor
    for bit = 0..7
        ONE_COM(M[base2+9*pixel])
    endfor
endfor
```

2) Add all 8-bit pixel differences (without the bit b_8). The result is stored in M[base3]. After that apply the mininum search to the sum. The total time is 16*(1+3*12) + 1+3*12 = 629 cycles.

```
for pixel = 0..15
    FULL_ADD(M[base2+9*pixel], M[base3])
endfor
MIN(M[base3], X, 10) /* result is stored in X register */
```

3) Output index: 8 cycles

```
OUT_INDEX(X)
```

Performance measures:

Total number of cycles required for encoding 1 input vector = 432 + 629 + 8 = 1069 cycles

Total time for block encoding: 1069 cycles ~ 37.4 usec (average cycle time = 35ns)

Total frame execution time for step 1: 1069cycles * 35ns * 6480 = 242ms

step 2: codebook replenishment

Let B_1 be the numbers of bits of the partial sum of each O_i's region,

B_2 the number of bits for the counter for O_i's region,

B_3 the number of bits for the counter for R_i's region,

C the number of input vectors required the updated codewords, and

R the number of codewords being updated, respectively.

For a 360 x 288 pixels image, there are 90 x 72 = 6480 blocks; B_1 is at most $\log_2(255*6480) = 21$ bit long, B_2, B_3 are at most $\log_2(6480) = 13$ bit long.

In the simulation with Miss America sequence, B_1 is chosen to be 12, B_2 and B_3 are chosen to be 7. And C and R are typically 150 and 90, respectively.

For each input vector, the following operations are required:

If the input vector belongs to R'_i, increment counter(R'_i);

$1*(1+3N)$, where $N = B_3$. (22 cycles)

Else (If the input vector belongs to O_i), increment counter(O_i), and Update partial sum;

$1*(1+3N) + 16*(1+3S)$, where $N = B_2=7$; $S=B_1=12$ (22+16*37 = 614 cycles)

At the end of each frame:

Divide partial sum O_i by counter for O_i (in parallel). Division is executed using comparison and subtraction.

```
for pixel = 0..15
    for bit = 0..8
        compare{SUM[11-bit,8-bit], COUNTER[3,0]}
        if SUM[11-bit,8-bit] >= COUNTER[3,0]
            WE,QUOTIEN[8-bit] = 1
        else
            WE,QUOTIEN[8-bit] = 0
        subtract{SUM[11-bit,8-bit], COUNTER[3,0]}
    endfor
endfor
```

$16*[(B_1-B_2)*B_2 + (B_1-B_2)*(1+3N)]$, where $N=B_2=7$ (16*[5*7+5*22] = 2320 cycles)

Hadamard transform the temporary means and shift out 10 zig-zag scan coefficients: R * 808 cycles.

(see hadamard transform)

Update the codeword i in parallel (provided that counter(O_i)!= 0) by:

a) adding counter(O_i) to counter(R'_i),

(1+3N), where N = B_3+1; (25 cycles)

b) and quantizing O_i to one of the 16 level of the sum (O_i + R'_i)

If (counter(O) >= sum(O+R')/2) then Q[0]=1, counter(O)-=sum(O+R')/2; else Q[0]=0;

(B_2-1) + (1+3N) + 1, where N=B_3; (7+ 22 = 29 cycles)

If (counter(O) >= sum(O+R')/4) then Q[1]=1, counter(O)-=sum(O+R')/2; else Q[1]=0;

(B_2-2) + (1+3N) + 1, where N=B_3; (6 + 22 = 28 cycles)

If (counter(O) >= sum(O+R')/8) then Q[2]=1, counter(O)-=sum(O+R')/2; else Q[2]=0;

(B_2-3) + (1+3N) + 1, where N=B_3; (5 + 22 = 27 cycles)

If (counter(O) >= sum(O+R')/16) then Q[3]=1, counter(O)-=sum(O+R')/2; else Q[3]=0;

(B_2-4) + (1+3N) + 1, where N=B_3; (4 + 22 = 26 cycles)

The total number of cycles = (6480-150)*22 + 150*(614) + 2320 + 90*808 + 135 = 306535 cycles or 47*6480 cycles.

Total time for step 2: 306535 cycles *35 ns = 10.7ms

step3: index-based motion estimation

load 8-bit labels to the memory:

6480*8 = 51,840cycles

motion vector determination:

6480*4*(1+3N) = 648,000 cycles, where N is the number of label bits for exact match.

macro-block matching:

(72/4)*(360/4)*(4*4) = 31,680 cycles

at the end of each row, the last 2 columns are nulled (in parallel):

72rows * 8bits = 576cycles

at the end of each frame, the last 2 rows are nulled:

2*8 = 16cycles

Total cycles = 737112 cycles or 114*6480

Total time for step 3: 737112 cycles * 35ns =25.8 ms

Total number of cycles for each input vector during steps 1, 2, and 3 = 1069+47+114= 1230 cycles.

APPENDIX B

Design of the Data Transposer

B.1 Introduction

There have been literatures concerning the designs of the data transposer. The data transposer is often used in 2-D Fourier Transform, matrix transposing, and bit-to-word transformation. In this project, the 8 x 8 data transposer (DT) is used to transpose bits stored in the Computational-RAM (C*RAM) into words to be processed by the host computer (HOST). This implementation can be used for other memory structures where the words stored in the memory and those stored in the host computer are orthogonal. This project is served as a pilot project for building a 32 x 32 compact corner-turning cache.

B.2 Overview of the C*RAM

C*RAM is a fast and efficient memory design which integrates the processing elements (PE) into each memory column to achieve on-chip computation. The C*RAM architecture (Figure B.1) was derived from the largest single-port SRAM module designed by Silburt *et al.*[103].

The memory core is a 256x256 array of 6-transistor CMOS cells. Theoretically, each cell is addressed by a 8-bit row decoder and a 8-bit column decoder. The compact design signifies the 10-bit row decoder and 6-bit column decoder, instead. Data in each column are, however, accessed in a bit-serial manner and 8 consecutive columns are accessed at a time, only 3-bit (BLK<0:2>) column decoder are needed. In the most recent C*RAM, a bidirectional data bus was implemented.

