Université d'Ottawa · University of Ottawa

Visual Communications on a Memory-Embedded Array Processor: The Computational*RAM

by

Thinh Minh Le, B.A.Sc, M.A.Sc

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of
the requirement for the degree of
Doctor of Philosophy
in
Computer Engineering

Ottawa-Carleton Institute of Electrical Engineering
School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, Canada

To my family

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 scholarly 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 scholarly research.

Thinh M. Le

The University of Ottawa requires the signatures of all persons using or photocopying this thesis. Please sign and date below with your email and postal addresses.

Abstract

In this thesis, image and video processing algorithms, especially the compression algorithms, are first studied in their natural formats to appreciate the needs for real-time operations and hence, parallel computing. The computational intense, memory-bound problems are next approached from two directions: algorithmic and architectural. Algorithmic approach tends to systematically analyze the flow independence and data independence of a program, while architectural approach tends to gain speed-up by resource multiplicity and time sharing.

The majority of image and video processing algorithms are inherently data-parallel in nature. The vectorization of these algorithms requires consistent practices, and new challenge in parallel programming seems endless. The data-parallel nature of image/video processing algorithms map well onto the Single-Instruction stream, Multiple-Data stream (SIMD) of an increasingly popular Memory-Embedded Array Processor classified as the Intelligent RAMs, specifically, the Computational*RAM (C*RAM). C*RAM is a SIMD-memory hybrid where the processing elements are pitch-matched to memory columns of a conventional computer RAM at the sense-amplifiers to take advantage of the inherently high memory bandwidth, and the emulation of the massively parallel processors.

Throughout the thesis, speed-ups from 1 to 3 orders of magnitude are obtained. Memory-bound algorithms such as Motion Estimation, and Mean-Absolute-Error for Nearest Neighbor Distortion Computation are among the most efficient implementations.

At its best, this thesis will, definitely, put forward the promising research direction which involves fast and efficient in-memory parallel computing for visual communications.

Acknowledgements

First of all, I would like to thank Professor Sethuraman Panchanathan, formerly with the University of Ottawa, at the Arizona State University for his continuous supervision and encouragement. I am also grateful of Professor Ralph Mason and Professor Martin Snelgrove at Carleton University for their advice, guidance, and support. I also thank my colleague, Peter Nyasulu, for his cooperation throughout my research.

This work was made possible by the earlier supports from Microelectronics Network (MicroNet) under the Network Centers of Excellence (NCE) program of the Government of Canada, the School of Graduate Studies and Research and the School of Information Technology and Engineering (SITE) at the University of Ottawa.

My appreciation are also to Professor Tet Yeap for his valuable comments which have helped in the progress of this thesis; and the staffs: Ms. Lucette Lepage of SITE, and Mr. Nagui Mikhail of the Department of Electronics, Carleton University.

Finally, I would like to thank my parents, my family, and my loving fiancee for their continuous encouragement and understanding. It would not have been possible to accomplish this innovative and rewarding work without their love and supports.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ALU: Arithmetic Logic Unit

AIS: Applied Intelligent System

ASCII: American National Standard Code for Information Exchange

ASIC: Application Specific Integrated Circuit

AVO: Audio-Video Object

BiCMOS: Bipolar Complementary Metal-Oxide Semiconductor

BMA: Block Matching Algorithm

BPM: Bit Plane Matching

BP-BPM: Band-Pass Bit Plane Matching

bpp: bit per pixel

BSP: Burroughs Scientific Processor

CAM: Content-Addressable Memory

CAPP: Content Addressable Parallel Processors

CBR: Constant Bit Rate

CCIR: The International Radio Consultative Committee of the International Telecommunications Union (now ITU-R)

CCITT: Consultative Committee on International Telephony and Telegraphy

CG: Carry Generator

CIF: Common Intermediate Format

CIS: Canadian Ice Services

CRT: Cathode Ray Tube

CM: Connection Machine

CMOS: Complementary Metal-Oxide Semiconductor

COP: Control Op-code

CPU: Central Processing Unit

CR: Codebook Replenishment

C*RAM: Computational Random Access Memory

CU: Computing Unit, sometimes refers to a PE in the 1 x n^2 configuration, and a group of 8 PEs in the n x n configuration

CW: Codeword

DAP: Distributed-memory Array Processor

DCT: Discrete Cosine Transform

DFP: Differential Frame Prediction

DMA: Direct Memory Access

DPCM: Differential Pulse Code Modulation

DRAM: Dynamic Random Access Memory

DSP: Digital Signal Processing / Digital Signal Processor

FCC: Federal Committee on Communications

FEXOR: Feature Extraction and eXclusive OR.

FFT: Fast Fourier Transform

FIR: Finite Impulse Response

FPGA: Field Programmable Gate Array

fps: frame per second

FSVQ: Full-search VQ

GIPS: Giga Instructions per Second

GOP: Group Of Pictures

HDPP: High Density Parallel Processors

HDTV-GA: High Definition TeleVision - Grand Alliance

HVS: Human Visual System

HW: hardware

IBM: International Business Machine Corp.

IDCT: Inverse Discrete Cosine Transform

IEC: International Electrotechnical Commission

IMAP: Integrated Memory Array Processor

I/O: Input - Output

IRAM: Intelligent RAM

ISDN: Integrated Services Digital Network

ISO: International Standards Organization

ITS: Ice Tracking System

ITU-T: International Telecommunications Union - Telecommunication Standards Sector

JEDEC: Joint Electron Devices Engineering Council

JPEG: Joint Photographic Experts Group

KBps: Kilo Bytes per second

Kbps: Kilo bits per second

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

LR: Label Replenishment

LSB: Least Significant Bit

MAE: Mean Absolute Error

Mb: Mega-bits

MB: Mega-bytes

Mbps: Mega-bits per second

MCVQ: Multiple Codebook Vector Quantization

MD: Maximum Descent

ME/C: Motion Estimation/ Compensation

MEG: Morphological External Gradient

MIG: Morphological Internal Gradient

MIP: Morphological Image Processing

MIMD: Multiple-Instruction stream, Multiple-Data stream

MISD: Multiple-Instruction stream, Single-Data stream

MOS: Mean Opinion Score, also Metal-Oxide Semiconductor

MOPS: Million Operations per second

MPEG: Moving Picture Experts Group

MPP: Massively Parallel Processor

MSB: Most Significant Bit

MSE: Mean Square Error

MU: Memory Unit

NMSE: Normalized Mean Square Error

NTSC: National Television System Committee

PC: IBM-compatible Personal Computer

PCB: Printed-Circuit Board

PD: Pixel Decimation, also Pixel Distortion

PDC: Pixel Difference Classification

PE: Processing Element

PIP-RAM: Parallel Image Processing Random Access Memory

PIT: Progressive Image Transmission

PNN: Pairwise Nearest Neighbor

PPM: Portable Pixel Map

PSNR: Peak Signal-to-Noise Ratio

PSB: Programmable Segment Bus-Tie

PWB: Programmable Word Boundary

QCIF: Quarter CIF

RAM: Random Access Memory

RISC: Reduced Instruction-Set Computer

RLC: Run-Length Coding

ROM: Read-only Memory

SA: Search Area

SCVQ: Sub-Codebook Vector Quantization

SIF: Source Input Format

SIMD: Single-Instruction stream, Multiple-Data stream

SIMM: Single Inline Memory Module

SISD: Single-Instruction stream, Single-Data stream

SNR: Signal-to-Noise Ratio

SRAM: Static RAM

SRC-PIM: Supercomputing Research Center Processor-in-Memory

SW: software

TSVQ: Tree Search VQ

TTOP: Truth Table Op-code

VBR: Variable Bit Rate

VHDL: VHSIC Hardware Description Language

VHSIC: Very High Speed Integrated Circuit

VLC: Variable Length Code

VLSI: Very Large Scale Integrated Circuit

VRAM: Video RAM

VQ: Vector Quantization

CHAPTER 1

Introduction

For more than a decade, visual communications has been an important area of research. Digital services such as: tele-conferencing, tele-medicine, digital library, distant education, and video-on-demand, etc., to name a few, are being made possible with continuing developments in the area of visual computing and communications.

During the last Mars exploration, 3-D images taken by the Sojourner Rover have once again excited the scientific community and general public. The visual information was processed and losslessly compressed before being sent back to Earth, which was, at the time, 310 million miles away from Mars. Recently, with the advances in tele-medicine, more than 100,000 X-ray images of injured soldiers and civilians have been sent from Bosnia, where Serbian Croatian conflict occurs, to radiologists in Germany for diagnosis and suggestions for treatments [Hardin99]. Another example of useful image and video processing is the Ice Tracking System (ITS) used at Canadian Ice Services (CIS), Environmental Canada [Lee97]. This system is to generate sea-ice

kinematic information over navigable Canadian waterways. ITS must efficiently extract the ice motion between sequential satellite images with minimum errors, and operate in different ice regions without human intervention.

In order to provide reliable visual services, vast quantity of visual data must be processed and compressed in a very short period of time, while maintaining an acceptable fidelity. The raw data stream entails a significantly large bit rate, for example, an uncompressed color video sequence sampled at a frame rate of 30 frames per second (fps), where each frame is typically of size 720 x 576 pixels, requires the transmission of nearly 300 Mbits per second (Mbps). For terrestrial broadcast in North America, the emerging High Definition Television - Grand Alliance (HDTV-GA) has adopted a 6MHz broadcasting channel [Petajan95]. With the presence of significant noise levels and National Television System Committee - TV (NTSC) interference, the Federal Committee on Communications (FCC) service requirements limit the bit rate to about 20 Mbps. Hence, a compression ratio of 15:1 is required, not including error detection/correction and signalling/controlling bit streams. These processing and compression requirements pose a difficult problem to both algorithm developers and computer architects.

During the last few years, the international standardization committees have been developing many visual compression standards. The Joint Photographic Experts Group (JPEG) of the International Standards Organization (ISO) has specified a compression algorithm for still photographic images [ISO/IEC 10918-1]. The International Telegraph and Telephone Consultative Committee (CCITT) has proposed the H.261, and subsequently, the low bit-rate H.263 standards for video telephony and teleconferencing [ITU-T Rec. H.261 & H.263]. The Moving Picture Experts Group (MPEG) of ISO has also proposed the MPEG-1 and MPEG-2 standards for storage and transmission of video sequences [ISO/IEC 11171 & 13818-2]. A new

standard for content based manipulation and coding of video, MPEG-4, has been scheduled to be finalized in early 2000 [Haskell98, Koenen99].

Visual communications involve acquisition, storage, understanding, processing, transmission, and representation of image and video information. Image and video processing, a part of visual communications, involves: representation and modeling, enhancement, restoration, analysis, reconstruction, and compression. Without loss of generality, in many existing techniques, visual data processing algorithms and compression standards possess a high degree of parallelism.

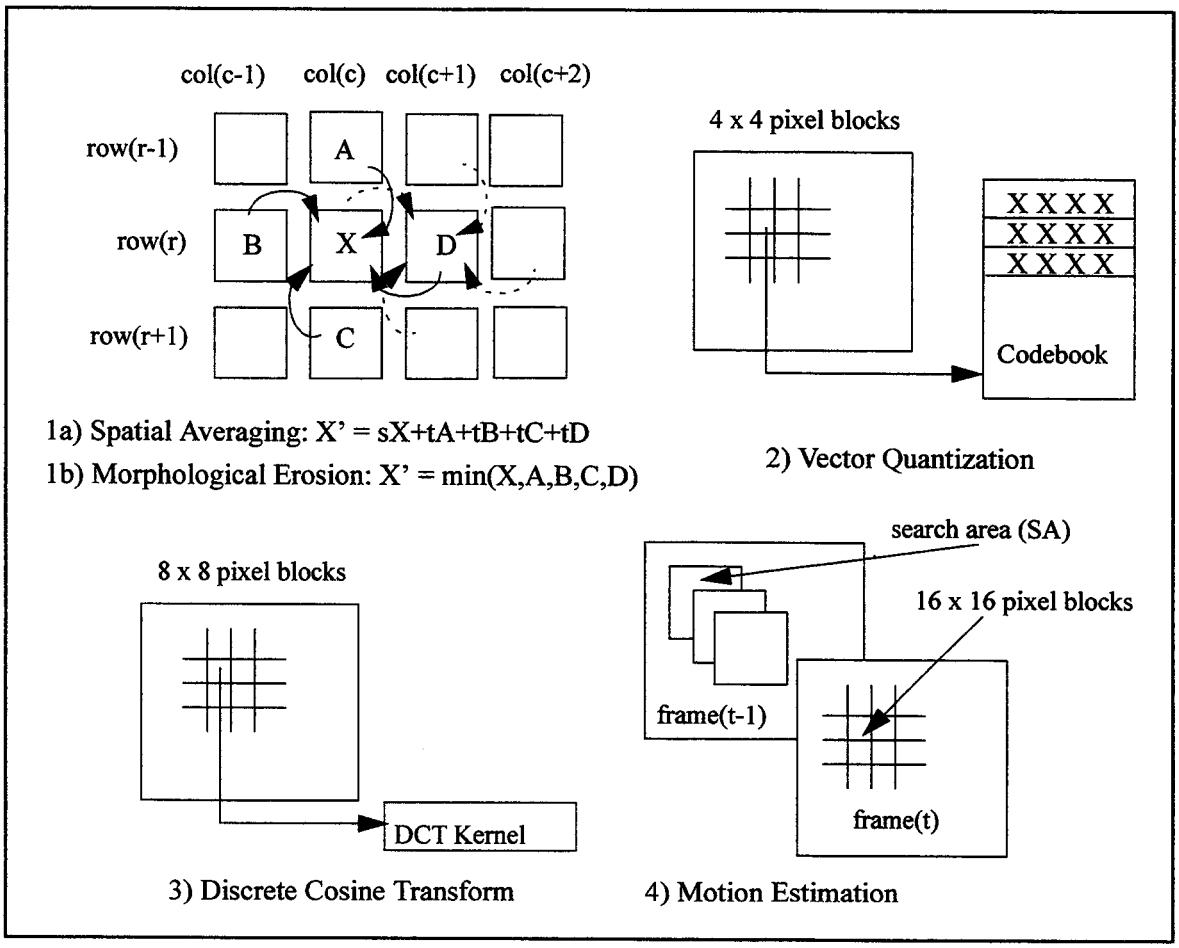


Figure 1.1 Some examples of parallel image and video processing

In particular, the processing of a certain pixel/block of data does not generally require data from distant pixels/blocks, and most of the instructions for processing these pixels/blocks are identical.

Examples of parallel image and video processing operations, such as pixel-based and block-based operations, are shown in Fig. 1.1.

The first example involves image filterings using a linear filter (1a), and a non-linear filter(1b). An image can be low-pass filtered by spatial averaging. The final result is the blurring effect on the original image. In spatial averaging, the new value X' at X is a linear combination of X and the surrounding 4-connected[1] neighbours A, B, C, and D. It is noted that D' can also be computed using the same set of instructions. The same image can also be operated upon by a non-linear filter, a major branch of which is Morphological Image Processing. In the morphological erosion example, X' is, by definition, the minimum of (X, A, B, C, and D) depending on the window defined. Similarly, D' can be similarly derived using the same relationship. Vector Quantization (VQ), in example 2, is generally referred to as the process of representing k-dimensional vectors by their indices in a pre-generated codebook (or a set of representative vectors). In VQ, each image is partitioned into non-overlapping 4 x 4 pixel blocks. Each block, called an input vector, is compared against a set of codevectors in the codebook, and the index of the best match codevector will be used as the address to a lookup-table for reconstruction. In example 3, the Discrete Cosine Transform (DCT), commonly used in spatial image compression, is performed by applying the same set of instructions to every 8 x 8 pixel block of an image. Finally, in Motion Estimation (ME), shown by example 4, the same search procedure is applied to each of the non-overlapping 16 x 16 pixel macro-blocks. Therefore, parallelism can also be applied at the macro-block level. Details of the image and video processing algorithms will be presented in Chapter 2. In light of the above examples, one can conclude that the image and video processing algorithms map naturally onto Single-Instruction stream, Multiple-Data stream (SIMD) architecture [Flynn72].

1. The 8-connected neighbours include the 4-connected neighbours: A, B, C, and D, and the 4 diagonally adjacent neighbours.

The mapping of image and video processing algorithms onto a parallel architecture requires a thorough understanding of the underlying hardware. It has been reported that every year the processor speed of a computer system increases 60% and the memory speed increases by a modest 7% [Patterson96], while the memory capacity is quadrupled every three years [Hennessy94]. The growing processor-memory bandwidth gap cannot be ignored as it results in a loss of performance due to long wait-state and inefficient task scheduling. While the increase in memory capacity enables the designers to place more memory cells on a single chip, it does not bridge the aforementioned bandwidth gap or allow efficient utilization of the increasing speed of the processor. On the other hand, interleaved/pipelined memory modules and hierarchical memories [Patterson98], such as multi-level caches, can enhance the processor-memory interface. For instance, the Rambus architecture [Farmwald92], which supports multiple very high-speed ports, can be used to increase the processor's efficiency. However, these correction measures do not fully take advantage of the parallelism in the visual data processing algorithms. Another important consideration is that the word lengths needed for image and video processing are 8, 16, or 24 bit long, which are much shorter than the common 32- and 64-bit words of today's microprocessor[1].

Recently, several Intelligent Random Access Memory's (IRAM) have appeared in the literature [Patterson96]. One of the IRAM's, namely the Computational*RAM (C*RAM) [Elliott92, Cojocaru93, Cojocaru94, Foss96, McKenzie97, and Torrance98], is a special memory where modular logic circuits are tightly coupled to the compact memory arrays at the sense-amplifiers of the existing RAM. The C*RAM's concept enables the large *on-chip* memory bandwidth to be fully utilized, while emulating array processors for massively parallel operations. On-chip computing will, in theory, result in either a lower operating power, or a higher operating frequency. This is because the total capacitance at the I/O pads and bus are reduced. Let us take a

1. As an example, a Matlab constant requires 8 bytes of storage.

look at the following expression for the dominant dynamic power dissipation P_D of Very Large Scaled Integrated (VLSI) Complementary Metal-Oxide Semiconductor (CMOS) circuit:

$$P_D = C_L V_{DD}^2 f_p \tag{1.1}$$

Eq. 1.1 indicates that, for a given voltage V_{DD} and an operating frequency f_p, the lower the load capacitance C_L, the lower the power dissipated P_D. Also, by rearranging this equation, given a voltage and a power level, it can be seen that the lower the load capacitance, the higher the potential operating frequency.

For instance, if switching energy is 3μW/MHz for a 0.3pF bitline at V_{DD}=3.3 V and 300μW/MHz for a 30pF bus, a typical 16Mb DRAM with 4K bitlines cycling at 20MHz requires 240mW, while its 16 buses running at 100MHz takes 480mW [Elliott97]. This analysis will not be further discussed since they are best claimed by the circuit designers. The C*RAM's concept addresses the above-mentioned image and video processing problems in that:

- Memory bandwidth is fully utilized which is beneficial to memory-bound operations such as motion estimation;
- The emulation of arrays of SIMD processors allows natural mappings of image and video processing/compression algorithms onto the C*RAM;
- Variable word lengths are readily achievable through bit-serial operations. Furthermore, arithmetic and logic operations can be optimized at the bit level;
- Applications can be programmed at the controller level which alleviates the needs for specialized hardware;

With parallel image and video processing algorithms and the SIMD-based on-chip array processors in mind, it is apparent that the effective mappings of such algorithms onto the SIMD-based C*RAM's parallel architecture is an important task.

1.1 Thesis outline

According to [Patterson98], given an instruction set, the CPU performance can be increased by the following three factors:

- Increase in clock speed;
- Improvement of processor organization; and
- Enhancement of its compiler.

The clock speed is technology dependent and, therefore, is dealt with by the device physicists and silicon engineers. The processor, in this case, the C*RAM, is designed and fabricated by the C*RAM design team. Issues such as operating power, operating frequency, and data bandwidth have been extensively discussed [Elliott92, Cojocaru93, Cojocaru94, Foss96, McKenzie97, and Torrance98]. The compiler enhancement issue should be most comprehended by application engineers and best addressed by software engineers. The processor performance can only be optimized when there are strong interactions between software developers and hardware designers. Therefore, there is a need for understanding the underlying architecture and the intended applications. Having mentioned the difficult issues in image and video processing such as intense computation and high data bandwidth, a solution to the given challenging problems will be addressed in this thesis entitled Visual Communications on a Memory-Embedded Array Processor: The Computational*RAM.

In the following chapters, reviews on image and video processing algorithms and C*RAM-related SIMD designs are provided along with the simulation results *to prove that such mappings are, indeed, efficient, economical and yet practical.* In Chapter 2, visual data (image and video) processing algorithms and compression standards are reviewed. In Chapter 3, the SIMD-based logic-in-memory designs are examined. Chapter 4 provides the proposed architecture improvements to the existing C*RAM's along with the underlying C*RAM assembler, necessary

for C*RAM implementations and performance analysis. In Chapter 5, implementations of image processing algorithms will be presented. Such implementations include general and morphological image processing algorithms, DCT and RLC, and Sub-Codebook VQ for compressing image using large codebook. Chapter 6 presents a newly proposed low complexity ME algorithm for video compression and its C*RAM implementation. Finally, Chapter 7 provides the overall implementations of image and video compression standards, followed by the conclusions and future work in Chapter 8.

Appendix A provides other image compression issues which are referred, but not directly related to the thesis contents. Appendix B provides an example of matrix multiplications using C*RAM. Finally in Appendix C, typical program listings for VHDL-based[1] C*RAM models, and test files which are used to generate stimuli for performance comparisons in different processing and compression algorithms are presented.

At the end of this thesis, references to the compression standards are listed first followed by references to websites, for example, WSh263. References to other publications are subsequently listed in alphabetical order of principal authors' surnames followed by the publication years.

1.2 Thesis contributions

Two VHDL models, a baseline C*RAM and an enhanced C*RAM, have been built. Based on the resulting circuit characteristics, software simulations of C*RAM functionalities have been performed. Macros for image and video processing have been written as well as suggestions for improvements. Contributions to this thesis include:

- Development of an instruction set for C*RAM operations (Sections 4.4 and 4.5). The assembly codes enable better memory and register allocations at the bit level, and variable

1. Very-high-speed-integrated-circuits Hardware Description Language.

word-length signed and unsigned arithmetic operations[1]. Variable word-length results in a speed-up of 48% in distortion and filtering computations (Section 6.2.3);

- Enhancements to C*RAM designs for improving performances of image and video processing algorithms. Such enhancements are the parallel addition/subtraction features (Section 4.1.1) which are used to generate the sum of absolute differences[2] for VQ and ME. This enhancement results in a speed-up of 116% in mean absolute error (MAE) calculations (Section 6.2.3). In addition, the implementation of the one-to-many communication network (Section 4.1.2) enables emulations of Hypercubes, Illiac mesh, and Barrel shifter; and hence, enlarges the range of applications for C*RAM;
- Proposed Sub-codebook VQ (SCVQ) implementation for fast and efficient VQ search (Section 5.5). SCVQ is proved to result in a speed-up of 400% or higher on any 1-D SIMD array processor; and
- Proposed algorithm for fast motion estimation by extracting the frame features and XORing the binary feature maps for motion vectors (FEXOR) (Section 6.1). Motion estimation using FEXOR generally results in a speed-up of 500% compared to full-search block matching algorithm (FBMA) implemented using the same C*RAM configuration, and as high as 5,000% over the FBMA on a RISC (Section 6.2.9).

In addition to the proposed algorithms and architectural enhancements, the existing image and video processing algorithms and compression standards have been mapped onto the C*RAM which are listed below:

- General and morphological image processing techniques (Section 5.2);

1. The sum of two 8-b operands is a 9-b intermediate result, and the sum of two 9-b intermediate results is a 10-b final result. Hence, there is no need for a fixed 10-b addition on the 8-b pixels.
2. Conventional MAE distortion operation requires 3 steps: subtraction, absolute operations, and addition to the partial sum. The new PE computes the same distortion operation in 2 steps: subtraction, and parallel addition to or subtraction from the partial sum.

- 2-D DCT (Section 5.3) and Run-length coding (Section 5.5.2);
- ME algorithms: Full-search Block Matching Algorithm (FBMA), Pixel Decimation (PD), and Band Pass - Bit Plane Matching (BP-BPM) (Section 6.2); and the
- JPEG (Section 7.1), and H.26x and MPEG's compression standards (Section 7.2).

CHAPTER 2

Overview of Image and Video Processing Algorithms and Compression Standards

---

In this chapter, parallel image and video processing algorithms are reviewed, followed by summaries of the compression standards. In order to facilitate understanding of visual information of images, color components are first discussed in Section 2.1. In Section 2.2, general image processing techniques are presented. These techniques are listed from sub-pixel operations, pixel operations, to local operations. Mathematical morphology, a major branch of non-linear image processing, is presented in Section 2.3. Data compression techniques, both lossless and lossy, are discussed in Section 2.4 along with the subjective and objective evaluation methods. In Section 2.5, the spatial-domain compression techniques are presented, emphasizing the standardized compression techniques such as DCT. In Section 2.6, temporal-domain compression techniques are reviewed, and a study of fast Motion Estimation techniques is provided. In Section 2.7, a low bit-rate compression technique for table look-up applications, namely, Vector Quantization, is

reviewed. Finally, in Section 2.8, a review of the JPEG image compression standard, H.261 & H. 263, and MPEG-1, -2, and -4 video compression standards is presented.

2.1 Image Color Components

When an image is processed, unless otherwise stated, its *luminance* component, commonly known as the grayscale, is implied. The luminance of an image is computed using the following expression:

$$Y = 0.299R' + 0.587G' + 0.114B' \tag{2.1}$$

where R', G', and B' are the gamma-corrected values[1] of the red, green, and blue components, respectively, obtained by an image sensing device [Netravali89]. The magnitudes of the three multiplicative factors correspond to the responsiveness to red, green, and blue of the cones in the *human visual system* (HVS). For color image processing, other color-difference components, *chrominance*, are used. The derivation of chrominance components will be discussed in detail in Section 2.8 when image and video compression standards are reviewed.

2.2 General Image Processing Techniques

In this section, *sub-pixel operations* are first presented followed by the *pixel operations* (or point operations). Sub-pixel operations such as bit extraction, level-shifting, and re-quantizing bit-depth to lower resolutions[2] involve the manipulations on the bits representing the pixel . In pixel operations, where no memory is required, a given grayscale level $u \in [0, L]$ is mapped onto another level $v \in [0, L]$ according to some transformation $v = f(u)$. The *local operations*

1. Color components appear as voltage waveforms whose instantaneous values are directly proportional to the illumination falling on the corresponding spots of a camera tube target. Such a video signal is not directly suitable for the CRT display because of its non-linearity. Therefore, the received voltage values are compensated using the expression $I_{display} \propto V^{\gamma}_{received}$ where I is the light intensity, V is the signal voltage, and γ is some power from 2.0 to 3.0.

2. The 12-b medical images can be displayed at 8-b or 4-b without the need for the pixel co-processor.

where a processed pixel is determined by a window of its neighboring pixels are presented. *Global operations* which involve the pixels of the entire image are not vectorizable in the SIMD context, and therefore, not discussed.

Unless otherwise stated, the origin of the coordinates is considered the upper-left corner of an image; and u(m,n) and v(m,n) are the pixels at row m, column n of the input and output images, respectively.

2.2.1 Bit Extraction

Progressive image transmission requires the display of images in various qualities from coarse to fine. In general, the image pixel is uniformly quantized to b bits, and some of the bit planes are more visually significant than others. Let the image pixel be represented by:

$$u = k_1 2^{b-1} + k_2 2^{b-2} + \ldots + k_{b-1} 2 + k_b \tag{2.2}$$

If the n^{th} most-significant bit (MSB) is extracted for display, the output is formed by setting:

$$v = \begin{cases} L & k_n = 1 \\ 0 & otherwise \end{cases} \tag{2.3}$$

On most of the 8-b monochrome images, it is noticed that only the first 6 MSB's are visually significant [Jain89]. In applications where image quality is not important such as image browsing, only the more visually significant bits are stored or transmitted, and hence, compression can be achieved.

Sometimes the pixel values are required to be level-shifted from (0, 255) to (-128, 127) with zero being the center. Sub-pixel operations are applied by flipping the MSB of the raw pixel[1]. Typical mappings are shown in Table 2.1:

1. in unsigned representation.

Table 2.1 Level-shifting on 8-b pixels.

Decimal	Binary	Decimal	Level-shifted Binary
0	0000 0000	-128	1000 0000
1	0000 0001	-127	1000 0001
...	...	...	...
127	01111111	-1	11111111
128	10000000	0	00000000
129	10000001	1	00000001
...	...	...	...
254	1111 1110	+126	0111 1110
255	1111 1111	+127	0111 1111

2.2.2 Contrast Stretching, Clipping, and Thresholding

Due to poor / non-uniform lighting conditions, or due to non-linearity / small dynamic range of the imaging sensor, image contrast may be too low for viewing. To stretch the image contrast to a visible level, transformations can be performed on the image (Fig. 2.1) as defined below:

$$v = \alpha u \qquad 0 \le u \le a \tag{2.4}$$

$$v = \beta(u-a) + v_a \qquad a < u \le b \tag{2.5}$$

$$v = \gamma(u-b) + v_b \qquad b < u \le L \tag{2.6}$$

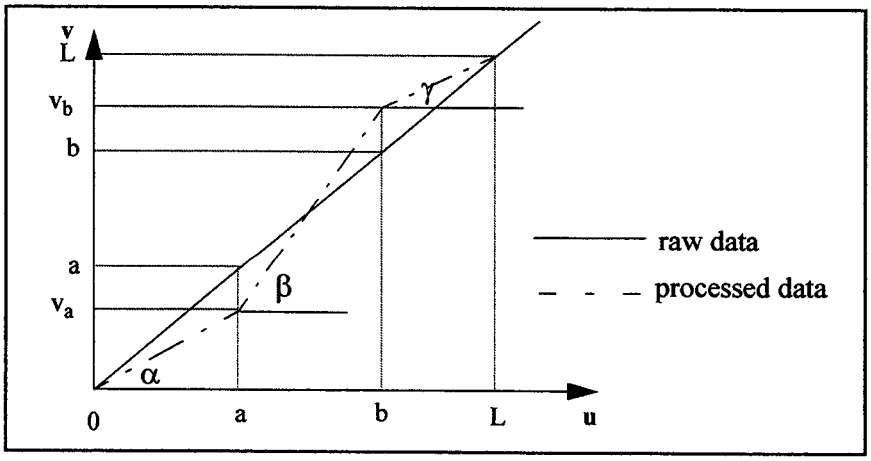


Figure 2.1 Contrast stretching

where the slope β is chosen to be greater than unity in the region of stretch [a,b].

When the input image is known to be in the range [a,b], clipping can be performed by setting $\alpha = \gamma = 0$ and $\beta = \frac{L}{b-a}$. Therefore:

$$v = 0 \qquad 0 \le u \le a \tag{2.7}$$

$$v = \beta(u-a) \qquad a < u \le b \tag{2.8}$$

$$v = L \qquad b < u \le L \tag{2.9}$$

A special case of clipping is thresholding, where a = b = t, the output hence becomes a binary image.

Intensity Level Slicing: Images from weather maps often contain regions of different interests such as clouds, precipitations, cold/warm fronts, hurricane, etc. Intensity level slicing is transformation which allows segmentation of certain grayscale regions from the rest of the image. In cases where background is of concern, regions of interest are set to white, while the rest of the image (background) remains unchanged:

$$v = \begin{cases} L & a < u \le b \\ u & otherwise \end{cases} \tag{2.10}$$

2.2.3 Image Subtraction

In pattern analysis, it is desirable to compare two images to identify, for example, the missing components on a printed circuit board (PCB) on an assembly line, or the blood-flow in radiology. The two images are aligned, and then subtracted. Let $u_1(m,n)$ and $u_2(m,n)$ be the pixels at row m, column n of two input images 1 and 2, respectively. The output image is defined as:

$$v(m,n) = |u_1(m,n) - u_2(m,n)| \tag{2.11}$$

v(m,n) is later enhanced by thresholding.

A special case of image subtraction is *grayscale reversal.* A negative image can be obtained by reversing the grayscale defined by the following transformation:

$$v = L - u \tag{2.12}$$

This method is of particular interest in machine vision since the dissimilar object, resulting from subtracting the image of a standard object from the image of the object in study, can be detected.

In the following section, pixel operations are performed on the neighboring pixels. The output image is obtained by convolving the input image with a *finite impulse response* (FIR) filter called a *spatial mask.*

2.2.4 Convolution

If W is a chosen window, and a(k,l)'s are the filter weights, the output pixel is obtained by replacing the corresponding input pixel with a weighted sum of the pixels in window W.

$$v(m,n) = \sum_k \sum_l a(k,l)u(m-k,n-l) \qquad k,l \in W \tag{2.13}$$

A typical filter, the *Spatial Averaging* filter, often used for noise smoothing, or low-pass filtering. is given by:

$$v(m,n) = \frac{1}{2}u(m,n) + \frac{1}{8}u(m-1,n) + \frac{1}{8}u(m+1,n) + \frac{1}{8}u(m,n-1) + \frac{1}{8}u(m,n+1) \tag{2.14}$$

Similarly, the *unsharp masking* filter, commonly used for edge enhancement, is expressed by:

$$v(m,n) = u(m,n) + \lambda g(m,n) \tag{2.15}$$

where $\lambda > 0$ and g(m,n) is a suitably defined gradient at location (m,n). A commonly used gradient function is the discrete Laplacian:

$$v(m,n) = u(m-1,n) + u(m+1,n) + u(m,n-1) + u(m,n+1) - 4u(m,n) \tag{2.16}$$

It has been mentioned earlier that spatial *low-pass filter* can be obtained by spatial averaging. If $h_{LP}(m,n)$ denotes a FIR low-pass filter, then the FIR *high-pass filter* $h_{HP}(m,n)$ can be obtained by:

$$h_{HP}(m,n) = \delta(m,n) - h_{LP}(m,n) \tag{2.17}$$

A spatial *band-pass filter* $h_{BP}(m,n)$ can be obtained by:

$$h_{BP}(m,n) = h_{L1}(m,n) - h_{L2}(m,n) \tag{2.18}$$

where $h_{L1}(m,n)$ and $h_{L2}(m,n)$ denote the short-term and long-term averaging low-pass filters, respectively.

While low-pass filters are useful for noise smoothing and interpolation, high-pass filters are useful in edge extraction and image sharpening. Band-pass filters are useful in edge enhancement and emphasis of other high-pass image characteristics in the presence of noise.

While general image processing requires arithmetic operations such as: addition, subtraction, and multiplication, the fundamental operations of morphological image processing involve comparisons and maximum/minimum searches. In the following section, morphological image processing will be presented.

2.3 Image Processing using Mathematical Morphology

Mathematical morphology refers to a branch of non-linear image processing and analysis that concentrates on the geometric structures in an image [Doughety92]. Morphological image processing includes enhancement, segmentation, restoration, edge detection, texture analysis, particle analysis, curve filling, and general thinning. These methods have been successfully applied to robot vision, material sciences, industrial inspection, medical imaging, remote sensing, and automatic character recognition.

Mathematical morphology considers images as *algebraic sets*. Consider the two images, sets A and B. The basic set operators on A and B are the union $A \cup B$ and the intersection $A \cap B$. These operators are applied directly to binary images. For a grayscale digital image, the corresponding operators are the *supremum* $A \vee B$ (maximum) and the *infimum* $A \wedge B$ (minimum) operators, respectively. Another basic set operator is the *complement*. The complement of an image f, denoted f_c, is defined as:

$$f_c(x) = L - f(x) \tag{2.19}$$

where L is 1 for binary images, and 255 for 8-b monochrome images.

The basic idea of all morphological operators is to consider the image as a set. The image is then touched with a set of known shapes called *structuring elements*[1] (SE). SE's can be a small disk, a horizontal bar or a vertical bar, to name a few. A SE B can be represented by a matrix, and B^T is the transpose of B. An origin must be specified for each SE. Fig. 2.2 shows some applications where a centered-origin disk SE and a off-centered-origin horizontal bar SE are used.

2.3.1 Erosion and Dilation

Erosion and dilation are the basis of all morphological operations. The *erosion* of a set X by a SE(B), denoted $\varepsilon_B(X)$, is defined as the locus of points x such that B is included in X when its origin is placed at x:

$$\varepsilon_B(X) = \{x | B_x \subset X\} \tag{2.20}$$

1. A structuring element can also be considered as a convolution kernel.

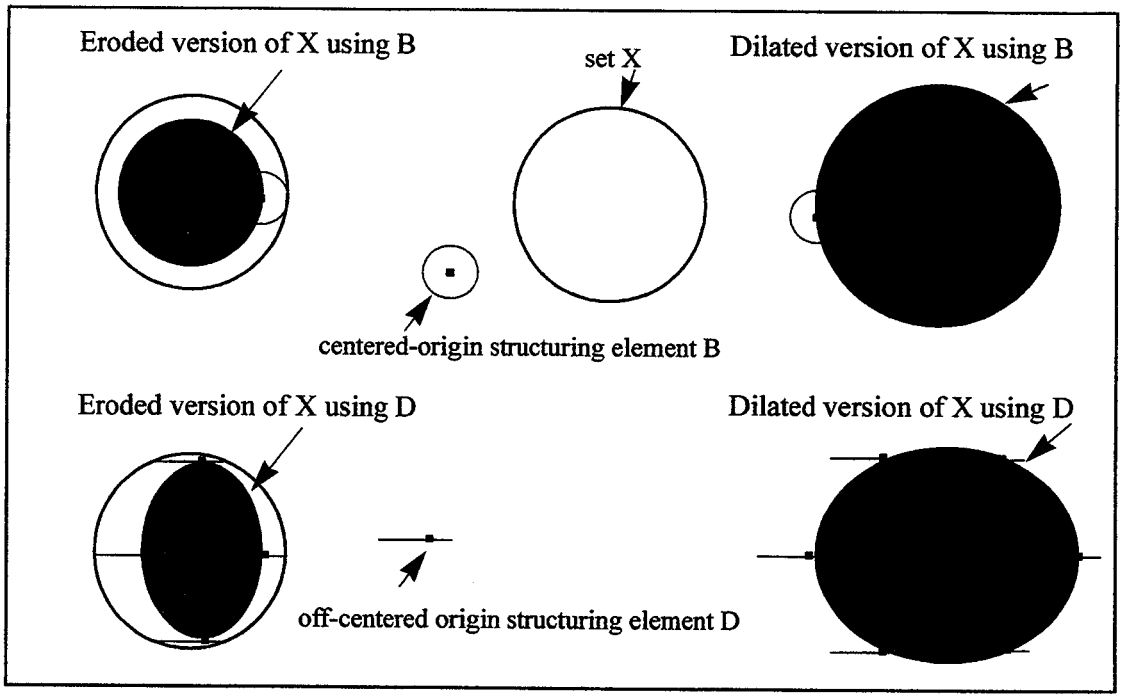


Figure 2.2 Erosions and Dilations with different structuring elements B and D

Dilation is the dual operator of erosion. The *dilation* of a set X by a SE(B), denoted $\delta_B(X)$, is defined as the locus of points x such that B hits X when its origin coincides with x:

$$\delta_B(X) = \{x | B_x \cap X \neq \varnothing\} \tag{2.21}$$

Dilation of a previously eroded set does not allow, in general, the recovery of the initial set. In fact, there exists no inverse transformations for erosion and dilation.

2.3.2 Opening and Closing

The *opening* of an image X by a SE(B), denoted $\gamma_B(X)$, is defined as the erosion of X by B followed by the dilation by B^T (the transpose of B).

$$\gamma_B(X) = \delta_{B^T}[\varepsilon_B(f)] \tag{2.22}$$

The erosion by SE(B) primarily removes objects (or noise) of sizes equal to or less than B. Subsequent dilation restores the shape of the remaining objects to some extent. Sometimes, the remaining objects become larger than their originals.