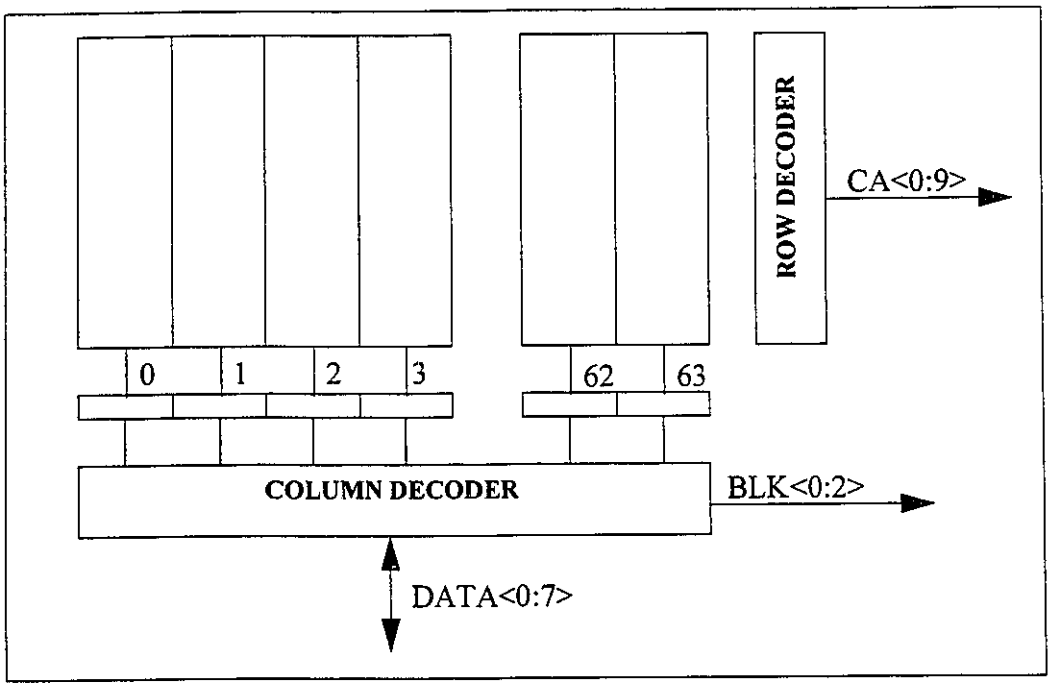

FIGURE B.1 C*RAM architecture

B.3 Data Transposer (DT)

B.3.1 Principle of Operation

As mentioned before, the data transposer, suggested by its name, is used to transpose information in memory hierarchies where physical storage locations of the contents are orthogonal. In this particular application, the words stored in the C*RAM and those stored in the HOST computer are orthogonal. Words in the C*RAM are addressed in blocks of 8 words. The word size can be 8, 16, or 32 bits long. As a pilot project, words are assumed of size 8. Each 8x8 block of memory cells is

unwrapped as shown in figure 2. Eight clock cycles are required to transpose each block. First, a block is addressed by its row and column addresses. Within each block, elements are arranged from the LSB (top) to MSB (bottom). In the first cycle, the set {A7, B7, ... , H7} are read and got written to the corresponding positions in the DT. In the next cycle, the set {A6, B6, ... , H6} are read and transposed, and so on... In the eighth cycle, the set {A0, B0, ..., H0} are finally transposed. Additional blocks may be then transposed until the DT is filled.

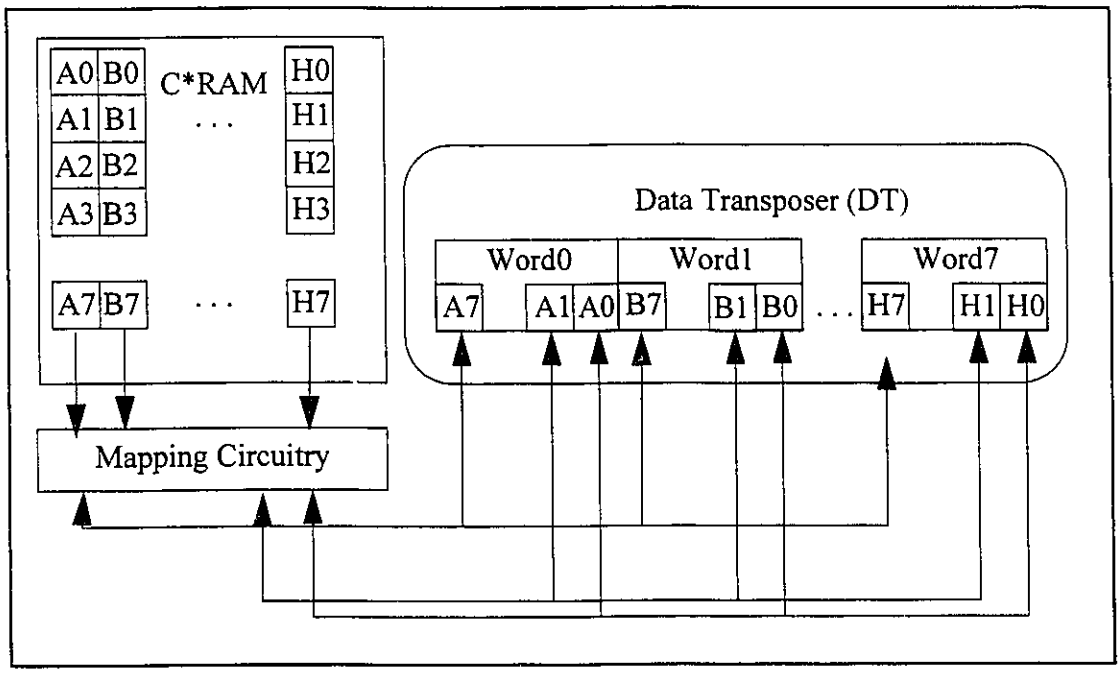

FIGURE B.2 Principle of Data Transposer Operation

On the other hand, when data are written from HOST to DT, the consecutive words are written one beside another until a row is filled up. As shown in Figure B.2, word0 is written first, then word1, and so on up to word 7. Then word8 is written on the position above word0, word9 above word1, and so on... The process is repeated until the whole DT is filled up.

B.4 System Overview

The core of the DT is derived from the same SRAM module as the C*RAM. The 0.8um BiCMOS SRAM design was based on a synchronous self-timed architecture which achieves a fast 5ns nom-

inal access and cycle time. In this implementation, the memory core is considered as a black box with the given inputs and output and their respective positions at the interface.

Each cell of the 8x64 SRAM core is, for compaction purpose, addressed by 3-bit row decoder and 6-bit column decoder. The 6-bit column decoder is used to address 64 memory columns which formed blocks of 8 words, each having 8 bits. Since 8-bit words are accessed by C*RAM in parallel - with each word being accessed in a bit-serial fashion - only 3 bits are required for addressing. These 3 bits are directly derived from the 3 least significant bits of the C*RAM row address (CA<0:2>). Similarly, the 8-bit words are serially accessed by the HOST - one word at a time - thus, 3 bits are needed. Since, the HOST memory is assumed to have 2^N words, the 3 least significant bits of the HOST (HA<0:2>) can be retained for column access. In the row direction, the 3 C*RAM column address bits (BLK<0:2>) can be used when accessed by C*RAM; and the next 3 least significant bits (HA<3:5>) can be used when accessed by HOST. These row-mapping circuitries can be built on-chip when the word size of HOST are known.

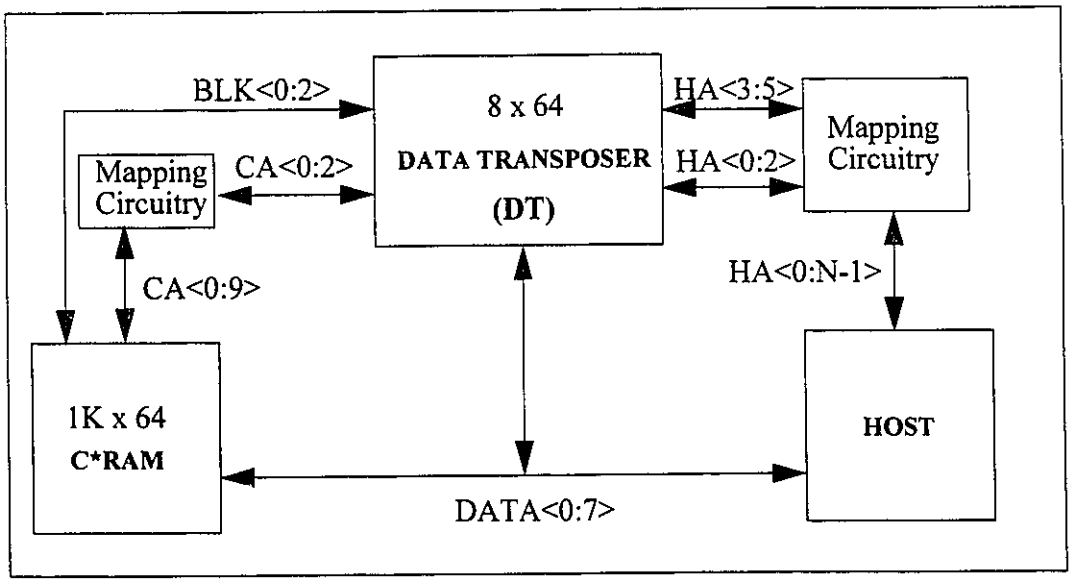

FIGURE B.3 System Overview

The C/Hbar control line selects the path between C*RAM and the DT, and between HOST and the DT. The R/Wbar selects the read and write operations, respectively. The overall picture of the

C*RAM, DT, and HOST is shown in Figure B.3. Interface to each memory column of the memory core includes 3 accesses: write enable (WE), input (IP), output (OP). The WE enables the memory write by a logic high, and disables by a logic low. IP and OP are linked to the 8-bit bidirectional bus via tristate buffers which are controlled by the R/Wbar and the predecoded addresses.

B.5 Design Considerations

In order to take full advantage of the compact and modular memory design, the memory core and sense-amplifiers were taken from BNR's SRAM single-port module. The design objectives are then to build a set of column decoders that effectively map data to and from C*RAM/HOST in a predefined manner and to layout the DT using BiCMOS technology.

- Single-port versus dual-port: Single-port is simple and ideal for pilot project. Dual-port transposer speedups the process but possesses problems with data inconsistency when the actual corner-turning cache is designed.
- Bidirectional bus versus unidirectional bus: To make DT compatible with the current C*RAM, bidirectional data bus was chosen.
- Modes of operation: Since this is a single-port DT (and a future corner-turning cache), there should exist a set of operations which enables C*RAM and HOST to access the DT in a mutually exclusive manner that is only one operation is enabled at a time. (This is somewhat restricted because a read from C*RAM and a read from HOST should not impose any problem. But this can only be done only with a dual-port DT). The 4 modes of operation are summarized in Table B.1 below.

TABLE B.1 Modes of Operations.

R/Wbar	C/Hbar	Operation
0	0	Write from HOST to DT
0	1	Write from C*RAM to DT
1	0	Read from DT to HOST
1	1	Read from DT to C*RAM

- Data written from DT to HOST are in blocks of consecutive words, each occupies 8 consecutive columns in the DT. In the read operation, the reverse path is used.
- Data written from DT to C*RAM are also in blocks of 8 bits occupying every 8th memory column. In the read operation, the reverse path is used.
- HOST computer should have 8-bit words for compatibility.

B.6 Circuit Implementations

The overall design involves four basic cells (Figure B.4):

- The complement_cell takes in the R_Wbar and the C_Hbar signals and produces the write control signal and the HOST address.
- The dec_cell predecodes the CA column addresses along with C_Hbar to ensure proper setup time for read and write operations from/to DT.
- The dec_cell2 predecodes the HA column addresses along with C_Hbar to ensure proper setup time for read and write operations from/to DT.
- The access_cell accommodates circuitries such as access-enable and buffers for data inputs and outputs. This cell is repeated 64 times across the memory interface.