It is sometimes desirable to preserve the shapes of the remaining objects. Therefore, *opening-by-reconstruction* operation is required and is defined as the erosion of a set using SE(B) followed by a dilation using SE(B^T) with a condition. This condition ensures that the sizes of the remaining objects do not get larger after undergoing dilation. Therefore, if an object is not eliminated, its size and shape are preserved.

The dual operation of opening is closing. The *closing* of an image X by a SE(B), denoted $\phi_B(X)$, is defined as the dilation of X by SE(B) followed by the erosion by SE(B^T). *Closing-by-reconstruction* is similarly defined.

2.3.3 Morphological Gradients

The basic morphological gradient is the gradient of Beucher *g*. It is defined as the arithmetic difference between the dilated and the eroded versions of the original image X with SE(B).

$$g(X) = \delta_B(X) - \varepsilon_B(X) \tag{2.23}$$

g(X) represents the maximum variation of the grayscale intensities within a neighborhood rather than a local slope.

It is crucial to distinguish between the internal and external boundaries of a region. *Morphological internal gradient* (MIG) of an image is the arithmetic difference between the original image X and its eroded image $\varepsilon_B(X)$, while *morphological external gradient* (MEG) of an image is the arithmetic difference between the dilated $\delta_B(X)$ and its original image X.

MIG and MEG operations are shown in Fig. 2.3. Morphological gradient operations remove noise and pixel variations within a solid region while preserving sharp edges. Therefore, morphological smoothing is generally preferred over linear blurring for edge detection [Serra82].

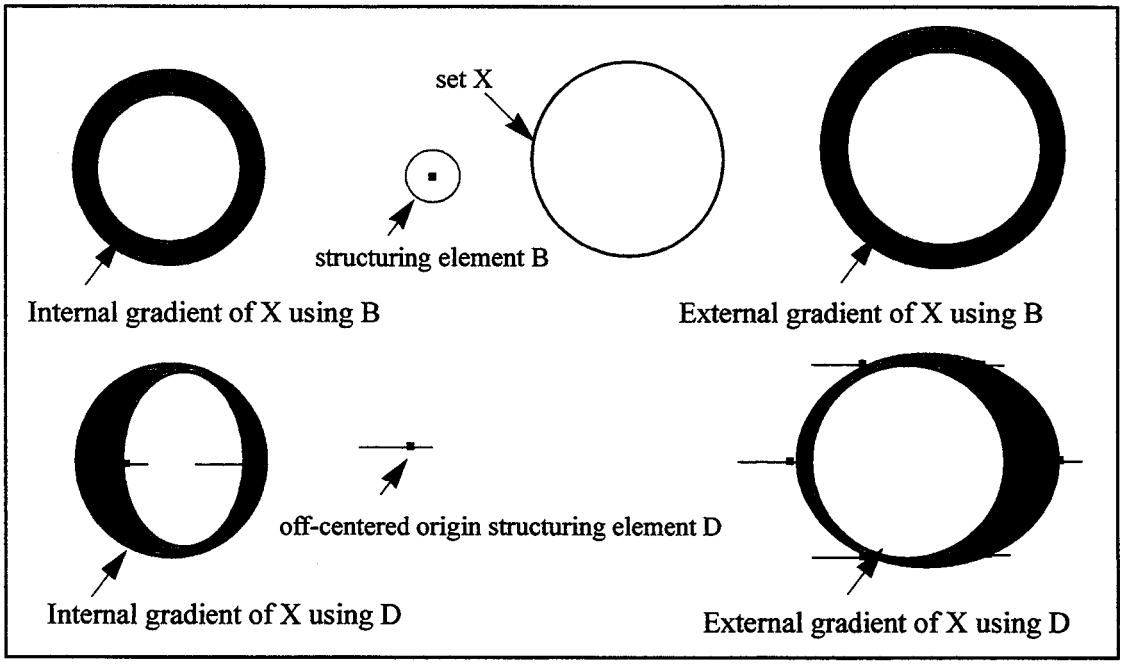


Figure 2.3 Internal and External Gradients using structure elements B and D

2.3.4 Edge Detection

Having introduced various linear and non-linear filters, it is now appropriate to present a few edge detection techniques. Edge detection has been employed for enhancing the features of objects in an image to a machine detectable or humanly visible level where they can be used to make further decisions. In machine vision, edge information can be used to separate nuts and bolts in an assembly line, and to identify whether a product is defective based on its size and shape. As will be introduced in Section 6.1, edge detection techniques are also applied to transform an b-bit image to a 1-bit image for low-complexity motion estimation. There are mainly 3 types of edges: step (as in step function), ramp, and texture edges. Step and texture edges can be detected using first-order filters, while ramp edges can only be effectively detected using multi-scale morphological gradient [Wang96], or second-order filters.

To detect edges, there are four steps: pre-processing, slope approximation, thresholding, and post-processing. Pre-processing involves noise removal and blurring. Slope approximation is to find

gradients in different directions at each pixel. Thresholding is to select the highest gradient at a certain direction of each pixel, and post-processing involves noise removal (if any) and edge connection or merging. Simple edge detection techniques only require steps 1 and 3. A difference image obtained by subtracting the blurred version from its original image is an example. The morphological counterparts are MEG and MIG. More elaborate edge detection techniques require all four steps. First-order techniques are Prewitt, Sobel, Laplacian, and the one proposed by [Canny86] using linear filters, and those proposed by [Lee87, Verbeek88, Hertz92] use non-linear operations. Second-order techniques include [Johnson90], and Laplacian of Gaussian.

For low-complexity motion estimation, the candidate edge detection techniques should be simple and efficient[1], in which all 3 types of edges should ideally be recognized. The connectivity of edges are not critical since unconnected edges are common to both current and reference frames.

In subsequent sections, image and video compression techniques and standards are reviewed. In order to better understand techniques, some data compression concepts and evaluation methods are introduced next.

2.4 Data Compression Techniques

The large memory capacity and high channel bandwidth requirements for storage and transmission of visual data necessitates the use of efficient image and video compression techniques while acceptable fidelity is maintained. Image and video compression, whose goal is to minimize the number of bits required to represent an image and video signal, is essentially a reduction process [Jain81a] in which spatial, temporal, structure, and psycho-visual redundancies are exploited.

1. The low-complexity motion estimation algorithms only remain attractive if their computation requirement is kept low compared to the compute intensive of the full-search block matching algorithm.

In an image and video frame, the pixels in close vincinity are usually highly correlated. For example, pixel values in a local region do not change significantly from one location to the next; and hence, a pixel value can be approximately inferred from the neighboring pixel values. This correlation, called the *spatial redundancy*, can be easily observed and is removed by employing *intraframe* coding techniques such as, predictive coding, vector quantization, transform coding, entropy coding, and their hybrids.

Temporal redundancy, refers to the correlation between the successive frames in a video sequence. The differences between successive frames are due to object motions and/or camera operations[1]. Temporal redundancy is usually detected and removed by applying *interframe* coding techniques such as, predictive coding, adaptive vector quantization, conditional frame replenishment, motion estimation and compensation, and predictive coding with motion estimation.

Structure redundancy results from the fact that an image is the projection of 3-D objects onto a 2-D plane [Aizawa87]. Structure redundancy also refers to the representation of objects based on their basic forms and movements. For instance, human facial expression - including forehead, eye-browns, eyes, cheek, nose, mouth, and chin - can be categorized into many classes depending on the individual mood at a particular time. Each expression can be characterized by a set of controllable parameters which is much less than the amount of image data themselves, therefore, high compression can be achieved.

1. The seven basic camera operations include: fixed (no motion involved), zooming (camera's lens forward and backward while the whole mechanism is fixed on its stand), panning (camera head's movement to the left and to the right), tilting (camera head's movement up and down), booming (camera movement in the horizontal direction), tracking (camera movement in the forward and backward direction), and dollying (camera movement in the vertical direction).

Finally, *psycho-visual redundancy* refers to information which may be removed without sacrificing the subjective image quality due to the properties of the HVS [Chen90, Xie91]. For example, human eyes are more sensitive to small signal changes in dark areas. Other properties include: more sensitivity to noise with patterns compared to random noise[1], less sensitivity to faster moving objects, and less sensitivity to 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 compression minimizes the average number of bits per pixel (bpp) without any loss in objective image quality, while lossy compression is to minimize the bit rate R(D) for a given average distortion D or equivalently, to minimize the average distortion for a given bit rate. More detailed descriptions of lossless and lossy compression schemes are discussed in Section A.1.

In studying the performance of image compression schemes, image quality can be evaluated using *subjective* and *objective* measures. Subjective measures refer to the judgement of a human viewer using *quality rating scales* such as: excellent, good, fair, poor, and bad; or *impairment rating scales* such as imperceptible, perceptible but not annoying, slightly annoying, annoying, and very annoying. The average of the ratings for a given image is called the *mean opinion score* (MOS) and 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 Mean Square Error (MSE), Normalized MSE (NMSE), Mean Absolute Error (MAE), Signal-to-Noise ratio (SNR), and Peak Signal-to-Noise ratio (PSNR). NMSE generally provides a more precise judgement to a particular coding algorithm. On the other hand, many use PSNR as the performance indicator. PSNR is, however, very image dependent. Images with little details tend to

1. Evenly scattered small spots on the windshield can be tolerated compared to the streaks made by the wipers.

have higher PSNR's, compared to those with many details, assuming the same coder is applied. The definitions of the objective measures are reviewed in Section A.2.

2.5 Spatial-Domain Compression Techniques

2.5.1 Predictive Coding

Predictive coding exploits the mutual redundancy among neighbouring pixels. Rather than encoding the pixel intensity, its value is first predicted from the previously encoded pixels. The predicted pixel value $\hat{y}_k$ is then subtracted from the actual pixel value x_k, and the difference e_k (prediction error) is quantized and coded for transmission. The quantized prediction error is used at the receiver to reconstruct the image (Fig. 2.4).

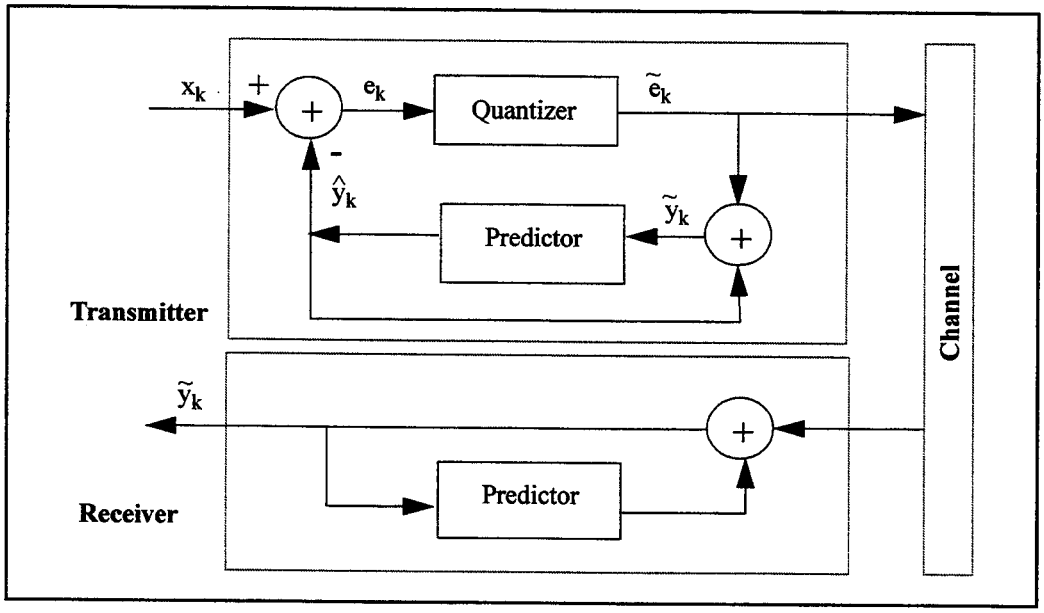


Figure 2.4 Block diagram of a predictive coding system

The coding performance of a predictive coding system can be improved by employing adaptive techniques which track 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. Predictive coding is used in JPEG lossless compression mode (Section 2.8.1).

2.5.2 Run Length Coding (RLC)

In fascimile applications, binary images are to be compressed by some lossless techniques. For instance, the image of a fascimile page is composed of only two grayscales, black (0) and white (1). If this page is scanned line by line, the results are series of 0's and 1's. The total length of these series, or the number of source symbols, can be reduced by replacing a pair: run of 0's and a 1 immediately follows it by a pre-defined bit pattern.

RLC can also be applied to series of zero-valued and nonzero-valued transformed coefficients. The most commonly used transform is the DCT on n x n pixel blocks. The results, after quantization, are a series of n^2 non-zero coefficients and many more zero coefficients. If the resulting block is zig-zag scanned, the runs of zero coefficients become apparent and a run length coder can be easily applied. Assuming that the following partial zig-zag scanned sequence of DCT coefficients:

55, 0, -32, 0, 0, 0, 0, 0, 0, 0, 0, 0, -2, 0, 0, 1

The coder outputs, as a result, are $(0,55)^1$, (1, -32), (9,-2), (2, 1).

2.5.3 Discrete Cosine Transform (DCT)

DCT has been adopted in the JPEG, H.26x, and MPEG-1 and -2 compression standards due to its excellent energy compaction for highly correlated data, and the availabilities of fast algorithms among the orthogonal transforms[2]. Given a data sequence x_i, i = 0, 1, 2, ..., N-1. The 1-D N-point forward DCT X_k is defined by:

1. In practice, the first DCT coefficient, also referred to as DC coefficient, is coded differently.
2. In an orthogonal transform, the inner product of any pair of its basis vectors is zero.

$$X_k = \frac{2c_k}{N}\left(\sum_{i=0}^{N-1} x_i \cos\frac{(2i+1)k\pi}{2N}\right) \tag{2.24}$$

And, the inverse DCT is defined by:

$$x_i = \sum_{k=0}^{N-1} X_k \frac{2c_k}{N} \cos\frac{(2i+1)k\pi}{2N} \tag{2.25}$$

where $c_k = \frac{1}{\sqrt{2}}, k = 0$, and $c_k = 1, otherwise$.

The first DCT algorithms was proposed by Ahmed *et al.* [Ahmed74] where a double-size FFT algorithm was used with complex arithmetic throughout the computation. Chen *et al.* [Chen77] later presented a faster 1-D DCT computation by exploiting the sparseness[1] of the matrices involved. For N=8, the numbers of additions and multiplications are 26 and 16, respectively. Lee [Lee84] further proposed a technique where the computation is sped up by reducing the number multiplications down to 12.

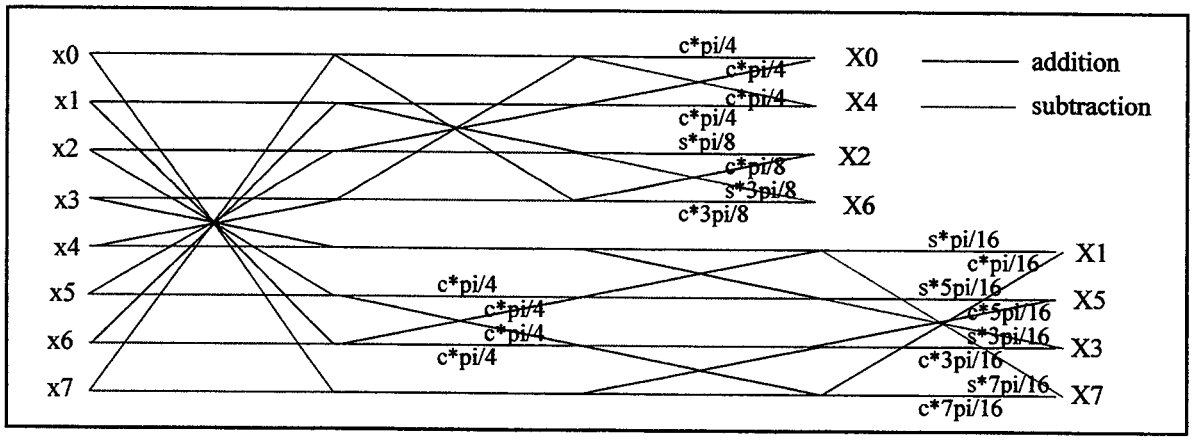


Figure 2.5 Forward DCT flow graph for N=8, Ci = cos i, Si = sin i [Chen77, Rao90]

For image and video compression, 2-D DCT are required. Given a 2-D data sequence $x_{i,j}$, where i,j = 0, 1, 2, ..., N-1. The N x N forward DCT $X_{k,l}$ is defined by:

1. Sparseness implies that most of the elements of the matrix are zeros.

$$X_{k,l} = \frac{2c_k c_l}{N}\left(\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} x_{i,j}\cos\frac{(2i+1)k\pi}{2N}\cos\frac{(2j+1)l\pi}{2N}\right) \tag{2.26}$$

The inverse DCT is defined by:

$$x_{i,j} = \sum_{k=0}^{N-1}\sum_{l=0}^{N-1} X_{k,l}\frac{2c_k c_l}{N}\cos\frac{(2i+1)k\pi}{2N}\cos\frac{(2j+1)l\pi}{2N} \tag{2.27}$$

where $c_k = c_l = \frac{1}{\sqrt{2}}$, *if* k,l = 0 and $c_k = c_l = 1$, otherwise.

By observing the pairs of equations 2.24 and 2.25, and 2.26 and 2.27, it can be seen that the 2-D DCT is separable. Separability enables the transformation to be performed by taking a 1-D DCT on the rows of the input block, then on the columns of the semi-transformed matrix, resulting in the final transformed matrix. This method is sometimes referred to as row-column decomposition. The computational complexity is, therefore, 2N times the numbers of additions and multiplications required for 1-D DCT. For N=8, the number of additions and multiplications are 416 and 256, respectively, using Chen's algorithm; and 464 and 192, respectively, using Lee's. Later, Cho *et al.* [Cho91] proposed a 2-D DCT algorithm which required only 466 additions and 96 multiplications. In this algorithm, the 1-D DCT's are first applied to the groups of odd-numbered samples and even-numbered samples. The results of these operations then undergo some additions and shiftings (divisions by 2) before producing the final results.

In some applications, the DCT coefficients are scaled immediately after the transformation. The scaling factors (i.e. quantization coefficients) may be incorporated onto the matrices if no change to the quantization matrix is anticipated. Feig *et al.* [Feig92] have proposed a 2-D scaled-DCT method resulting in significant savings in the number of additions and multiplications. For N=8: 462 additions and only 54 multiplications are needed. In theory, algorithms with the least number of multiplications are preferable. However, practical implementations emphasize ease of

implementation, i.e regular structure and testability, as well as the small amount of temporary storage required in a particular algorithm.

2.6 Temporal-Domain Compression Techniques

2.6.1 Interframe Transform Coding

By inspecting equations (2.26) and (2.27), the separability property enables us to extend transform coding such as DCT to dimensions larger than two. As an example of a sequence of images, 3-D DCT can be implemented by a series of 1-D DCTs along each of the dimensions [Rao90]. Implementation of the 3-D DCT of a sequence of L frames on the n x n pixel blocks is performed as follows: First, the L-point 1-D transform is executed along the temporal direction, following by the 2-D transform of the frame data.

In video sequences, the statistics along the temporal dimension may vary significantly, therefore, adaptive techniques can substantially improve the coding performance [Habibi77]. For example, the effective number of bits assigned to each coefficient can be made proportional to the coefficient variance [Roese77]. Although transform coding can be extended to multiple dimensions, practical applications appear to be limited [Rao90].

2.6.2 Conditional Replenishment

The conditional replenishment technique [Netravali80] is based on dividing each frame into stationary and non-stationary parts. Only the changed parts are coded. If $x_{i,j,t}$ is a pixel at location (i,j) in the current frame(t), then the predicted value of $x_{i,j,t}$ is the reconstructed value $\hat{x}_{i,j,t-1}$ at the same spatial location in the previous frame(t-1). The prediction error in this case is calculated using:

$$e_{i,j,t} = x_{i,j,t} - \hat{x}_{i,j,t-1} \tag{2.28}$$

If the magnitude of $e_{i,j,t}$ is greater than a pre-specified threshold, then it is quantized, coded, and transmitted along with the address (i,j) of $x_{i,j,t}$.