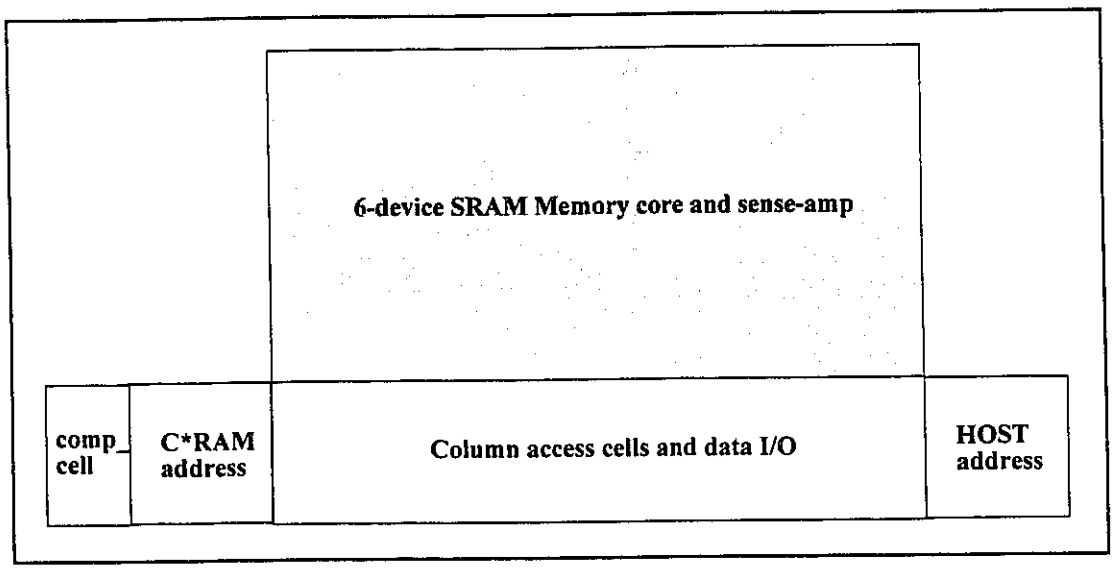

FIGURE B.4 Symbolic layout diagram of the DT

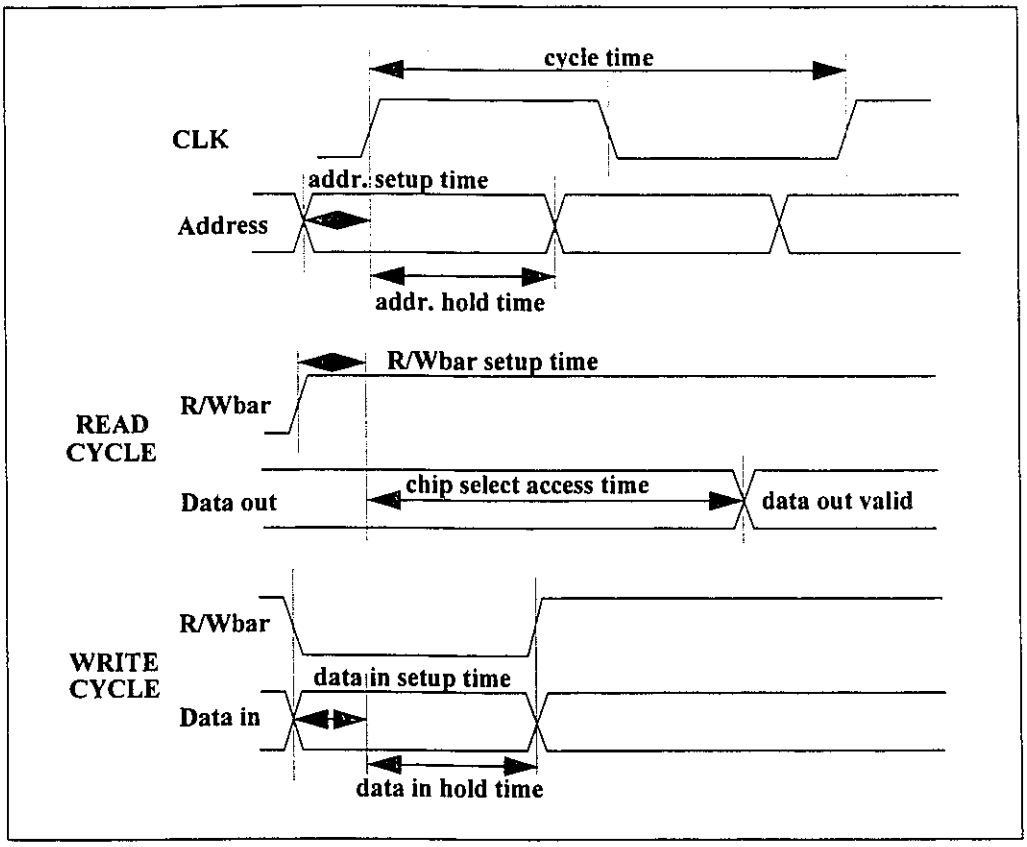

FIGURE B.5 Data Transposer I/O timing diagram

All the files are in ~thinh/vlsi585 directory. The layout rep.'s are complement_cell, dec_cell, dec_cell2, access_cell and DT_interface. The simulation files are in dec10_demo.

- Clock period is 20ns, setup time and hold time are 4ns and 0ns, respectively.
- The size of the DT_interface is 2800umx225um.

B.7 Conclusions

The design of the 8x8 data transposer has been described. The single-port DT is ideal for building a corner-turning cache. Implementation of the DT has been demonstrated with 20ns access and cycle times in a 0.8um BiCMOS process. The circuit could be running faster if transistor sizing had been done and the memory core had been given.

APPENDIX C

Bit Rate Analysis

In this appendix, we study the effects on the bit rate equations (6.1-6.5) in the following two implementations:

- Method A: is the AVQ+ME method.
- Method B: In cases where any PD of the best match codeword is greater than a threshold Δ, the closest codeword is replaced by the incoming input vector.

Recall in chapter 6 that the bit rate of AVQ+ME is expressed as follows:

$$BR_{AVQ+ME} = BR_1 + BR_2 \tag{C.1}$$

$$BR_1 = BR_{codebook} + BR_{mismatch} + BR_{exact-match} + BR_{updated-block} \tag{C.2}$$

$$BR_1 = \frac{NB_c}{N_p} + \frac{I(\log_2 N + F_i)}{N_f} + \frac{EF_e}{N_f} + \frac{C(\log_2 N + F_c)}{N_f} \tag{C.3}$$

$$BR_2 = BR_{macroblock-match} + BR_{neighbor-match} + BR_{tempo-mean} \tag{C.4}$$

$$BR_2 = \frac{B(M_b + F_b)}{N_b \times N_f} + \frac{M(M_m + F_m)}{N_f} + \frac{R(\log_2 N + F_r + B_c + Q)}{N_f} \quad \text{(C.5)}$$

where N = the number of codewords;

B_c = the average number of bits per codeword;

N_p = the number of pixels in the input source;

I, F_i = the number of mismatch blocks that required new labels, and the number of bits for its flag, respectively;

E, F_e = the number of exact match blocks and the number of bits for its flag, respectively;

C, F_c = the number of blocks that requires updated codewords, and the number of bits for its flag, respectively;

N_f = the number of pixels per frame;

R, F_r, Q = the number of temporary means transmitted, the number of bits for its flag, and the number of bit to quantized the weight factor, respectively. Note that R<C. In cases where the codeword is replaced by the incoming input vector, R=C. C can then be eliminated.

B, F_b, M_b = the number of macro blocks detected (with zero or non-zero motion vector), the number of bits for its flag, and the number of bits for its motion vector, respectively;

N_b = the size of the search area (4x4);

M, F_m, M_m= the number of blocks detected with non-zero motion vector, the number of bits for its flag, and the number of bits for its motion vector, respectively;

It can be seen that: I+E+C+16B+M = (X*Y) / L = N_f / L.

Now, let

$$BR_{AVQ+ME} = BR_0 + \frac{C(F_c + \log_2 N)}{N_f} + \frac{R(F_r + \log_2 N + B_c + Q)}{N_f} \quad \text{(C.6)}$$

where

$$BR_0 = \frac{NB_c}{N_p} + \frac{I(\log_2 N + F_i)}{N_f} + \frac{EF_e}{N_f} + \frac{B(M_b + F_b)}{N_b \times N_f} + \frac{M(M_m + F_m)}{N_f} \quad \text{(C.7)}$$

The bit rate in method B can be expressed as follows:

$$BR_B = BR_0 + \frac{C(F_c + \log_2 N + B_c)}{N_f} \tag{C.8}$$

The AVQ+ME becomes less efficient than method B when:

$$BR_{AVQ+ME} \geq BR_B \tag{C.9}$$

$$\frac{C(F_c + \log_2 N)}{N_f} + \frac{R(F_r + \log_2 N + B_c + Q)}{N_f} \geq \frac{C(F_c + \log_2 N + B_c)}{N_f} \tag{C.10}$$

$$R(F_r + \log_2 N + B_c + Q) \geq C \times B_c \tag{C.11}$$

$$\frac{R}{C} \geq \frac{B_c}{F_r + \log_2 N + B_c + Q} \tag{C.12}$$

For $B_c = 128$; $F_r = 5$; $N = 256$, and $Q = 4$, we have,

$$\frac{R}{C} \geq 0.88 \tag{C.13}$$

We note that in all simulations, R is always less than 0.62C. Therefore, AVQ+ME is more superior than method B.

Bibliography

[1] A. K. Jain, "Image Data Compression: A Review", *Proceedings of the IEEE*, Vol. 69, No. 3, pp.349-389, March 1981.

[2] N.M. Nasrabadi and R. A. King, "Image coding using vector quantization: A review," *IEEE Trans. Commun.*, vol. COM-36, no. 8, pp. 957-971, Aug. 1988.

[3] R. M. Gray, "Source Coding Theory", *Kluwer Academic Publishers*, 1990.

[4] L. D. Davisson, "Rate-Distortion Theory and Application", *Proceedings of the IEEE*, Vol. 60, No. 7, pp.800-808, July 1972.

[5] R. M. Gray, "Vector Quantization", *IEEE ASSP Mag.*, pp. 4-29, Apr. 1984.

[6] Y. Linde, A. Buzo, and R. M. Gray, "An Algorithm for Vector Quantizer Design", *IEEE Trans. Commun*, Vol. COM-28, pp. 84-95, Jan. 1980.

[7] K. Aizawa, H. Harashima, and T. Saito, "Model-base synthetic image coding - Construction of a 3-D model of a person's face", *Picture Coding Symposium '87*, Stockholm, pp.3-11, 1987.

[8] D. Chen and A. C. Bovik, "Visual Pattern Image Coding", *IEEE Trans. on Communications*, Vol. 38, No. 12, pp.1662-1671, December 1990.

[9] Z. Xie and T. G. Stockham, "Previsualized Image Vector Quantization with Optimized Pre and Postprocessors", *IEEE Trans. on Communications*, Vol. 39, No. 11, pp .1662-1671, November 1991.

[10] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", *Proc. IRE*, vol. 40, Sept. 1952.

[11] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression", *Commun. of the ACM*, vol.30, pp.520-540, June 1987.

[12] N. S. Jayant and P. Noll, *Digital Coding of Waveforms - Principles and Applications to Speech and Video*, pp.465-482, Prentice-Hall, 1984.

[13] A. Habibi, "Survey of Adaptive Image Coding Techniques", *IEEE Trans. on Communication*, Vol. COM-25, No. 11, pp.1275-1284, November 1977.

[14] W. K. Pratt, *Digital Image Processing*, Academic Press, 1990.

[15] K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, and Applications*, Academic Press, second edition 1990.

[16] P. A. Wintz, "Transform Picture Coding", *Proceedings of the IEEE*, Vol. 60, No. 7, pp.809-820, July 1972.

[17] C. L. Yeh, "Color Image Sequence Compression Using Adaptive Binary-Tree Vector Quantization with Codebook Replenishment", *Proc. International Conf. on Acoustic, Speech, and Signal Processing*, pp. 1059-1062, Dallas, Texas, April 1987.

[18] A. Gersho and B. Ramamurthi, "Image Coding Using Vector Quantization", *IEEE 1982 International Conference on Acoustics, Speech, and Signal Processing*, pp.428-431, May 1982.

[19] D. S. Kim and S. U. Lee, "Image Vector Quantization Based on Classification in the DCT domain", *IEEE Trans. on Communications*, Vol. 39, No. 4, pp.549-556, April 1991.

[20] Y. Huh, J. J. Hwang, C. K. Choi, R. L. de Queroiz, and K. R. Rao, "Classified Wavelet Transform Coding of Images Using Vector Quantization", *SPIE Visual Communication and Image Processing '95*, Vol. 1, pp.207-217, Chicago, Illinois, 1994.

[21] I. Dinstein, K. Rose, and A. Heiman, "Variable Block Size Transform Image Coder", *IEEE Trans. on Communications*, Vol. 38, No. 11, pp.2073-2078, November 1990.

[22] S. Gupta and A. Gersho, " Image Vector Quantization with Block Adaptive Scalar Prediction", *SPIE Visual Communications and Image Processing: Visual Communications*, Vol. 1605, pp. 179-189, 1991.