2.6.3 Motion Estimation

Motion estimation is a widely used technique to exploit the temporal correlation in a video sequence. Motion estimation attempts to obtain the motion information for the various regions/ objects in a scene. Using motion estimation, high compression can be achieved by coding an n x n pixel block in the current frame by a motion vector (with respect to the best match block in a search area - SA- of the previous frame), and followed by the DCT coefficients of the estimated errors. There are two main classes of motion estimation/compensation algorithms: *pel-recursive* [Netravali89] and *block matching* (BMA's) [Jain81b]. 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. The advantage of a pel-recursive algorithm is the ability to overcome problems of multiple moving regions/objects as well as part of a region/object undergoing different displacements. The drawback of pel-recursive algorithms is the overwhelming information involved. Therefore, it is often operated in a predictive manner, that is the motion vectors between frame(t) and frame(t-1) are predicted using the corresponding motion vectors between frame(t-1) and frame(t-2).

Block matching algorithms, on the other hand, assume that all pixels within a block have the same motion and behaves well provided that the following conditions are met:

1. Zooming and rotation of objects are not considered. This is because the objects are assumed to move in translational mode in a plane that is parallel to the camera plane;
2. Pixel illumination between frames is spatially and temporally uniform;
3. Object displacement is constant within a small 2-D block of pixels; and

4. Matching distortion increases monotonically as the displaced candidate block moves away from the direction of the minimum distortion.

The most popular ME technique is the full-search block matching algorithm (FBMA) and is used as a benchmark to other simplified algorithms. FBMA searches all possible displaced locations within an n x n pixel SA to find the best match. Many objective matching criteria, MSE and MAE, to name a few, have been used to find the best match block [Chou89]. The MAE criterion is preferred because it requires no multiplication while achieving similar performance compared to the MSE.

2.6.3.1 Full-search Block Matching Algorithm (FBMA) [Jain81b]

Let $X_{i,j}$ be the n x n pixel *reference block* at coordinates (i,j), $Y_{i+k,\,j+l}$ be the n x n pixel *candidate block* at coordinates (i+k, j+l), and p be the maximum displacement. The SA is, therefore, of size $(n+2p)^2$ while the number of search locations is $(2p+1)^2$. The MAE distortion measure is given by:

$$MAE_{(k,l)}(X, Y) = \frac{1}{n \times n} \sum_{i=1}^{n} \sum_{j=1}^{n} |X_{i,j} - X_{i+k,j+l}| \qquad -p \le k, l \le p \tag{2.29}$$

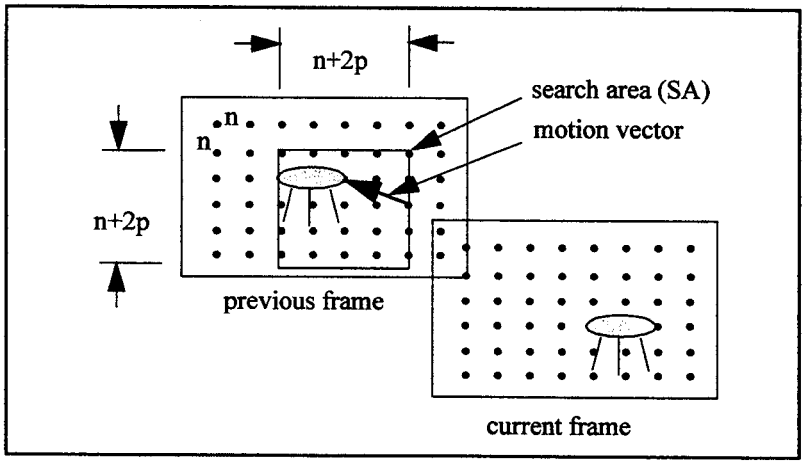


Figure 2.6 Block matching process

Fig. 2.6 illustrates the block matching process between adjacent frames. The current frame(t) is divided into non-overlapping reference blocks. Each reference block in frame(t), is compared with the candidate blocks within a SA in the previous frame(t-1) in order to obtain the best match. Prior to searching, the center[1] of the SA can be aligned with the center of the reference block. And after the search, the offset between the reference block and the best match (candidate) block specifies the motion (displacement) vector.

For each of the $(2p+1)^2$ search locations, the MAE calculation requires $3n^2$ operations (one subtraction, one absolute operation, and one addition). Thus, for K reference macro-blocks, the computational complexity of FBMA is of order $O\{3Kn^2(2p+1)^2\}$. In an MPEG video frame, for instance, typically of size 720 x 576 pixels, with 16 x 16 pixel reference macro-block and a maximum displacement of 16 pixels, more than 1.35 billion operations are required.

In order to reduce the computational complexity, three main approaches have been proposed in the literature. The first group suggests to reduce the number of search locations in the SA based on condition 4 of the block matching algorithms. This group includes: Three-Step hierarchical Search [Koga81], and its variant, Simple and Efficient Search [Lu97]; 2-D Logarithmic Search [Jain81b]; and Conjugate Direction Search [Srinivasan85]. The second group proposes to reduce the number of pixels involved in the distortion calculation (while maintaining the same number of search locations) based on condition 3 of the block matching algorithms. They are Pixel Decimation and Motion Field by [Liu93], and its variant "16:1 Decimation" by [Kim95]. The third group has recently emerged where the bit-depth of the involved images/frames is first reduced and the estimation process can be done using simpler hardware, such as XOR gates, on the binary edge images. The underlying assumption of this group is related to condition 2 of the

1. The upper-left coordinations may also be used.

block matching algorithms. Among the techniques are Pixel Difference Classification [Gharavi90], Bit-Plane Matching [Feng95], and Band-Pass Bit-Plane Matching [Natarajan97].

Incidentally and independently, a new low-complexity motion estimation algorithm [Le98a], called FEXOR, will be proposed and implemented in this thesis.

2.6.3.2 Group 1: Reduction in the number of search locations

In the Three-Step hierarchical Search technique (TSS), Koga *et al.* have proposed a *divide-and-conquer* search method where the entire SA is first divided into 9 sub-SA's. After the first iteration, the sub-SA which results in the lowest MAE will be further divided and searched until a 3 x 3 search window is obtained, and the final motion vector is determined (Fig. 2.7).

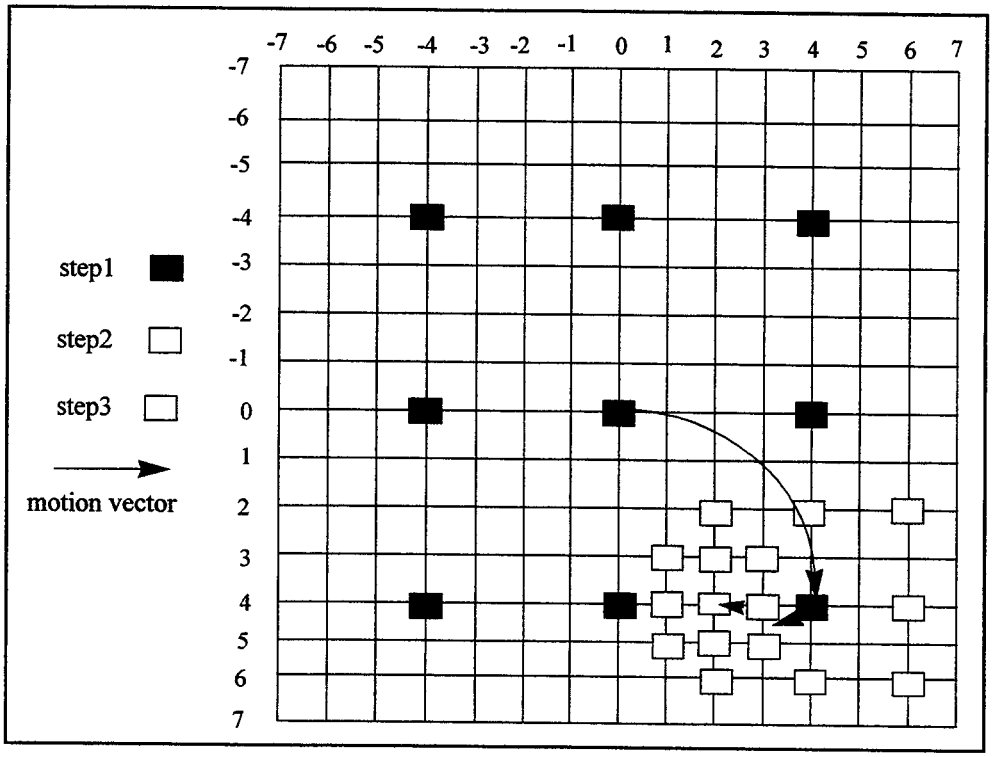


Figure 2.7 Three-Step Search

The TSS algorithm is reported to achieve a speed-up of 9 while it suffers a loss of 0.5 -1.5dB in performance compared to FBMA [Lu97].

Lu *et al.* [Lu97] further optimized the TSS by making directional decision, referred to as Simple and Efficient Search (SES), based on half of TSS's search locations. There are two phases to SES. Phase one is to identify a search quadrant, and phase two is to find the minimum error location in the selected quadrant. Let point A be the center of the SA (Fig. 2.8), points B and C (shown by clear boxes) be the search locations on the right of and below point A, respectively. The quadrants are numbered from I to IV in the counter clockwise direction. The procedure for identifying a search quadrant can be described as follows:

- If $MAE(A) \geq MAE(B)$ and $MAE(A) \geq MAE(C)$, quadrant I is selected;
- If $MAE(A) \geq MAE(B)$ and $MAE(A) < MAE(C)$, quadrant II is selected;
- If $MAE(A) < MAE(B)$ and $MAE(A) < MAE(C)$, quadrant III is selected; and
- If $MAE(A) < MAE(B)$ and $MAE(A) \geq MAE(C)$, quadrant IV is selected.

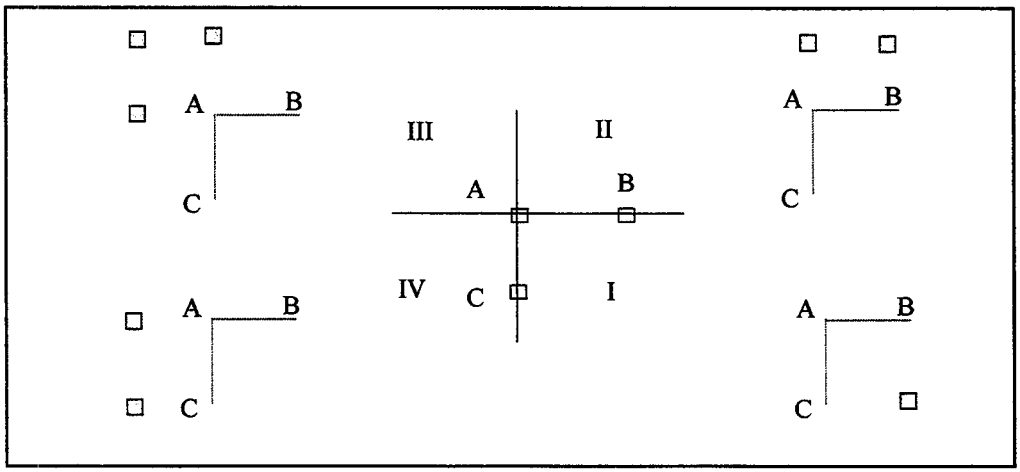


Figure 2.8 Simple and Efficient Search

Once a quadrant is selected, an average of two other locations are searched (solid boxes). Therefore, the number of search locations required in the first step is 5 (A, B, C, and 2 others); and the number of search locations needed in subsequent steps are 4 (new B, new C, and 2 others). Therefore, SES achieves a speed-up of nearly 2, while having a degradation of less than 0.1 dB compared to TSS.

In the 2-D Logarithmic Search (2-D LS), a moving window technique is applied. Five locations which contain the center of the search window and the four points corresponding to the North, South, East, and West directions will be searched. The search procedure starts from the center of the SA and proceeds to the N, S, E, and W directions, whichever results in the smallest distortion. Once the search focuses at a particular center, the step size between the search locations is reduced by a factor of 2. The search procedure is repeated until the search window is reduced to a 3 x 3 pixel window. In this final step, all 9 locations within the window are searched.

Finally, in Conjugate Direction Search (CDS), Srinivasan *et al.* propose a multiple 1-D search. Starting at point A, the search proceeds, location by location, in the horizontal direction. Once the lowest MAE is obtained, at point B, the search proceeds in the vertical direction. When the lowest MAE is obtained, at point C, the diagonal connecting A and C will then be searched. The best match location is assumed to lie on the line connecting AC.

2.6.3.3 Group 2: Reduction in the number of pixels involved

Liu *et al.* [Liu93] have proposed two algorithms for block motion estimation that produce similar performance to that of FBMA with computation reduced by a factor of 8 or 16. The algorithms are based on *pixel decimation* (PD) and *motion-field* (MF).

PD is developed based on the assumption that all pixels in a block move by the same amount, therefore, a good estimate of the motion could be obtained by using only a fraction of the total number of pixels in that block. This technique employs every other pixel of the block with alternating patterns for motion vector determination. Thus, a computation reduction of 4 is achieved while preserving the motion-estimation accuracy.

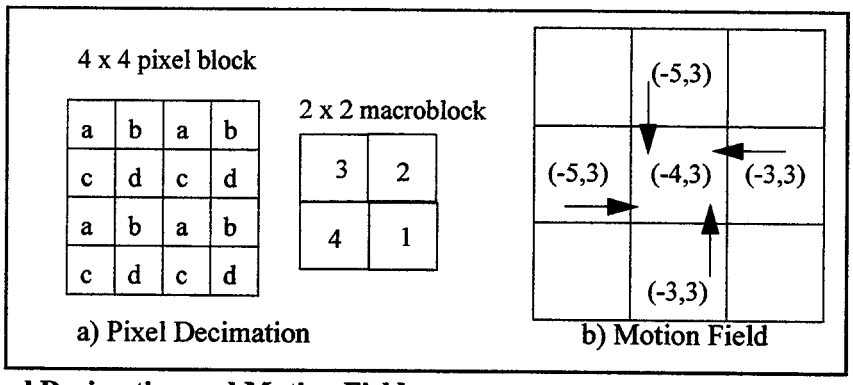


Figure 2.9 Pixel Decimation and Motion Field

Fig. 2.9a shows an example of alternating patterns used on a 4 x 4 pixel block. The motion vector of macroblock 3 is determined by group of pixels a, macroblock 2 uses group of pixels b, and so on. PD is reported to degrade the performance by approximately 0.1 dB compared to FBMA.

A collection of all motion vectors defines a motion field. Motion fields of image sequences are usually smooth and slowly varying, with discontinuities limited to the boundary of objects moving in different directions or with different velocities. As a result, it is not uncommon to find neighboring blocks with identical or near identical motion vectors. Motion estimation using motion field is performed in two steps. First, every other macroblock is estimated using any block matching algorithm. Then, the motion field is appropriately interpolated so that the motion vector of a block is determined from the predetermined motion vectors of the surrounding macroblocks (Fig. 2.9b). Interpolation requires only a small number of operations, so a subsampling of the motion field by a factor of 2 reduces the computational complexity by the same factor. The author also reported that subsampling the motion field by a factor of 2 causes only a minimal increase in prediction error [Lu93].

Later, Kim *et al.* [Kim95] extended PD to a 16:1 subsampling technique. This algorithm results in a degradation of less than 0.6 dB compared to PD.

2.6.3.4 Group 3: Reduction in the bit-depth of the pixel

Belonging to the class of low-complexity BMA's, Gharavi *et al.* [Gharavi90] have proposed a technique where the pixels involved are classified into matched or mismatched pixels, and hence the name Pixel Difference Classification (PDC). By definition, a pixel is matched when the absolute difference between the pixel in the reference block and the corresponding pixel in the candidate block is less than a threshold. The candidate block with the largest number of matching pixels is chosen to be the best match. The selection of matched/mismatched pixels primarily relies on a threshold which is subject to illumination changes within a frame or among frames. Feng *et al.* [Feng95] have introduced a Bit Plane Matching (BPM) technique, where the block mean is used as the threshold in contrast to an arbitrarily chosen threshold in PDC. If a pixel value is greater than the block mean, it is set to 1, otherwise, it is set to 0. Since block mean is required, there is a blocking effect on the generated binary frames, hence, the estimation performance is reduced by the block boundaries. In Natarajan *et al.*'s technique [Natarajan97], a 16 x 16 convolution kernel K which behaves like a band-pass filter, is applied, where:

$$K_{i,j} = \begin{cases} \frac{1}{25} & i,j \in [1, 4, 8, 12, 16] \\ 0 & otherwise \end{cases} \tag{2.30}$$

If a pixel in the original frame is greater or equal to the corresponding pixel in the filtered frame, the resulting binary pixel is set to 1; otherwise, it is set to 0. This technique, referred to as Band-Pass Bit Plane Matching (BP_BPM), is superior to PDC and BPM.

In fact, BP_BPM is a modification to BPM. While BPM calculates the thresholds using 256 values on fixed 16 x 16 windows, BP_BPM calculates the thresholds based on a 4-spacing-grid moving 16 x 16 windows. BP_BPM is, therefore, able to remove the blocking effect.

The image and video processing techniques presented are included in the compression standards. In the following section, a non-standard compression technique is presented.

2.7 Vector Quantization

Vector Quantization (VQ) [Gray84] is an alternative technique for low bit-rate image and video compression. Nasrabadi *et al.* [Nasrabadi88] have presented a review of VQ for image compression. In VQ, the input image is first decomposed into a set of input vectors. A codebook is available at both the transmitter and receiver prior to transmission.

The basic steps in VQ are quantization and decoding (Fig. 2.10). In the quantization process, for each input vector, the codebook is searched to obtain the closest representative vector (or codeword). Compression of $(L*b)/(\log_2 N)$ is achieved by representing a vector of L b-bit elements by its $\log_2 N$-bit address to a codebook. Reconstruction of image can be implemented by a simple table look-up procedure where the label is used as an address to a table containing the codewords. The steps involved in VQ as applied to image and video compression are: vector formation, codebook generation, and quantization.

2.7.1 Vector Formation

The first step in VQ is to partition the input image into 2-D 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 original pixel values of the block [Gersho82b]; the color components of a pixel [Yeh87]; the pixel values of a block normalized by the mean and standard deviation [Murakami82]; the pixel values normalized by the prediction error of a block [Rutledge87, Gupta91]; and the transformed coefficients of blocks of pixels [Dinstein90, Kim91a, Huh94].

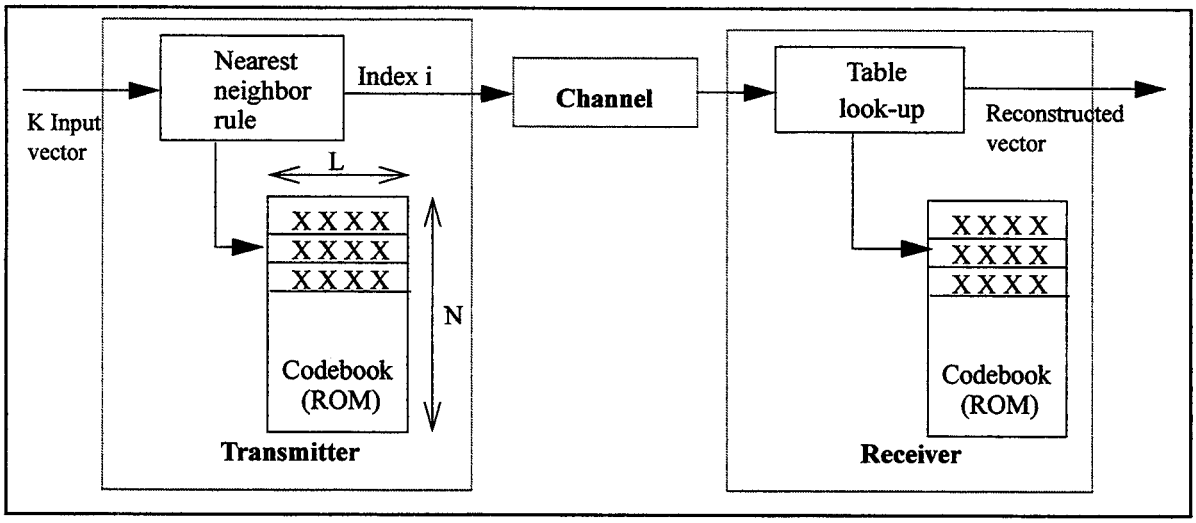


Figure 2.10 VQ for Image Compression

2.7.2 Codebook Generation

The key element in VQ is the design of a codebook which is a good representative of the image vectors. An optimal codebook, using the MSE criterion, must simultaneously satisfy two necessary conditions for optimality [Gersho82a]. Details of codebook generation techniques are presented in Section A.3.