[23] C. W. Rutledge, "Variable Block DPCM: Vector Predictive of Color Images", *Proceedings of the IEEE*, Vol. 1, pp. 130-135, June 1987.

[24] N. M. Nasrabadi and Y. Feng, "A Dynamic Finite State Vector Quantization Scheme", IEEE, pp. 2261-2264, 1990.

[25] T. Murakami, K. Asai, and E. Yamazaki, "Image Sequence Coding", *Electronics Letters*, Vol. 18, No. 23, pp.1005-1006, November 1982.

[26] A. Gersho, "On the Structure of Vector Quantizer", *IEEE Trans. Inform. Theory*, Vol. IT-28, pp.157-166, March 1982.

[27] J. Tou and R. C. Gonzalez, *Pattern Recognition Principles*, Reading, MA: Addition-Wesley, 1974.

[28] A. Buzo, A. H. Gray, R. M. Gray, and J. D. Markel, "Speech Coding Based upon Vector Quantization", *IEEE Trans. on Acoustic, Speech, and Signal Processing*, Vol. ASSP-28, pp.562-574, Octorber 1980.

[29] J. R. MacCalla and M. V. Chang, " Multispectral Data Compression Using Hybrid Cluster Coding", *Proc. AIAA Commun. Satellite Syst. Conf.*, pp.404-407, 1980.

[30] J. K. Flanagan, R. D. Morrell, R. L. Frost, C. J. Read, and B. E. Nelson, "Vector Quantization Codebook Generation Using Simulated Annealing", *Proceedings ICASSP'89*, pp.1759-1762, 1989.

[31] J. Vaisey and A. Gersho, "Simulated Annealing and Codebook Design", *Proceedings ICASSP'88*, pp.1176-1179, April 1988.

[32] Y. He, Q. Zhang, Y. Ye, and Z. Li, "Vector Quantization by Neural Network", *SPIE Medical Imaging IV: Image Capture and Display*, Vol. 1232, pp.359-362, 1990.

[33] J. Li and C. N. Manikopoulos, " Multi-Stage Vector Quantization Based on the Feature Maps" *SPIE Visual Communications and Image Processing IV*, Vol. 1199, pp.1046-1055, November 1989.

[34] W. H. Equitz, "A New Vector Quantization Clustering Algorithm", IEEE Trans. on Acoustics, Speech, and Signal Processing, Vol. 37, No. 10, pp.1568-1575, October 1989.

[35] C.K Chan and C. K. Ma, "A Fast Image Codevector Generation Method", *IEEE VSPC '92*, pp.68-73, 1992.

[36] B. Ramamurthi and A. Gersho, "Classified Vector Quantization of Images", *IEEE Trans. on Communications*, Vol. COM-34, No. 11, pp. 1105-1115, November 1986.

[37] D. S. Kim and S. U. Lee, "Classified Vector Quantizer Based on Minimum Distance Partitioning", *SPIE Visual Communications and Image Processing: Visual Communications*, Vol. 1605, pp. 190-201, 1991.

[38] J. W. Kim and S. U. Lee, "A Tranform Domain Classified Vector Quantization for Image Coding", *IEEE Trans. on Circuits and Systems*, Vol. 2, No. 1, pp549-556, March 1992.

[39] J. Y. Nam and K. R. Rao, "Image Coding Using a Classified DCT/VQ Based on Two-Channel Conjugate Vector Quantization", *IEEE Trans. on Circuits and Systems for Video Technology*, Vol. 1, No. 4, pp.325-336, December 1991.

[40] S. Quackenbush, "Hardware Implementation of a 16Kbps Subband Coder Using Vector Quantization", *IEEE* , pp386-389, 1988.

[41] P. H. Westerink, D. E. Boekee, J. Biemond, and J. W. Woods, "Subband Coding of Images Using Vector Quantization", *IEEE Trans. on. Comm.*, Vol. COM-36, pp.713-719, 1988.

[42] T. Kim, "Side Match and Overlap Match Vector Quantizer for Images", *IEEE Trans. on Image Processing*, Vol. 1, No. 2, pp. 170-185, April 1992.

[43] R. L. Baker and R. M. Gray, "Differential Vector Quantization of Achromatic Imagery", *Proc. Int. Picture Coding Symp.*, Mar. 1983.

[44] R. L. Baker and R. M. Gray, "Image Compression using Non-adaptive Spatial Vector Quantization", Proc. Conf. Rec. Sixteenth Asilomar Conf. Circuits, Syst., Compu., pp.55-61, October 1982.

[45] B. H. Juang and A. H. Gray, " Multiple Stage Vector Quantization for Speech Coding", *IEEE 1982 International Conference on Acoustics, Speech, and Signal Processing*, pp.597-600, April 1985.

[46] L. Wang and M. Goldberg, "Block Transform Image Coding by Using Multi-stage Vector Quantization with Optimal Bit Allocation", *IEEE Trans. on Communications*, Vol. 39, No. 9, pp. 1360-1369, September 1991.

[47] F. Kossentini, M. J. Smith, and C. F. Barnes, "Image Coding with Variable Rate RVQ", *IEEE 1992*, Vol. 40, No. 11, ppIII-369- III-372, 1992.

[48] W. Y. Chan, S. Gupta, and A. Gersho, "Enhanced Multi-stage Vector Quantization by Joint Codebook Design", *IEEE Transactions on Communications*, Vol. 40, No. 11, pp. 1693-1697, November 1992.

[49] S. A. Rizvi and N. M. Nasrabadi, "Multi-stage Vector Quantizer Design Using Competive Neural Networks", *Proc. Visual Communications and Image Processing*, Vol. 2308, pp.150-164, September 1994.

[50] N. M. Nasrabadi and Y. Feng, "Image Compression Using Address-Vector Quantization", *IEEE Trans. on Communications*, Vol. 38, No. 12, pp. 2166-2173, December 1990.

[51] N. M. Nasrabadi and Y. Feng, "A Multilayer Address-Vector Quantization Technique", *IEEE Trans. on Circuits and Systems*, Vol. 37, No. 7, pp. 912-921, July 1990.

[52] N. M. Nasrabadi, S. E. Lin, and Y. Feng, "Interframe Hierarchical Vector Quantization", *IEEE 1989 International Conference on Acoustics, Speech, and Signal Processing*, May 1990.

[53] A. Buzo, A. H. Gray, R. M. Gray, and D. J. Markel, "Speech Coding Based upon Vector Quantization", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-28, pp. 562-574, October 1980.

[54] W. H. Equitz, "Interframe Hierarchical Vector Quantization", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 37, No. 10, pp.1568-1575, October, 1989.

[55] V. Ramasubramanian and K. Paliwal, "Fast K-Dimensional Tree Algorithms for the Nearest Neighbor Search with Applications to Vector Quantization", *IEEE Transactions on Signal Processing*, Vol. 40, No. 3, pp. 518-531, March 1992.

[56] E. A. Riskin and R. M. Gray, "A Greedy Tree Growing Algorithm for the Design of Variable Rate Vector Quantizers", IEEE Transactions on Signal Processing, Vol. 39, No. 11, pp.2500-2507, November 1991.

[57] C. Braccini, A. Grattarola, F. Lavagetto, and S. Zappatore, "VQ Coding for Videophone Applications Adopting Knowledge-Based Techniques: Implementation on Parallel Architectures", *European Transactions on Telecommunications and Related Technologies*, Vol.3, No.2, pp 45-52, Mar-Apr 1992.

[58] W. Li and E. Salari, "Hybrid VQ of video sequences using Quadtree motion segmentation", *Proceedings of Visual Communications and Image Processing '94*, Vol. 2308, pp.1383-1390, September 1994.

[59] F. Idris and S. Panchanathan, "Image Sequence Coding Using Frame Adaptive VQ", Visual Communications and Image Processing '93, vol. 2094, pp.941-952, November 1993.

[60] M. Goldberg, P. R. Boucher, and S. Shlien, "Image compression using adaptive vector quantization," *IEEE Trans. Commun.*, vol. COM-34, no. 2, pp. 180-187, Feb. 1986.

[61] K. Zeger and A. Bist, "Universial Adaptive Vector Quantization with Application to Image Compression", IEEE 1992, Vol. COM-34, No. 8, pp.III-381 - III-384, 1992.

[62] Gersho and M. Yano, "Adaptive Vector Quantization by Progressive Codevector Replacement", *IEEE 1982 International Conference on Acoustics, Speech, and Signal Processing*, pp. 133-136, March 1985.

[63] C. L. Yeh, "Image Compression Using Non-Adaptive Vector Quantization with Adaptive Overhead Transmission for Codevector Replacement", *Proc. GLOBECOM '87*, pp.35.7.5-35.7.5, 1987.

[64] S. Panchanathan and M. Goldberg, "Adaptive Algorithms for Image Coding Using Vector Quantization", *Signal Processing: Image Communication*, Elesvier Science Publishers, pp. 81-92, 1991.

[65] S. Panchanathan and M. Goldberg, "A Mini-Max Algorithm for Image Adaptive Vector Quantization", *IEE Proceedings: Part I - Communication, Speech and Vision*, Vol. 138, No. 1, pp.53-60, February, 1991.

[66] A. N. Netravali and J. D. Robbins, "Motion Compensated Television Coding: Part I", *Bell Syst. Tech. Journal*, Vol. 58, pp. 631-670, March 1979.

[67] J. R. Jain and A. K. Jain, "Displacement Measurement and Its Application in Interframe Image Coding", *IEEE Trans. on Communications*, Vol. COM-29, no. 12, pp.1799-1808, December 1981.

[68] C. H. Chou and Y. C. Chen, "A VLSI Architecture for Real-Time and Flexible Image Template Matching", *IEEE Trans. on Circuits and Systems*, Vol. 36, No.10, pp.1336-1342, October 1989.

[69] L. D. Vos and M. Stegherr, "Parameterizable VLSI Architecture for the Full-Search Block Matching Algorithm", *IEEE Trans. on Circuits and Systems*, Vol. 36, no.10, pp.1309-1316, October 1989.

[70] A. N. Netravali and J. O. Limb, "Transform Picture Coding", *Proceedings of the IEEE*, Vol. 68, no. 3, pp.366-407, March 1980.

[71] J. A. Roese, W. K. Pratt, and G. S. Robinson, "Interframe Cosine Transform Image Coding", *IEEE Trans. on Communications*, Vol. COM-25, No. 11, pp.1329-1338, November 1977.

[72] Q.Guo, N. M. Nasrabadi, and N. Mohesenian, "An Interframe Dynamic FSVQ Codec for Video Sequence Coding", *IEEE 1992 International Conference on Acoustics, Speech, and Signal Processing*, pp. III-501 - III-504, March 1992.

[73] L. Corte-real and A. P. Alves, "Vector Quantization of Image Sequences Using Variable Size and Variable Shape Blocks", *Electronics Letters*, vol. 26, no. 18, pp. 1483-1484, August 1990.

[74] N. M. Nasrabadi, C. Y. Choo, and J. U. Roy, "Interframe Hierarchical Address - Vector Quantization", *IEEE Journal on Selected Areas on Communications*, Vol. 10, No. 5, pp.960-967, June 1992.

[75] W. Li and E. Salari, "Hybrid VQ of video sequences using Quadtree motion segmentation", Proceedings of Visual Communications and Image Processing '94, Vol. 2308, pp.1383-1390, September 1994.

[76] M. Goldberg and H. Sun, "Image Sequence Coding Using Vector Quantization", *IEEE Trans. Commun.*, vol. COM-34, pp. 703-710, July 1986.

[77] M. Goldberg and H. Sun, "Frame Adaptive Vector Quantization for Image Sequence Coding", *IEEE Trans. on Commun.*, vol. 36, no. 5, pp. 629-635, May 1986.

[78] P. Monet and C. Labit, "Codebook Replenishment in Classified Pruned Tree-Structured Vector Quantization of Image Sequences", *IEEE 1990 International Conference on Acoustics, Speech, and Signal Processing '90*, vol. 4, pp. 2285-2288, Albuquerque, 1990.

[79] T. M. Le, S. Panchanathan, and M. Snelgrove, "C*RAM Implementation of VQ for Image Compression", *Proceedings of IEEE Workshop on Visual Signal Processing and Communications '94*, pp.157-162, September 1994.

[80] T. M. Le and S. Panchanathan, "C*RAM Implementation of an Adaptive Vector Quantization for Video Compression", *IEEE Transactions on Consummer Electronics* '95, Vol. 41, No. 3, pp.738-747, August 1995.

[81] ISO, "JPEG Digital Compression and Coding of Continous Tone Still Images", Draft ISO 10918, 1991.