2.7.3 Quantization

VQ techniques can be broadly classified, with respect to training and codebook generation, as universal and adaptive.

Universal VQ: 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 compression complexity. Moreover, if the input images have different statistics, good compression performance may not be achieved for a variety of applications. The codebook size can be reduced using techniques which exploit the local image statistics.

In order to speed up the quantization process, fast algorithms, typically the Tree-Search VQ (TSVQ), have been proposed. In TSVQ, the computation necessary for finding the matched codeword can be reduced by arranging the codebook in a binary tree [Buzo80]. The tree is constructed with a codeword 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 Full Search VQ (FSVQ), the computational complexity is reduced to O(KLlogN) (Fig. 2.10) at the expense of doubling the storage cost, e.i. L*2(N-1) [Gray84].

The K-dimensional (K-d) tree is a generalization of the binary tree and provides a data structure which allows for multi-dimensional search. Ramasubramanian *et al.* [Ramasubramanian92] 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 FSVQ. To improve the coding performance of TSVQ, Riskin *et al.* [Riskin91] have presented an unbalanced TSVQ where the tree is designed one node at a time rather than a layer at a time.

Adaptive VQ: The universal codebook has two shortcomings which may degrade coding 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 be required 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.

2.7.4 Vector Quantization for Temporal Compression

Several image and video sequence compression 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.* [Murakami82] 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.* [Guo92] 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-element 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.

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 *et al.* [Goldberg86a, 86b] have presented several interframe coding techniques where changes in successive frames are tracked by incorporating *label replenishment* (LR) and *codebook replenishment* (CR). Generally, frame adaptive techniques result in a further increase in computational complexity.

Recently, Le *et al.* [Le95] 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 following frame. In the ME method, the label of the current frame is compared to a SA of the corresponding labels in the previous frame and the resulting motion vector is transmitted. Huffman-coded flags are required to achieve further compression among the different components of the resulting bit stream.

Having reviewed the standard and non-standard compression techniques, it is now appropriate to have an overview of the existing and emerging image and video compression standards.

2.8 Image and Video Compression Standards

2.8.1 The JPEG Image Compression Standard [ISO/IEC 10918-1]

JPEG, jointly developed by ISO and ITU-T, is a standard for still image compression. Compression ratios typically ranges from 10 to 50 without visibly affecting image quality. The JPEG standard supports the following four modes:

- **Baseline sequential mode**: provides the basic features of DCT-based compression for full quality and full size image reconstruction. Baseline sequential mode and the following 3 modes use a lossless coder at their final stages;
- **Lossless mode**: is targeted towards applications, such as medical imaging, which require perfect image reconstruction. A predictive coder is used instead of a DCT-based scheme. In this scheme, each pixel is predicted based on 3 adjacent pixels using one of eight possible predictor modes;
- **Progressive mode**: Image is encoded in multiple scans, from coarse to fine, and progressively improved by transmission over a low bandwidth transmission channel. Two procedures have been suggested for progressive coding: *spectral selection* (based on zig-zag scanned frequency bands) and *successive approximation* (based on bit planes of the DCT coefficients). Successive approximation is sometimes referred to as *SNR scalability*; and
- **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 stream. This mode is used for implementing *spatial scalability.*

Due to its popularity, the baseline sequential mode for grayscale image compression is chosen for review. Color image compression will be reviewed subsequently. In the baseline sequential mode, image sizes are allowed up to 4096 x 4096 and samples are quantized using 8 bits with values ranging from 0 to 255[1]. Compression is performed in 3 steps: DCT computation, quantization, and entropy coding (Fig. 2.11).

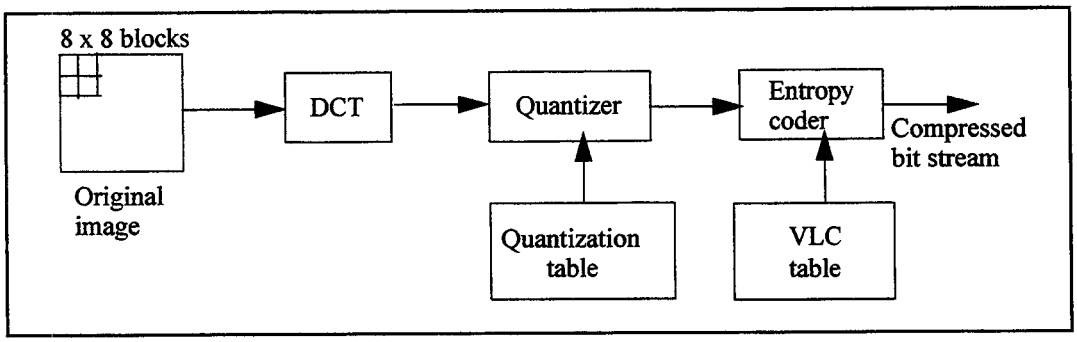


Figure 2.11 JPEG Baseline Sequential Mode

The original image is first partitioned into non-overlapping 8 x 8 pixel blocks. The pixel values are level-shifted into the range (-128, 127) with zero being the center[2]. The DCT of the block is then computed and quantized using a quantization table suggested by JPEG. This quantization table is usually fixed for common sets of images such as faces, medical, etc. The DCT computation results in 64 coefficients. Entropy coding is performed in 2 stages. In the first stage, the first zig-zag scanned coefficients is the DC coefficient[3], the rest are AC coefficients. DC coefficients of adjacent blocks are differentially coded using DPCM, while AC coefficients within a block are coded using run-length coding. In the second stage, either Huffman coding or arithmetic coding is used to generate the compressed bit stream. The decoding scheme is just the inverse of the encoding scheme and is, thus, not shown.

1. Higher bit depths such as 12-b and 16-b are also supported.
2. In general, a b-bits pixel is level-shifted to $(-2^{b-1}, 2^{b-1}-1)$.
3. This represents the average grayscale of the block.

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, 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, is therefore, not a part of the data stream.

Color Image Compression: Image data is captured and stored in memory as individual RGB components. Since the RGB components are still highly correlated, a color-space conversion is performed. Typical conversion is RGB-to-YC_bC_r. The luminance Y, and chrominances C_b and C_r are obtained using the following transformations:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} \tag{2.31}$$

where R', G', and B' are the gamma-corrected RGB values (Section 2.1). The luminance component Y represents the grayscale component, which carries most of the image information. The chrominance components, C_b and C_r, carry less information, and thus can be subsampled to reduce the amount of storage and processing.

Various subsampling schemes are available, including 4:2:2 where only the number of pixels on a line is subsampled by 2, and 4:2:0 where both the number of pixels and the number of lines are subsampled by 2.

Let the sizes of the image be 2H and 2V, respectively. The sizes of (Y,C_b,C_r) under (4:2:2) and (4:2:0) subsampling schemes are (2Hx2V, Hx2V, Hx2V) and (2Hx2V, HxV, HxV), respectively. Therefore, by simple color-space conversion, the compression ratios are (3x2Hx2V):(2Hx2V, Hx2V, Hx2V) and (3x2Hx2V):(2Hx2V, HxV, HxV); or (1.5:1) and (2:1), respectively.

After color-space conversion and subsampling, the pixels are level-shifted and processed as mentioned in the grayscale compression scheme.

Motion JPEG: The expansion in low cost hardware for JPEG has led to the development of an additional mode of operation for video sequences: motion-JPEG. Under motion-JPEG, each frame of a video stream is compressed independently using the baseline sequential mode. However, there is no standard syntax for motion-JPEG coded streams.

2.8.2 The MPEG Video Compression Standards

2.8.2.1 MPEG-1 [ISO/IEC 11172]

Many applications have been proposed based on the assumption that an acceptable quality of video can be obtained with a data bandwidth of about 1.5 Mbps. The key requirements outlined in MPEG-1 are compression and random accessability. Many of the applications using MPEG-1 video have been optimized for the Source Input Format (SIF), which is a simple derivative of the CCIR 601 formats for video frame in digital television. The CCIR 601 and MPEG-1 frame characteristics are listed in Table 2.2 below:

Table 2.2 CCIR 601 and MPEG frame characteristics

Frame type	Processing rate	Lines	Luminance	Chrominance	Format
CCIR 601/ NTSC	60 frames/sec.	525	720 x 480	360 x 480	4:2:2
CCIR 601/PAL	50 frames/sec.	625	720 x 576	360 x 576	4:2:2
MPEG / SIF	30 pictures/sec.		360 x 240	180 x 120	4:2:0
MPEG / SIF	25 pictures/sec.		360 x 288	180 x 144	4:2:0

A *group of pictures*[1] (GOP) is defined. A GOP is a combination of one or two *intra-picture* (I), *predicted pictures* (P), and the rest of *bidirectional pictures* (B). Typical GOP, where N - the distance between I-frames - is 9 and M - the distance between 2-P frames - is 3, has 1 I-frame, 2 P-frame, and 6 B-frames (Fig. 2.12).

1. Inputs are non-interlaced pictures.

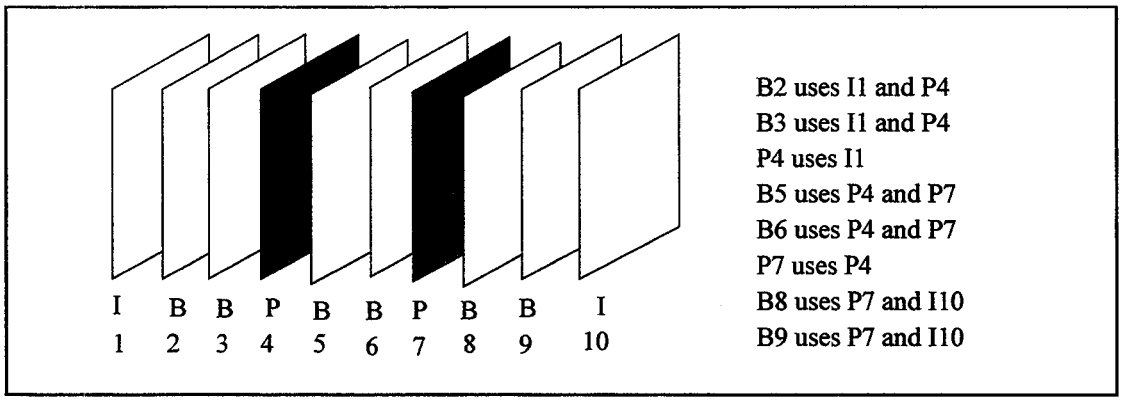


Figure 2.12 A typical MPEG-1's GOP

The I pictures provide random access points and are also used as references for P pictures. The I pictures are encoded using DCT on 8 x 8 pixel blocks and the resulting coefficients are quantized. The quantized DC coefficients of adjacent blocks are differentially encoded using DPCM, while the quantized AC coefficients are zig-zag scanned and ordered into {RUN, AMPLITUDE} pairs. A variable length coder is used to encode each pair. I picture is encoded similarly to applying JPEG compression to individual frames.

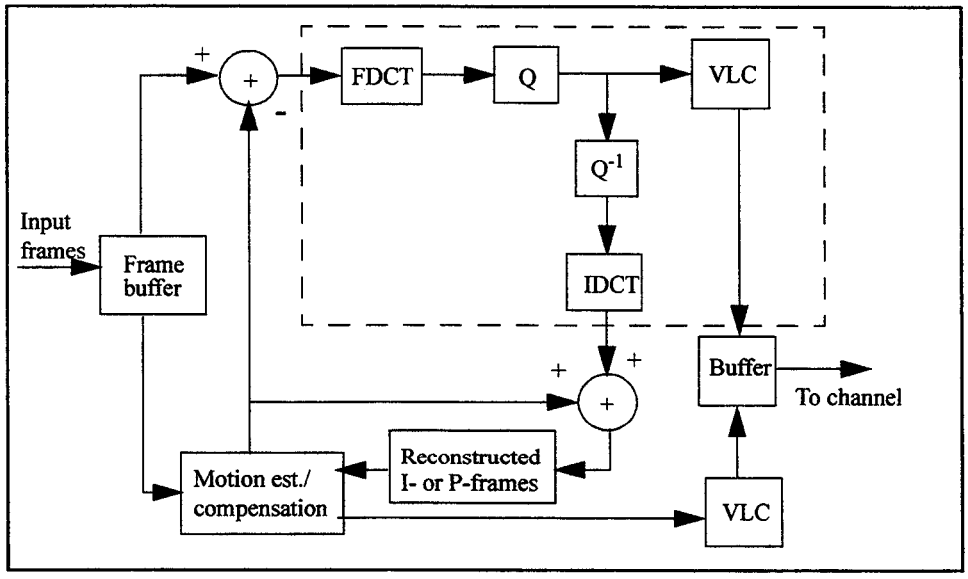


Figure 2.13 Block diagram of MPEG-1 encoder

The dotted line in Fig. 2.13 depicts the encoding process of I frame. In addition to FDCT and Quantization, a copy of the encoded picture represented by Inverse Quantization and IDCT, as obtained by the decoder, is needed for future predictive coding. It is noted that the VLC blocks are similar to the entropy coder block in Fig. 2.11.

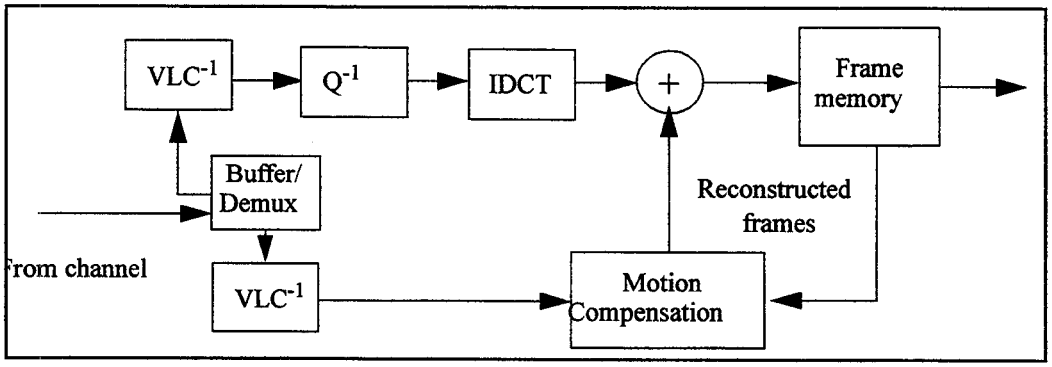


Figure 2.14 Block diagram of MPEG-1 decoder

The P and B frames, on the other hand, are partitioned into 16 x 16 pixel macro-blocks and the motion vector for each block is determined using *interframe predictive coding* and *interframe interpolative coding*, respectively. The motion-compensated Differential Frame Prediction (DFP) error is partitioned into 8 x 8 pixel blocks which then undergo a 2-D DCT. The DC and AC coefficients are quantized, ordered along the zig-zag scanned line into {RUN, AMPLITUDE} pairs and coded using a VLC. The motion vectors are also variable length coded and transmitted. Block diagrams of the MPEG-1 encoder and decoder are shown in Fig. 2.13 and Fig. 2.14, respectively.

2.8.2.2 MPEG-2 [ISO/IEC 1318-2]

MPEG-2 is targeted at higher bit rates - 4 to as high as 80 Mbps - and broader range of applications including consumer electronics and telecommunications. MPEG-2 supports multiple profiles[1] and multiple levels. Some features of MPEG-2 video are summarized below:

1. Profile refers to the functionalities such as non-scalable, scalable. Level indicates the picture resolutions.

- Supports more than one frame size and multiple formats, for instance 4:2:2 and 4:4:4 in both progressive and interlaced modes in addition to a single progressive 4:2:0 SIF in MPEG-1. Under interlaced mode, a frame is divided into odd- and even-fields. Fields can be coded separately or interleaved and coded as one frame;
- Supports various scalable modes such as: SNR, spatial, temporal;
- DCT Field (progressive) or frame (interlaced), and scanning of DCT coefficients: both zig-zag scan and alternate scan for interlaced video;
- Motion Estimation: Field, frame, and dual-prime based; and motion vectors: concealment motion vectors for I pictures besides motion vector for P and B;

Commonly used configurations are Main profile at Low level (MP@LL: 360 x 240 at 30 fps), Main profile at Main level (MP@ML: 720 x 480 at 30 fps), and Main profile at High level (MP@HL: 1920 x 1080 at 60 fps). The MP@HL configuration has been adopted by the HDTV Grand Alliance.

2.8.2.3 MPEG-4

The focus of MPEG-4 standard is audiovisual coding in multimedia applications, allowing for interactive, high compression, and scalability for video and audio contents. MPEG-4 is centered around a basic unit of content called the audio-visual object (AVO). The object-based encoder codes data at rates ranging from 8Kbps to as high as 1Mbps. The object notion allows for mixing of natural and synthetic objects as well as other data types such as text overlay and graphics.

In the MPEG-4 architecture, one or more AVOs, including their spatio-temporal relationships, if any, are transmitted from a source to an MPEG-4 decoder. At the source, the AVOs are error protected, compressed, and multiplexed for transmission. At the decoder, the AVOs are demultiplexed, decompressed, and presented to the end user. The end user may interact with the

presentation. Interaction information can be processed locally, or can be transmitted upstream to the encoder for action.

This type of representation requires that the compression scheme is able to handle arbitrary shapes (as opposed to block-based coding), and also that the receivers not only have the capability to decompress the compressed information but also able to compose them together.

2.8.3 The Teleconferencing Video Compression Standards

2.8.3.1 H.261 standard [ITU-T rec. H.261]

CCITT[1] H.261 is designed for video telephony and teleconferencing systems using the existing Integrated Services Digital Network (ISDN). The standard encodes video at a *constant bit rate* (CBR) of p x 64Kbps, where p=1, 2, ..., 30 by using DCT and motion estimation/compensation techniques. The coder output rate is kept constant for transmission over a CBR channel by using quantization step control based on the fullness of the output buffer.

Two resolution formats each with an aspect ratio 4:3 are specified, the *Common Intermediate Format* (CIF) defines a luminance component of 288 lines, each with 352 pixels per line . The chrominance components have a resolution with a rate of 144 lines and 176 pixels. Quater CIF (QCIF) has exactly half the CIF resolution in both dimensions. CIF is made optional, while all H.261 implementations must be able to encode and decode QCIF.

The H.261 standard uses both intraframe and interframe coding approaches. For interframe coding, information from previous or subsequent frames is used; this corresponds to the P- and B-frame encoding of MPEG.

1. CCITT stands for Consultative Committee on International Telephony and Telegraphy.

Similarly to JPEG, for intraframe coding, each 8 x 8 pixel block is transformed into 64 coefficients by a DCT. After quantization, the subsequent step is to apply entropy coding to the AC and DC coefficients, resulting in variable length encoded words. Interframe coding is based on motion estimation technique. Note that according to H.261, the coder need not be able to determine 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 vectors and the DPCM coded macro blocks are processed. A DPCM coded macro block is transformed by a DCT if and only if its value exceeds a certain threshold. This threshold is set depending on the quantization matrix and the buffer fullness. If the difference is less than this threshold, the corresponding macro block is not encoded any further; the relevant motion vector is just entropy coded with a variable length coder. If a DCT is needed, all of the transformed coefficients are quantized linearly and variable length coded.

For H.261, quantization is a linear function 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.

2.8.3.2 H.263 standard [ITU-T rec. H.263]

The purpose of H.263 is to improve the coding performance to provide an acceptable picture quality at low bit rate of less than 64 Kbps. The basic structure of the video coding algorithm H.263 is taken from the H.261 standard. H.263 can operate on sub-QCIF (128 x 96), 4CIF (740 x 576), and 16CIF (1408 x 1152) standardized picture formats in addition to QCIF and CIF. The coder has motion compensation capability with half-pixel precision, in contrast to H.261 which uses full-pixel precision and employs a loop filter. H.263 also includes four coding options which

provide improved coding efficiency: *unrestricted motion estimation* where vectors are allowed to point outside the picture, *arithmetic coding* as opposed to Huffman coding, *advanced prediction* which includes overlapped block motion compensation, and *PB-frame mode* which combines a bidirectional predicted picture with a normal forward predicted picture.

2.9 Summary

In this chapter, the image and video processing algorithms and JPEG, MPEG-1, -2, and -4, and H.261 and H.263 compression standards have been reviewed. JPEG has been selected as a intraframe compression scheme for video compression standards such as MPEG's and H.26x, therefore, it can be considered as part of MPEG's or H.26x standards regarding implementation.

Brief derivation of the luminance component of an image has been outlined in Section 2.1, and revisited in details in Section 2.8. General and morphological image processing techniques have been presented in Sections 2.2 and 2.3, respectively. Data compression techniques along with the subjective and objective evaluation methods have been discussed in Section 2.4. In Section 2.5, the spatial-domain compression techniques have been presented with emphasis placed on the techniques employed in the compression standards such as DCT and entropy coding. In Section 2.6, temporal-domain compression techniques have been reviewed, and a thorough study of Motion Estimation has been provided. Finally, in Section 2.7, a low bit-rate compression technique for table look-up applications, namely, Vector Quantization, has been reviewed.

CHAPTER 3

C*RAMs and Other Logic-in-Memory SIMD Designs: A Review

The first construction of a SIMD machine dates back to the 60's starting with Illiac IV. This machine was the most infamous supercomputer project in history because it scored less than one tenth the performance and exceeded four times the cost of the original goals [Falk76]. It was a 64-processing element (PE) system in which each PE had a 64-b ALU and a 128Kb memory module.

In spite of the failure of Illiac IV, subsequent SIMD projects were successfully built and can be categorized into two main groups: bit-slice and word-wide architectures [Hwang93]. The former group includes the Massively Parallel Processors (MPP) by Goodyear, the Distributed-memory Array Processors (DAP610) by Active Memory Technology, and the Connection Machines (CM2 and CM5) by Thinking Machine Corporation. CM5 can be operated in 3 different modes: synchronized MIMD (Multiple-Instruction stream, Multiple-Data stream), multiple SIMD, and SIMD. The latter group includes the BSP (Burroughs Scientific Processor) which uses shared

memory, GF11 by IBM (International Business Machine Corp.) which is developed for scientific simulation, and MP-1 by MasPar which is still in use. The SIMD machines built to date, mostly developed for scientific applications, were funded and supported by government or academic institutions. The dream of massively parallel computing at the personal or consumer level is yet to be realized.

During the last few years, SIMD array processors have been considered as an attractive alternative in personal computer peripherals and consumer visual products due to their low cost potential and fast parallel executions of low- to medium-level image and video processing tasks. The growing processor-memory bandwidth gap, as discussed in Chapter 1, will result in a loss of performance since a fast processor has to wait for a number of cycles to obtain the operands from a slower memory. To narrow down the processor-memory gap, the usual approaches, at the system level, are to use large data/instruction caches and to use specialized high speed interfaces such as Rambus [Farmwald92]. At the chip level, however, one solution is to embed memory modules into Application Specific Integrated Circuit (ASIC) processor arrays to remove the data transfer bottleneck at the I/O pins. The other is to integrate modular logic circuitry into the existing RAM to take advantage of the inherently high memory bandwidth while exploiting parallelism across the memory columns. The former is sometimes referred to as *logic-with-memory*, while the later is referred to as *logic-in-memory*. In the following sections, logic-in-memory array processors will be reviewed in the chronological order. Comparisons to the first C*RAM chip (C*RAM92) will be made to reflect the features of each design.

3.1 C*RAM (C*RAM92) [Elliott92]

Computational RAM (C*RAM) is a memory-SIMD hybrid architecture where a processing element (PE) is pitch-matched to a group of column sense-amplifiers for simple, yet massively parallel processing. C*RAM was primarily designed with the goal of augmenting conventional

RAM's (used in computer main memory and video buffers) with computation capability. The C*RAM concept make use of the large on-chip bandwidth to perform massively parallel bit-serial computations. There are two major functions in a C*RAM: memory and computation. When functioning as a memory device, C*RAM is read or written as part of the host processor address space. When functioning as a compute engine, all PEs execute operations (on their own local memory) in parallel, sequenced by a controller.

The first C*RAM has been fabricated in a 1.2μm SRAM technology (Fig. 3.1). There are 64 PEs attached to a 8Kb memory column which may effectively be considered as 64 independent computing units (CU's) having one PE and 128 bits of local memory. Each PE (Fig. 3.2) has two 1-b registers, X and Y, and a 1-b ALU implemented as a table look-up multiplexer. Inputs to the bit-serial ALU can be contents of any non-repeating ordered combination of the registers X, Y and a memory bit M. A 8-b global instruction is sent from the C*RAM controller and PEs with different (M,Y,X) combinations will select different 1-b result out of the 256 standard/user-defined functions. After each computation, the results are written back into the respective PEs local memories or registers. A truth table of a 3-input standard logic function (XOR) and a 3-input user-defined function (CARRY) for bit-serial addition are also shown in Fig. 3.2.

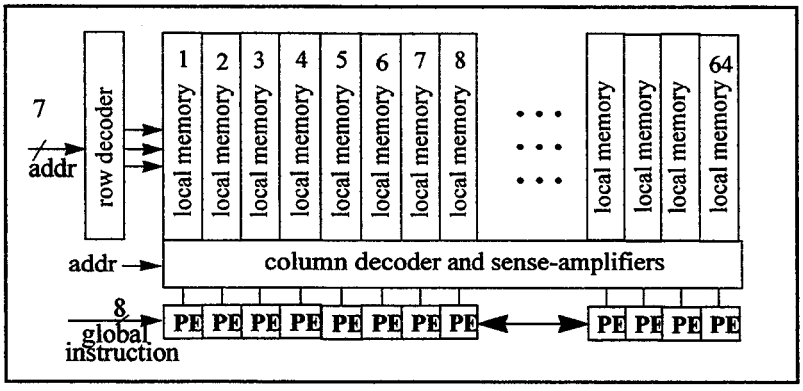
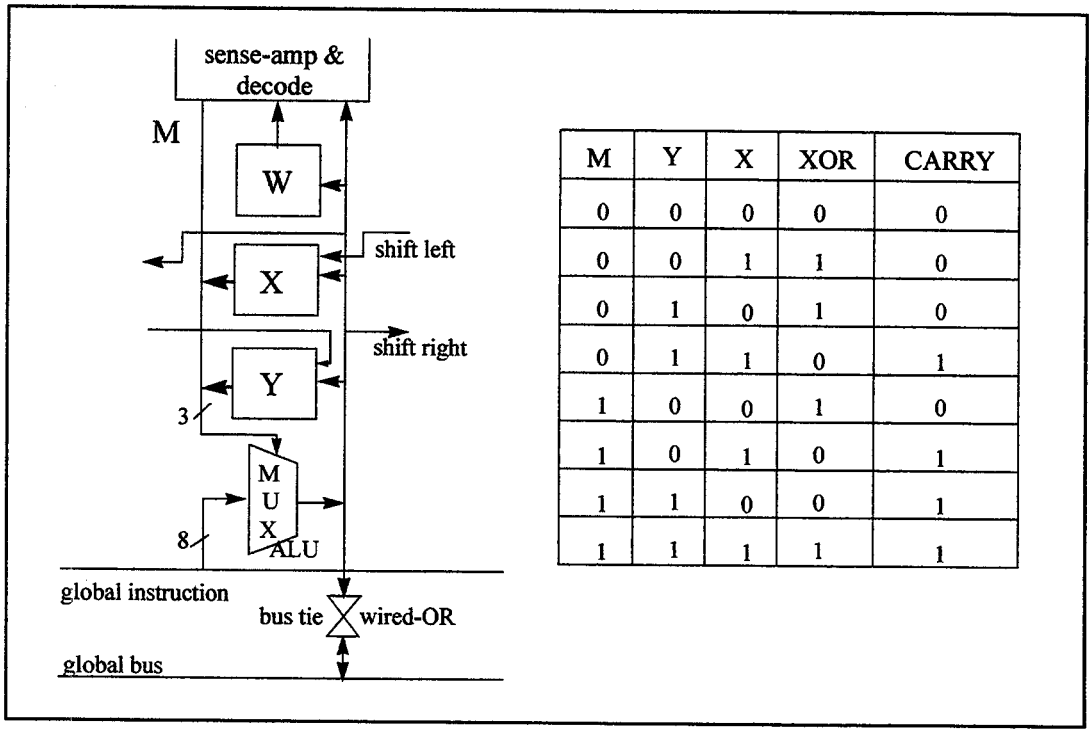


Figure 3.1 C*RAM Architecture

In addition to the X,Y registers and ALU, the Write Enable (W) register permits conditional operations. The W register functions as a mask, blocking write back to undesired memory columns in various operations. This feature enables all the PEs in the C*RAM to participate in global operations without modifying the masked memory contents.

Communication between PEs is via a nearest neighbor linear network. Each X (or Y) register has a left (or a right) connection enabling data shift to neighboring PEs. Global communication is performed via a bidirectional bus-tie circuit which performs the wired-OR logic of all PEs local results. The result of this global wired-OR can be written back to all the PEs registers or local memories.



M	Y	X	XOR	CARRY
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 3.2 The PE Model

A number of C*RAM variants [Cojocaru93], [Ross96], and [McKenzie97] have been designed and fabricated. In his first chip, Cojocaru designed a 64-PE C*RAM (C*RAM93) fabricated using a 0.8μm BiCMOS SRAM technology [Silburt92]. The architecture and PE design are

similar to that of C*RAM92 except that more memory is added to each PE (1Kb). The data and address buses are used in a time-sharing fashion. During memory accesses, these buses carry data and address as usual. Between consecutive memory accesses, they can be used to transmit 8-b global instructions and control instructions if C*RAM mode is set. In memory mode, the read/ write cycle is 40 ns, and in C*RAM mode, the *read-operate-write* cycle time is 60 ns. The *operate* cycle can sometimes be sandwiched between the C*RAM read/write cycle, which effectively results in a 20ns cycle time. Therefore, when all 64 PEs are operating in parallel at 25MHz, C*RAM93 can achieve a peak performance of 3.2GIPS.

A controller [Nyasulu98a] has been built for C*RAM93. It provides the interface between C*RAM and its host processor. Typically, a C*RAM system is implemented as an extension card on the host computer expansion bus. C*RAM and its controller integrated architecture have also been investigated. In this implementation, the controller interfaces the C*RAM to the PCI bus for PC environment. The PCI Interface Unit implements a 33MHz/32-bit PCI target interface protocol compliant with the PCI Local Bus Specification Revision. Burst data transfer and parity check are supported. Data and instruction transfer rates between host and the C*RAM controller is 132MBps, while the transfer rate between the C*RAM controller and C*RAM is 25MBps[1]. The current controller prototype has been implemented on a Xilinx XC4013E-2 Field Programmable Gate Array (FPGA).

Another variant of C*RAM was reported in [Foss96] (C*RAM96). A group of 4K PEs is connected to a 16Mb DRAM resulting in 4K CU's each having 4Kb of local memory. The PEs are pitch-matched to groups of 4 DRAM columns. Later in [McKenzie97], a 1K PE C*RAM is implemented on a 16Mb DRAM module (C*RAM97). Distinctions from previous C*RAM variants include: the introduction of column redundancy and the elimination of extra control pins.

1. Controller speed can be matched with C*RAM speed. In this case, C*RAM93 uses a 8-b bus running at 25 MHz.

This design, therefore, maintains Joint Electron Devices Engineering Council (JEDEC) pin-compatibility with conventional DRAM.

The most recent variant of C*RAM is the Integrated Graphics Accelerator and Frame Buffer [Torrance98] (IGA-FB). It's 4K PEs (or Pixel Processing Unit) are integrated onto a 13.4Mb dual-port DRAM array. A new orthogonal databus is implemented in addition to the standard bus direction. This allows up to 1 databus for each DRAM column. The PE registers are built using dual-port 6-transistor SRAM cells, while the ALU (or RASTOP) is implemented similar to C*RAM92 where a 8-to-1 multiplexer is used to select among 256 operations based on the 3 input variables. The main clock runs at 66MHz while the pixel clock runs at 135MHz. This architecture allows the graphics controller to accelerate some of the most common graphics operations such as: screen clear, block write, block move, color expand, and horizontal moves.

3.2 Integrated Memory Array Processor (IMAP94) [Yamashita94]

IMAP, a NEC project, consists of a 1-D array of 64 processors embedded onto a 2Mb video RAM[1] (VRAM), along with serial shift registers, row registers, and two I/O shift registers. IMAP does not have a sequencer nor a program memory.

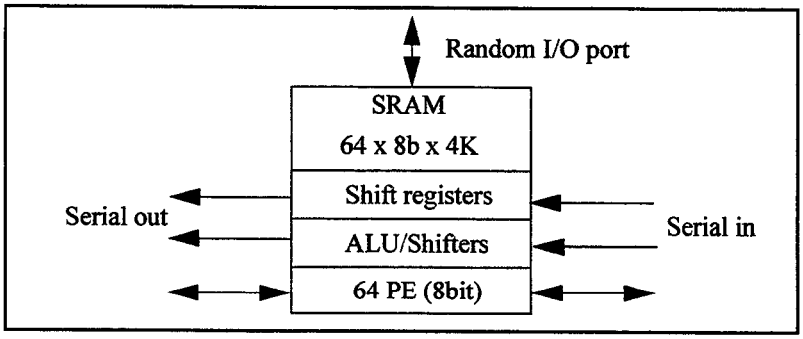


Figure 3.3 Integrated Memory Array Processor

1. VDRAM has two access ports: one random and one serial. The random port interfaces with the processor or controller. The serial port feeds data from a register, which is loaded by an internal transfer from the sense amplifiers. The register data flows out through the port at the speed required by the video display. VDRAM is excellent for very fast graphics manipulation in many high-end PCs, low-end workstations, and other applications that require great performance but only modest memory.

Similar to C*RAM92, instruction streams are provided from an off-chip controller and latched into the pipeline registers and then broadcast to all PEs. Video image I/O is performed independently using the internal processing clock via serial ports. The image data can also be accessed through a random access port (Fig. 3.3).

Each PE has an 8-b processor consisting of an ALU/shifter, 12 general registers, a data register, an address register, an inter-PE register, and a mask register. The mask register enables specific PEs local memory to stay out of global operations. Data exchange among adjacent PEs is performed through an inter-PE register. The address register is used to implement index-addressing mode which is preferable for histogram and Hough transform operations. Using the memory data register and the address register, the memory read/write operation and ALU/shifter computation can be executed simultaneously. Since all 64 PEs are operating in parallel at 40MHz, IMAP can achieve a peak performance of 3.84GIPS, and 1.28GB/s on-chip processing bandwidth. Compared to C*RAM92, the IMAP has been designed using VRAM for non-overlapping processing and I/O operations, with wider data path, and index-addressing for histogram and Hough transform operations.

3.3 PC-RAM [Cojocaru94]

In this 0.8μm SRAM-based parallel computational RAM (PC-RAM), the PEs functionality is extended from its original PE of C*RAM92 to allow non-transposed memory accesses and bit-parallel operations. The baseline PE offers only two data registers X and Y. The third data input to the ALU, M is assumed to be registered in the memory interface circuitry. In order to support bit-parallel multiplication, four new registers are defined in the extended PE (Fig.3.4). The M register is now the explicit source of data line M. M is a dual access latch and can be written by both the ALU and the local memory output MQ. Three more registers, AX, AY, and AM constitute the *alternate register set*. The word "alternate" indicates that only one in a register - alternate register

pair can source the ALU at a time. The selection is made by three 2-to-1 multiplexers, controlled by three new global signals: XS, YS, and MS.

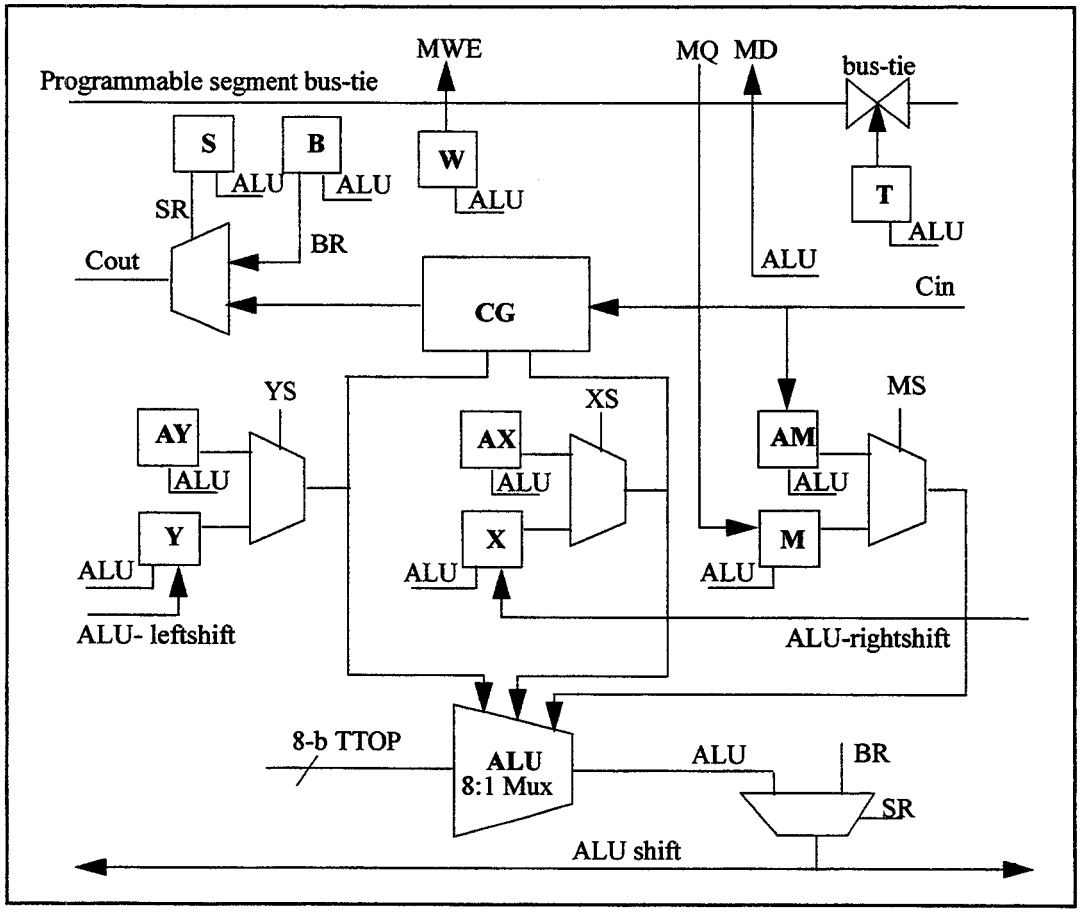


Figure 3.4 The Extended PE Model

Multiples of 2 PEs can be grouped to form a CU which is implemented using a bus-tie register T placed at every other PE. The value stored in this register is used to disable the transmission gate connecting the *programmable segment bus-tie* (PSB). All ALU outputs sharing the same segmented bus are wire-ORed, and the result is fed back to the registers of the corresponding CU. The PSB circuit is extensively used in zero-value checking and bit-extension operations.