[82] G. K. Wallace, "The JPEG Still Picture Compression Standard", *Communication of the ACM*, Vol. 34, No. 4, pp30-44, April 1991.

[83] ISO, "Coding of Moving Pictures and Associated Audio", Committee Draft of Standard ISO 11172, ISO/MPEG 90/176, December 1990.

[84] D. L. Gall, "MPEG: a video compression standard for multimedia applications", *Communications of the ACM*, Vol. 34, pp.46-58, April 1991.

[85] CCITT, "Video Codec for Audiovisual Services at p x 64kbits/s", CCITT Recommendation H.261, International Telegraph and Telephone Consultative Committee (CCITT), CDM XV-R 37-E, August 1990.

[86] R. Steinmetz, "Data Compression in Multimedia Computing - Standards and Systems", *ACM Multimedia Systems*, Springer-Verlag, pp.187-204, 1994.

[87] S. L. Tanimoto, "Architecture Issues for Intermediate-Level Vision", *Intermediate-Level Image Processing*, M.J. Duff, Academic Press Inc., London 1986.

[88] A. M. Wallace et al., "Dynamic Control and Prototyping of Parallel Algorithms for Intermediate and High-Level Vision", *Computer*, Vol. 25, No.2, pp.43-53, February 1992.

[89] M.J. Flynn, "Very High-Speed Computing Systems", *Proc. IEEE*, Vol.54, pp.1901-1909, 1966.

[90] B. P. M. Tao, H. Abut, and R. M. Gray, "Hardware realization of waveform vector quantization," *IEEE J. Select. Areas Commun.*,Vol. SAC-2, pp.343-352, Mar. 198

[91] S. Panchanathan and M. Goldberg, "A Content-Addressable Memory Architecture for Image Coding Using Vector Quantization", *IEEE Trans. on Signal Processing*, Vol. 39, No. 9, Sept. 1991.

[92] S. Panchanathan and M. Goldberg, "A Systolic Array Architecture for Image Coding Using Adaptive Vector Quantization", *IEEE Trans. on Circuits and Systems for Video Technology*, Vol.1, No.2, June 1991.

[93] Abut, B. P. M. Tao, and J. L. Smith, "Vector quantizer architectures for speech and image coding", *Proc. IEEE ICASSP'87*, Vol. 2 (Dallas, TX) pp. 756-759, Apr. 1987.

[94] G. A. Davidson and A. Gersho, "Application of a VLSI vector quantization processor to real-time speech coding," *IEEE Select. Areas Commun.*, Vol. SAC-4, no. 1, pp.112-124, Jan. 1986.

[95] G. A. Davidson, T. Stanhope, R. Aravind, and A. Gersho, "Real Time Speech Compression with a VLSI Vector Quantization Processor", *Proceedings of the IEEE Conference on Acoustic, Speech, and Signal Processing*, Vol.4, pp.1437-1440, March 1988.

[96] G. A. Davidson, P. R. Cappello, and A. Gersho, "Systolic Architectures for Vector Quantization", *IEEE Transactions on Acoustic, Speech, and Signal Processing*, Vol. 36, No. 10, pp.499-512, Oct. 1988.

[97] M. Yan and J. V. McCanny, "A Bit-level Systolic Architecture for Implementing a VQ Tree Search", *Journal of VLSI Signal Processing*, Vol.2, pp.149-158, 1990.

[98] H. Sun and H. Hsu, "A VLSI wavefront array for image vector quantization," *Proc. 30th Midwest Symp. Circuits Syst.*, pp. 1230-1233, August 1987.

[99] D. G. Elliott, Private Presentation on C*RAM, 1993.

[100] C. Cojocaru, M. Snelgrove, and D. G. Elliott, "A BATMOS Processing Element for 256Kb Computational RAM", internal report, Department of Electronics, Carleton University.

[101] D. G. Elliott, M. Snelgrove, and M. Stumm, "Computational-RAM: A Memory-SIMD Hybrid and Its Application to DSP", *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, Boston, May 1992.

[102] W. M. Loucks, M. Snelgrove, and S.G. Zaky, "VASTOR: A Microprocessor-Based Associative Vector Processor for Small Scale Applications", *Proceedings of the 1980 International Conference on Parallel Processing.*

[103] Allan L. Silburt et al, "A 200MHz 0.8um BiCMOS modular memory family of DRAM and multi-port SRAM's", *Proceedings of the IEEE 1992*, CICC May 1992.

[104] C. Cojocaro, "Computational RAM: Implementation and Bit-Parallel Architecture", Master's dissertation, Department of Electronics, Carleton University, , December 1994.

[105] C.Gonzales and E. Viscito, "Flexibly Scalable Digital Video Coding", *Signal Processing: Image Communication*, vol. 5, pp.5-20, Elsevier Publishers, 1993.

[106] S. Panchanathan, A. Jain, and N. Gamaz, "Scalable Image Compression Using Combined Wavelet Transform and Vector Quantization", *SPIE*, Vol. 2419, pp.354-364, 1994.

[107] L. Wang, and M. Goldberg, "Lossless Progressive Image Transmission by Residual Vector Quantization", *Proceedings of the Thirteenth Biennial Symposium on Communications*, pp. c1.9-c.1.12, Kingston, Ont., June 1986.

[108] M. I. Szean, M. Rabbani, and P. W. Jones, "Progressive Transmission of Images using a Prediction / Residual Encoding Approach", *Opt. Eng.*, vol. 28, no. 5, pp.556-564, 1989.

[109] A. Castonguay and C. Cojocaru, "C*RAM Micro Assembly Language, v0.7", 1994, Technical report, CUDOE-HSL-94-03.

[110] Y. Chu, "A Destructive Read-Out Associative Memory", *IEEE Transactions on Electron. Comput.*, Vol. 14, pp. 600-605, 1965.

[111] K. Hwang, F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, USA.

[112] M.J. Flynn, "Some computer organizations and their effectiveness", *IEEE Trans. Computer*, Vol. C-21, pp.948 - 960, 1972.

[113] B. Prince, *Semiconductor Design - A Handbook of Design, Manufacture, and Application*, Second edtion, John Wiley & Sons Ltd., 1991.

[114] N. Wang, *Digital MOS Integrated Circuits - Design for Applications*, Prentice Hall, Englewood Cliffs, New Jersey.

[115] T. Dillinger, *VLSI Engineering*, Prentice-Hall, 1988.