A *ripple-carry* circuit is implemented by adding the carry generator (CG) block. Recall that for addition (or subtraction), two operations are required: SUM (or DIFFERENCE) and CARRY (or BORROW) generations. Inputs to the CG block are the *carry-in* (C_{in}) from the right-neighbour

PE, and 2 other operands stored in registers X and Y (or alternatively, AX and AY). We note that the carry-in is set to 0 for addition and 1 for subtraction. The *carry-out* (C_{out}) will propagate to the left-neighbor PE if the *programmable word boundary* (PWB) is not reached. Otherwise, the carry-out is stopped. The PWB is implemented by the Select register (S) and By-pass register (B), placed at every other PE. The bit-parallel C*RAM eliminates the need for a data transposer between itself and the host computer. The extended PE is larger which effectively reduces the number of PEs by 30%. When all 512 PEs are operating in parallel at 25MHz, PC-RAM can achieve a peak performance of 25.6 GIPS.

In summary, in addition to the existing functionality offered by C*RAM92, PC-RAM has been extended with the following functionalities: PSB, PWB, and ripple-carry circuit for addition, and the extended register set to allow for word-parallel operations.

3.4 SRC-PIM [Gokhale95]

The Processor-in-Memory (PIM), developed at the Supercomputing Research Center (SRC) in Maryland, is a chip that integrates a linear array of 64 bit-serial PEs with 2Kb of local SRAM.

There are three primary registers, A, B, and C, supplying inputs to the ALU (Fig.3.5). At each clock cycle, the pipelined ALU can perform either a memory read or a memory write operation. Also, at each clock cycle, the ALU can produce three outputs that can be either selected for storage (under an internal mask control) or selected for recirculation. In addition, data can be sent to other processors via a routing network.

Interprocessor communication is conducted through a global OR network, a partitioned OR network, and a parallel prefix network (PPN). The partitioned OR network can execute logic-OR on groups of power of 2 of PIM chips. The PPN network allows PEs at fixed positions to send data to groups of neighbouring PEs in 16 configurations. Level 0 is a bidirectional linear nearest-

neighbour network. Levels 1 to 15 allow only left communications where at level i, any PE_n can communicates with PE_{n-1}, PE_{n-2}, ..., up to and including PE_{n-2**i}. Levels 1 to 9 are realized in hardware while the rest are configured in software. When all 64 PEs are operating in parallel at 10MHz, SRC-PIM can achieve a peak performance of 0.64 GIPS.

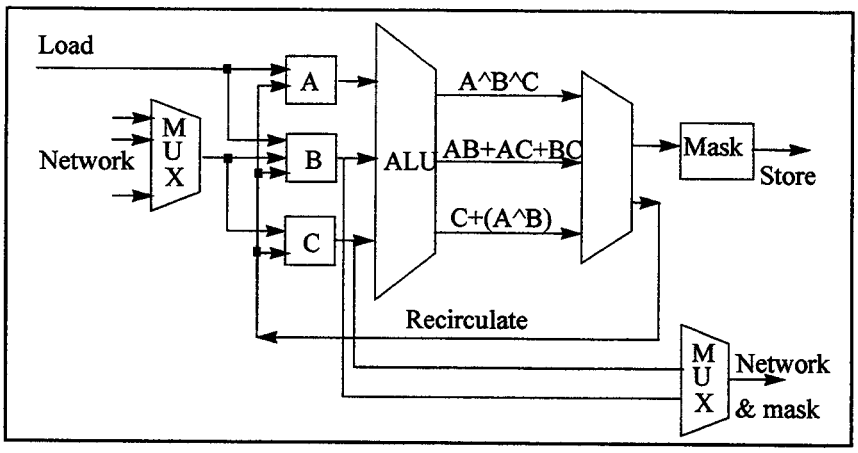


Figure 3.5 A PIM Processor

Compared to C*RAM92, this chip has an elaborate communication network with a 3-output ALU. Indirect addressing mode is also implemented.

3.5 CNAPS [Hammerstrom96]

CNAPS is a 1-D SIMD processor array developed by Adaptive Solutions Inc. Several different processor arrays have been designed: CNAPS-1064 and CNAPS-1016, corresponding to 64-, 16-PE (or processor node - PN) on the 0.8μm SRAM technology. The CNAPS-2032 (32 PEs) was later designed on a 0.6μm SRAM.

As shown in Fig. 3.6, the PEs are connected in a 1-D array with global broadcast from the sequencer to all the PEs (InBus), and a global output bus (OutBus) from all the PEs back to the sequencer. Each PE is connected to its nearest neighbour via a 1-D Inter-PE bus. The architecture offers several arbitration techniques for the OutBus, the most common being sequential transmission. In addition to data transfer, the inter-PE bus is also used to signal the next PE during

sequential arbitration. If multiple PEs transmit at once, the array performs a global AND of all the PE outputs (the default transmission is all 1's). The OutBus goes into the sequencer, which can wrap the data back around to the InBus and/or write it to memory via a DMA channel (StdOut). The sequencer likewise can read data from the memory (StdIn) and put it on the InBus for broadcast to the PE array. The PNCMD bus is used to broadcast instructions (or commands) to all the PEs.

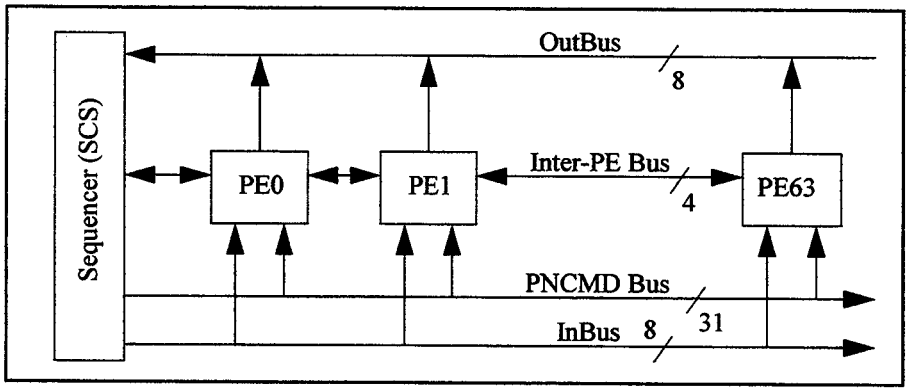


Figure 3.6 CNAPS Architecture

Similar to DSP chips, each PE on the CNAPS is a 16-b computing engine, having the following internal components: 9-b x 16-b multiplier; 32-b adder; 32 element register file (16-b); 16-b shifter/logic operation unit; 12-bit memory address adder; and 4KB of SRAM, accessible as either 8-b or 16-b values. When running at 25MHz, all 64 PEs of the CNAPS-1064 can achieve a peak performance of 3.2 GIPS, while the CNAPS-2032 can achieve a 2.56 GIPS maximum performance at 40MHz.

3.6 EXECUBE [Sunaga96]

EXECUBE is an 8 PE SIMD/MIMD chip where 100K gate custom logic circuits were integrated onto a 4.5Mb DRAM. It has independent address inputs, data I/O ports, access control circuits, and redundancy fuses and elements.

Each of the 8 PEs consists of 64KB memory, a 16-b ALU, and Direct Memory Access (DMA) interconnect circuits. The PEs are arranged in a 3-D hypercube configuration. Each PE is connected to other three PEs by inter-PE buses. In addition, it also has a separate external port which enables a direct connection to similar links on other chips in almost any topology. A broadcast logic circuit in the center of the chip works as an interface to an off-chip common control device. Inside the chip, it contains a broadcast interface which consists of common and separate buses to PEs. A 16-b multiplexed address/data bus is common to all PEs.

EXECUBE can be operated in either MIMD or SIMD mode. In MIMD mode, each of the 8 PEs fetches instructions and data from its own memory. The external common control device can send instructions into an arbitrary subset of the 8 PEs through the broadcast logic and interface. Each PE has a 16-b data read back output to the broadcast logic. The PE read back data can be shifted out of the chip at a rate of two bits per clock cycle. When running at 25MHz, each chip delivers a performance of 50 MIPS. The design goals of this chip was aimed at the high performance digital signal processing (DSP) applications. The 8 PEs are used to address the small number of taps/ inputs in the early Fast Fourier Transform (FFT) algorithms.

3.7 Parallel Image Processing RAM (PIP-RAM) [Aimoto96]

The PIP-RAM, a successor of IMAP, consists of 128 PEs and 128 DRAM elements (with 4 redundant PE-DRAM pairs), a main word decoder (MWD), clock buffers, and control circuits (Fig.3.7). Each 128Kb DRAM element is assigned to one PE. Each PE has a 8-b ALU, a shifter, 24 general purpose registers, 5 special purpose registers, and a DRAM element of 128Kb.

Since all the 128 PEs operate in a SIMD fashion at 30MHz, and read/write 8-b data from/to 128 DRAM elements at the same time, the PIP-RAM achieves 7.68 GIPS peak processing performance and 3.84GB/s peak memory bandwidth.

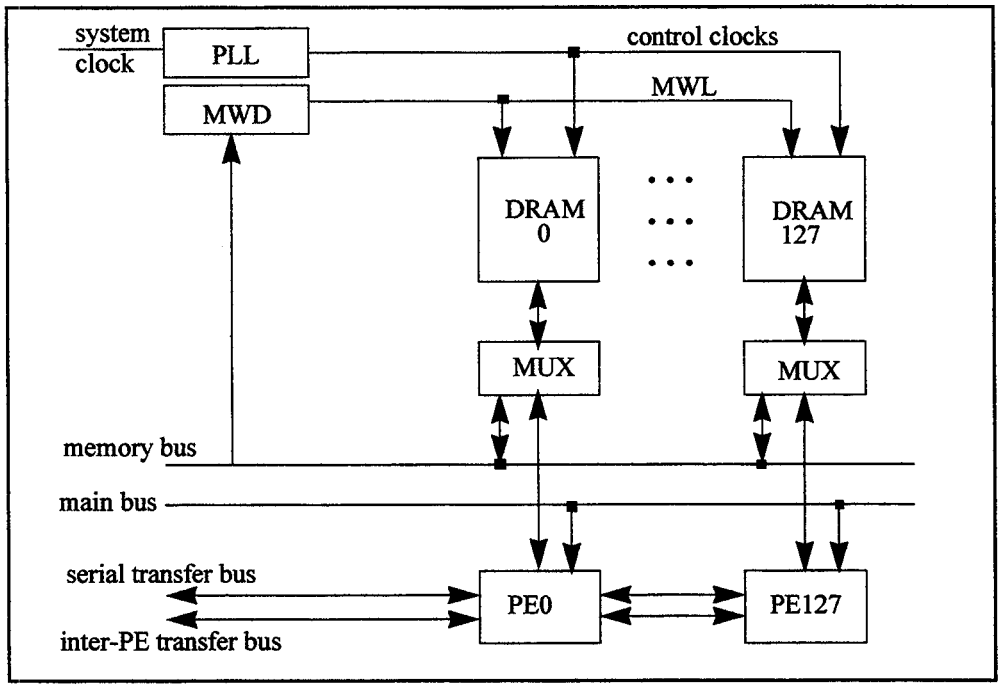


Figure 3.7 PIP-RAM Architecture

3.8 CAPP/HDPP [Gealow96]

These designs are based on large processor-per-pixel arrays implemented using VLSI technology. Two integrated circuit architectures are summarized: a content-addressable parallel processor (CAPP) and a high density parallel processor (HDPP) using DRAM cells. The former approach minimizes the amount of logic required in each processing element, while the latter minimizes memory cell size. In both architectures, the layout of the one-bit logic is pitch-matched to the memory cells to form high density 2-D PE arrays. Special 2-D routings result in an additional of 15% in logic area compared to the 1-D arrangment.

The system design features an efficient control path implementation, providing high processing element array utilization without demanding complex controller hardware. Sequences of array instructions are generated by a host computer before processing begins, then stored in a simple controller. Once processing begins, the host computer initiates stored sequences to perform pixel-parallel operations.

In the CAPP, each PE performs operations using ternary value composed of the three digits: 1, 0, and X (Fig. 3.8). The memory word includes 64 three-state memory cells. The match operation finds processing elements that match a pattern, with unused fields marked with don't care (X).

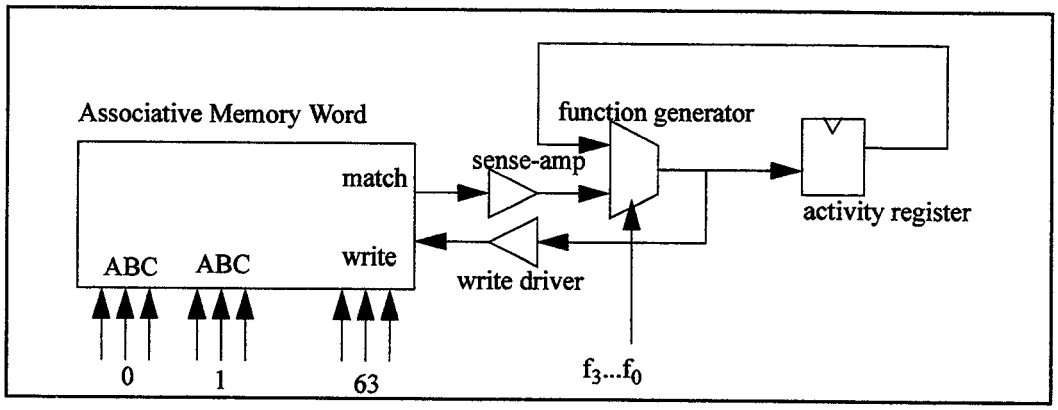


Figure 3.8 Associative PE of the CAPP

Using special cells included in the associative memory words, each PE can directly pass data to its four nearest neighbors (to the North, South, East, and West). Data movements can be combined with arithmetic operations.

In the HDPP (Fig. 3.9), each of the 256 PEs is integrated with 128b DRAM column in place of DRAM column decode logic. There is a similarity between PEs of C*RAM92 and HDPP. Three registers A, B, and C provide inputs to a function generator. Control signals $f_7...f_0$ may specify any of the 256 three-input boolean functions. Register A also provides inputs to neighboring PEs. Values from PE to the North, South, East, and West are combined with function generator results according to control signals f_N, f_S, f_E, and f_W, respectively. Final results may be loaded into registers B, C, D, and/or E. Value stored in E register is used to control the write driver.

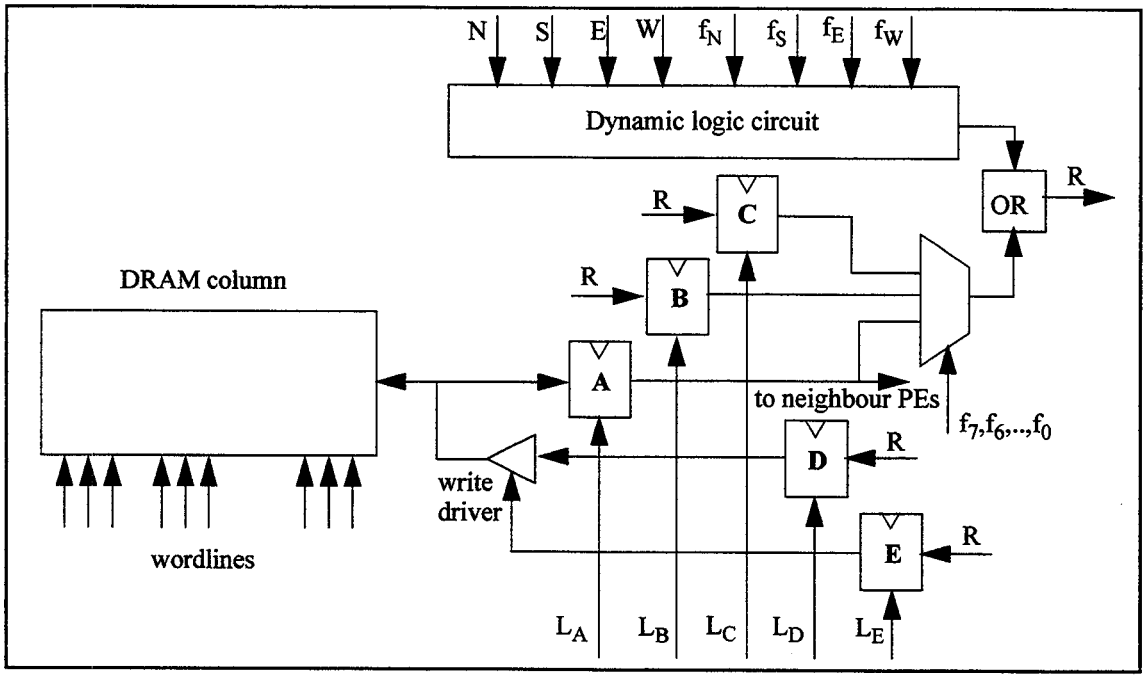


Figure 3.9 The PE of the HDPP

The PE instructions specify a memory operation, a Boolean function ($f_7...f_0$), and network function ($f_N...f_W$), and register load signals (L_A, L_B, L_C, L_D, and L_E). The execution of an instruction begins with the initiation of the memory operation, and the read or write value may be loaded into register A. Then the logic operation identified by the specified Boolean and network function is performed. Finally, the result of the logic operation may be stored into registers B, C, D, and/or E.

3.9 Summary of Features

In general, the PEs in the array processors are designed to fit the most foreseeable vectorized applications. Designs reviewed in this chapter tend to diverge in applications. Therefore, in order to systematically classify SIMD-based designs, there should be a set of parameters that are used as guidelines for the comparison. According to [Kwang93], the following SIMD model[1] is used:

1. [Hwang93] uses M on the left side instead of S, which can be confused with M as the set of masking schemes.

$$S = \langle N, C, I, M, R \rangle \tag{3.1}$$

where N is the number of PEs in a machine;

C: set of instructions directly executed by the control unit, including scalar and program flow control instructions;

I: set of instructions broadcast by the control unit to all PEs for parallel execution, including arithmetic, logic, data routing, and masking; On the other hand, I can also be looked at as the operand bits in the PE;

M: set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets; and

R: set of data routing functions or network configuration.

Since parameters C and M are not clearly defined in all designs, they are excluded in the summary. In addition to N, I, and R, the amount of memory per PE and the average performance (the number of instructions executed per second) of each design are of interest and therefore, they are listed. The features of the reviewed logic-in-memory SIMD array processors are summarized in the Table 3.1.

Table 3.1 Feature Summary of logic-in-memory SIMD designs

Name	N = No. of PEs	I =datapath	Memory/PE	R= Network	Performance (clk)
C*RAM92	64	1	128b SRAM	1-D or linear	---
C*RAM93	64	1	1Kb SRAM	linear	3.2 GIPS (25MHz)
IMAP	64	8	32Kb SRAM	linear	3.84 GIPS (40MHz)
PC-RAM	512	1 or 2n	480b SRAM	segment'ed linear	25.6 GIPS (25MHz)
SRC-PIM	64	1	2Kb SRAM	program'ed linear	0.64 GIPS (10MHz)
CNAPS 64,16	64(16), 16(4)	16	4KB SRAM	linear	3.2 GIPS (25MHz)
CNAPS 32	32(?)	16	4KB SRAM	linear	2.56 GIPS (40MHz)
EXECUBE	8	16	64KB DRAM	3-D hypercube	50 MIPS (25MHz)
PIP-RAM	128(4)	8	128Kb SRAM	linear	7.68 GIPS (30MHz)
HDPP	256	1	128b DRAM	2-D mesh	2.56 GIPS (10MHz)
C*RAM96	4096	1	4Kb DRAM	linear	80 GIPS (20MHz)
C*RAM97	1024(?)	1	16Kb DRAM	linear	---
IGA-FB98	4096 (256)	1	3.2Kb DRAM	linear	264 MIPS (66MHz)

First of all, the number of PEs serves as the degree of parallelism in the designs. The larger the number, the higher the possibility of executing operations in parallel. There is a general increase in the number of PEs as we traverse from the past to present. The numbers in brackets in the first column represent redundant PEs for yield improvement and fault tolerance. The question marks in brackets indicate the reported, but unknown, number of redundant PEs. The bit-slice architectures are dense enough to accommodate more PEs per chip compared to the word-wide architectures.

On one end of the spectrum is the single-bit PE while the other end is the 16-b PE. The leanest PE design is C*RAM92, while the most complex is CNAPS. Popular bit-serial PE designs involve three primary registers feeding inputs to a multiplexor-like ALU. While IMAP and PIP-MAP of NEC specifically address 8-b data for image processing applications, the multiple-bit PE designs such as CNAPS and EXECUBE focus on DSP applications.

On the memory side, designs of 1Kb or less are supposed to be prototypes, thus do not have any realizable implementation. Designs of 2Kb to 16Kb are suitable for single-bit PEs, while larger memories are suitable for multiple-bit PEs. Unlike the rest of the array processors, both C*RAM (including PC-RAM) and SRC-PIM designs insist on retaining the conventional memory functions which are very attractive to PC users.

The interconnection networks are mostly single-bit linear architectures. Linear networks are rigid in mappings of applications. The SRC-PIM, however, has an interesting PPN which can be programmed to one of the 16 network configurations. While most networks are linear, in some special cases, 2-D (HDPP) and 3-D (EXECUBE) networks, but not of higher dimensions, are observed due to the limitations in circuit designs and layouts. In terms of routing efficiency and scalability, linear networks retain the compactness of the existing memory modules while allowing for expansion through scalability.

Direct addressing mode is common in all designs. The IMAP and SRC-PIM, however, are the only designs with index-addressing for histogram computation and Hough transform. And finally, the most flexible design is the programmable dual mode PC-RAM where bit-serial and bit-parallel operations can be performed.

CHAPTER 4

Proposed Enhancements to C*RAM and Its Instruction Set

In this chapter, contributions to C*RAM architecture and programming framework will be presented. In Section 4.1, parallel addition/subtraction circuits which speed-up MAE computation will be proposed. Programmable segment bus-tie will also be presented and utilized to facilitate block operations and the one-to-many communication networks. In Section 4.2, VHDL models of C*RAM will be presented: a baseline and an enhanced versions. C*RAM programming framework will be described in Section 4.3 in two phases: the vectorization of image and video processing algorithms, and the conversion of such algorithms into C*RAM assembly instructions and subsequently, machine codes. In Section 4.4, the underlying C*RAM assembler, which makes all the implementations and performance analysis realizable, will be introduced. An example of an user-defined operation will also be demonstrated. C*RAM macros will be presented in Section 4.5. In addition to memory and arithmetic operations for baseline C*RAM, group parallel operations on the enhanced C*RAM will also be discussed.

4.1 Contributions to C*RAM Architecture

The C*RAM project has gone through many designs and enhancements. In this section, enhancements targeting image and video processing/compression algorithms will be proposed. For image and video applications, the following requirements are specified [Kodura98].

- The system should allow frequent memory access to a large memory space;
- Variable word-length operations are desirable to improve hardware efficiency, such as 8-b video and 20-b audio with increments of 1-b. Conventional processors or DSP's have large word-lengths of constant widths such as 16-b, 32-b, and 64-b; and
- Fast arithmetic operations such as MAE computations for ME and VQ should be supported.

By the term logic-in-memory, the reviewed bit-serial SIMD architectures have specifically been designed to address the first requirements. Processor - memory bandwidth gap is no longer an issue. The second requirement has not been addressed in constant word- length processors such as the IMAP, PIP-RAM, CNAPS, and EXECUBE.

The third requirement, fast operations for frequently used arithmetic operations, such as MAE, have not been addressed or implemented in many of the designs. In the following section, a parallel addition/subtraction circuit will be proposed to reduce MAE distortion computation time in C*RAM.

4.1.1 Parallel Addition / Subtraction Circuits

In DSP circuits, the MAC (multiply-accumulate) operation is common to every filter operation. In image and video processing, the MAE distortion measure is often employed and is usually computed by the following three operations: subtracting the candidate pixel from its corresponding reference pixel, applying the absolute operation on the difference, and accumulating the absolute differences. The absolute operation has the same complexity as an

addition since 2's complementing a number is performed by 1's complementing the number, and incrementing the result by 1. For b-bit addition/subtraction and ($b+\log_2 N$)-bit accumulation[1], the bit-serial complexity for each MAE computation is as follows:

- 3*b cycles for subtraction (one cycle to load first operand bit, one cycle to generate the SUM bit, and another to generate the CARRY bit);
- 3*b cycles for absolute operation; and
- $3*(b+\log_2 N)$ cycles for accumulation.

Therefore, a total of $9*b + 3*\log_2 N$ cycles is required for the baseline PE. This estimation does not include extra cycles for setting up and clearing the registers.

It is noted that addition and subtraction may use the same hardware, and only one bit can be used to indicate whether an addition or a subtraction is desired. A circuit which is capable of inverting one of the operands, depending on its sign, would allow addition and subtraction to be performed in parallel.

The MAE complexity implemented in the enhanced PE is as follows:

- 3*b cycles for subtraction (similar to before);
- $3*(b+\log_2 N)$ cycles for accumulation.

With the addition of new circuits, the total number of cycles has been reduced to $6*b + 3*\log_2 N$. This reduction is very significant especially with the intense MAE computation in the ME algorithm. The detailed hardware implementation will be presented in subsequent paragraphs.

Recall that the baseline PE of early C*RAMs has a 3-input ALU which operates like a multiplexor. The three inputs are one memory bit M, and the register values X and Y. For

1. When accumulating N b-bit words, the final word length is $b+\log_2(N)$ bits. For VQ, the distortion sum is $8+\log_2 16 = 12$ bits long; for ME, the distortion sum is $8+\log_2 256 = 16$ bits long.

addition, one of the register stores the CARRY bit, the other the first operand bit, and the memory bit M the second operand bit.

A sign register S along with an XOR gate can be added to keep or to invert the memory bit M (Fig. 4.1). When the sign bit is '0', the M bit is kept unchanged. When the sign bit is '1', the M bit is inverted by the XOR gate. The implementation of the parallel addition/subtraction circuits requires 2 additional control signals: S_i (not shown) allows the sign bit of the second operand (via ALU) to be stored in S register; the second signal, S_e is used to enable or disable the passing of the sign bit. The content of S is gated using a dynamic AND circuit which normally outputs a '0'.

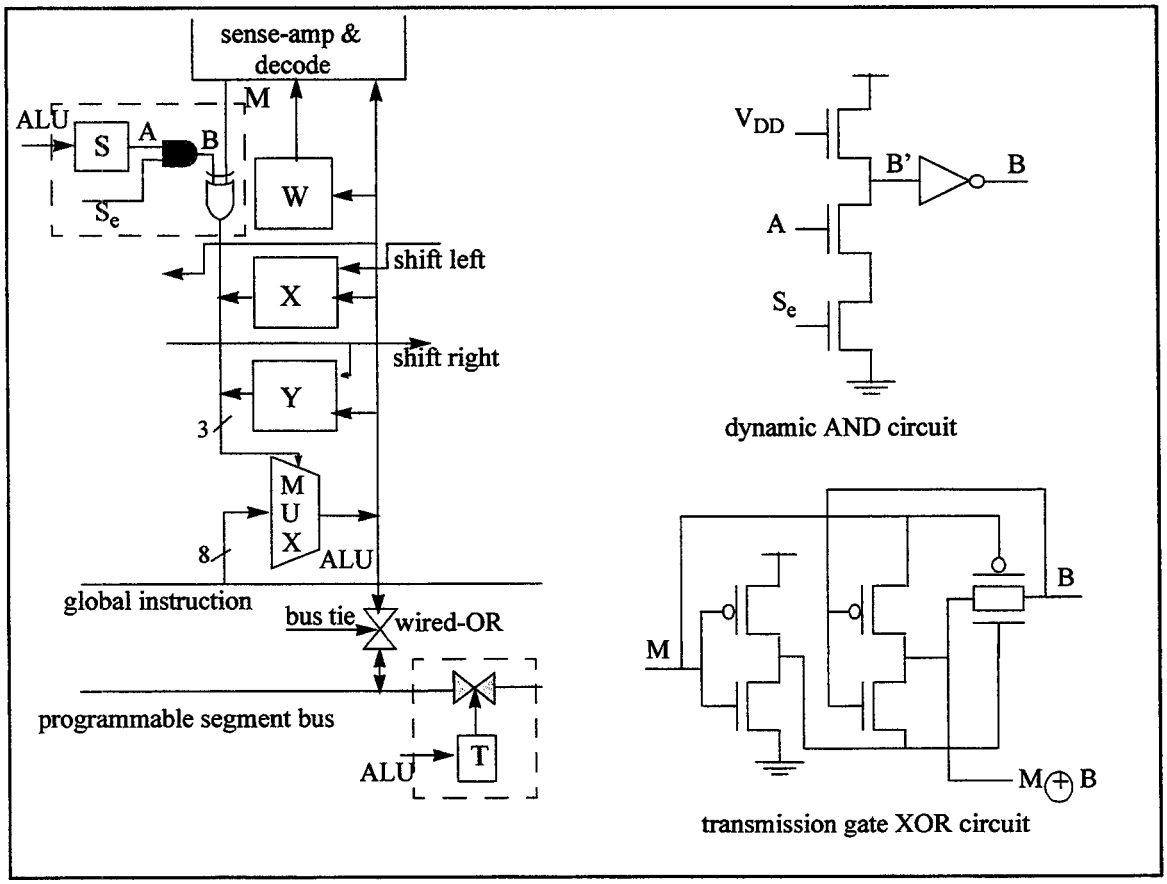


Figure 4.1 Enhanced features added to the baseline PE

The S register can be a 5-transistor latch detailed in [Cojocaru93]. In the dynamic AND circuit, node B' is, under the normal condition, precharged to V_{DD} and the output B is always '0'. When

parallel operation is intended, S_i will be set and the sign of the difference resulting from the previous subtraction will be stored in the S register. In the following cycle, S_i is reset while S_e will be set. Along with the presence of S_e and A, output B will be presented at the input to the XOR gate. A '1' at B will invert the value of M at the output of the XOR gate, and keep it otherwise. The XOR gate can be implemented using transmission gate technique as shown. The total number of transistors per PE has now increased from 76 [Elliott92] to 92[1].

4.1.2 Programmable Segment Bus-tie (PSB)

The PSB was not part of the baseline PE in C*RAM92 [Elliott92]. On the PC-RAM [Cojocaru93], a T-register (tie) and a transmission gate have been placed at every other PE (bottom of Fig. 4.1). The PSB is intended for word-parallel operations where the boundaries of the segmented buses define the n-bit computing units. This realization helps eliminate the need for a data transposer and enables multiple-bit operations. In the following paragraphs, the usefulness of the PSB is extended to block operations and to the one-to-many communication networks.

Block operations: For FFT-based operations, it is sometimes desirable to partition the PE array into groups of n PEs. For DCT computation, each 8 x 8 block may be stored either as a 8 x 8 array or 1 x 64 vector in C*RAM. The implementation efficiency will be discussed in Section 6.2.1. This section demonstrates the possibility of implementation in both configurations.

When using the 1 x 64 vector, each block is stored in one PE. Therefore, left-right communication is sufficient. When using the 8 x 8 array, each block is stored in 8 PEs in a parallel fashion. The PE array needs to be partitioned into groups in such a way that their local operations do not interfere each other.

1. 5-T for the S register, 5-T for the dynamic AND gate, and 6-T the transmission gate XOR.

For ME, the distortions of the blocks are first computed in parallel. The distortions corresponding to each block are then minimum-searched. One search strategy is to compare a block's distortion with its neighbors and keep track of the search boundary. Another is to use a bus-tie circuit with a mask to perform parallel search within the block search boundary. The latter strategy can be improved by segmenting the bus at the search boundary prior to performing the search procedure.

One-to-Many Communication Networks: The application scope of C*RAM can be enlarged by allowing a C*RAM to handle irregular data transfers, while retaining parallel computation. If the PE array is partitioned into groups A, B, C, etc., irregular data transfer within each group may be realized. For instance, the content of PE[A_1] may be transferred to PE[A_3] and PE[A_5] within group A, while PE[B_7] may be transferred to only one PE[B_4] within group B in the opposite direction. This can be programmed by setting the source mask using PE[A_1] and PE[B_7], destination mask using PE[A3], PE[A5], and PE[B4], and the by-8 segmented mask defined by the group boundaries. This technique, to be discussed in Section 6.2.4, is used to reorder the minima in different rows for parallel search. In general, the source nodes in groups of PEs are not necessarily the same, and neither are the destination nodes. One condition for data transfer is that the row address i of the source bits has to be the same for all groups of PEs. This must also be true for the row address (i + k) of the destination bits.

Within a range determined by the T registers, the contents (can be 1-b to 12-b datum) of any PE can be transferred to any other PEs with a maximum delay of t ns per bit, where t is the propagation delay between the two end PEs. Fig. 4.2 shows an example where the C*RAM is partitioned into multiple Hypercubes, where in each 8-node cube is formed by 8 consecutive PEs.

If a 5-transistor T register and a 2-transistor transmission gate are repeated at every 4^{th} PE, the average transistor count will be 94.

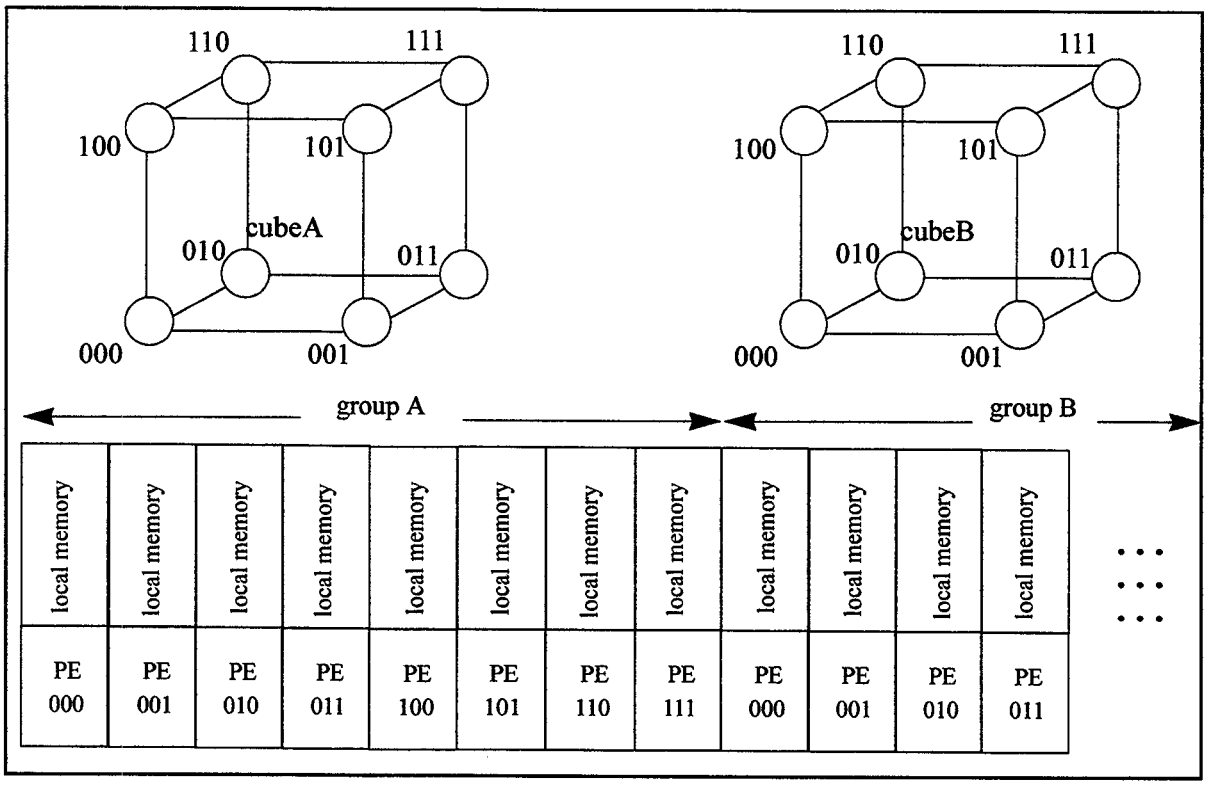


Figure 4.2 Multiple Hypercube, realized by enhanced PE with PSB.

4.2 VHDL Models

In order to analyze C*RAM functionalities on the image and video processing algorithms, two Very-high-speed-integrated-circuits Hardware Description Language (VHDL) models of C*RAM have been built (Section B.1). Three main modules are housed within the test bench: a Clock Generator module, a Address Decoder module, and a RAM module (Fig. 4.3).

The RAM module is composed of a RAM portion and the PE array. For the baseline model, no circuit enhancement is realized. In the case of the enhanced model, a sign-regulate circuit is placed at every PE, while a PSB circuit is placed at every 4^{th} PE. The number 4 has been selected since it is the smallest number of a good block size in both VQ and ME operations.

The memory of the baseline model remains unchanged at 1Kb x 64PEs for historical reasons. For applications which require moderate amount of memory, 1Kb is adequate. In fact, 64-word VQ

has been implemented using the baseline model [Le94, Le96]. For DCT and ME, additional memory is required to store long intermediate results and massive data for interframe prediction. Therefore, local memory has been increased to 4Kb per PE. The number of PEs remains unchanged.

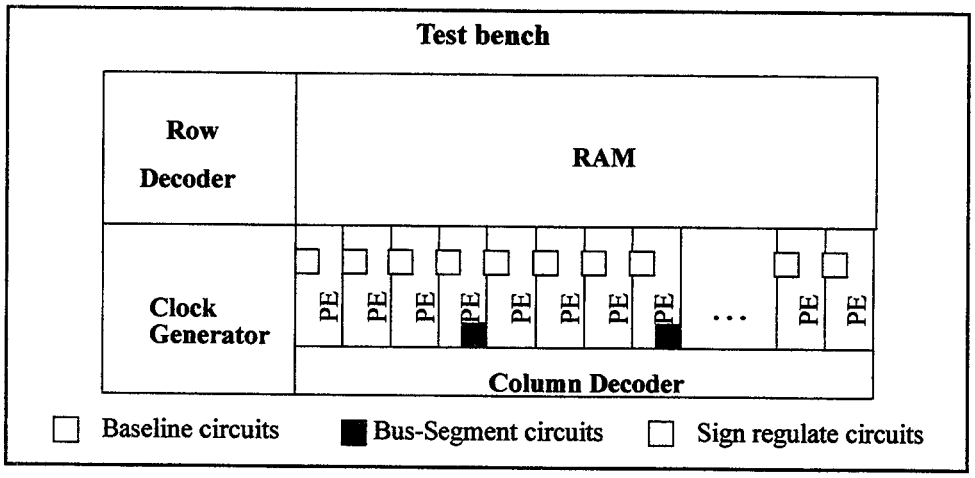


Figure 4.3 VHDL model for C*RAM with image/video-enhanced circuits

The 8-b truth table op-codes (TTOP) have been multiplexed with the data bus. No enhancement is made to the TTOP bits. On the other hand, the control op-code (COP) bits have been multiplexed with the address bits. The COP bits are allocated as follows:

$$COP<5:0> = \{BT,RY, XL, Y, X, WE\}$$

The signals are, from left to right: bus-tie for global tie, writing the ALU output to the Y-register of the right-neighbor PE, writing to the X-register of the left-neighbor PE, writing to own Y-register, writing to own X-register, and writing to own WE register. The remaining address lines are not used.

On the enhanced PE, additional address lines are used for the following control signals: S_i=COP<6> for writing to S register, S_e=COP<7> for enabling the sign-regulate circuit, and T_i = COP<8> for writing to T register.

4.3 Programming Framework

The programming framework has been developed using C/C++. C programming language is used because it supports a wide range of applications from numerical analysis to image processing. C++, an extension of C, provides additional facilities for defining class constructor, overload functionality, and inheritance.

From an implementation point of view, two phases have been carried out (Fig. 4.4). Phase one involves the vectorization of the existing image and video processing/compression algorithms, and phase two is to convert such algorithms into C*RAM assembly instructions and subsequently, machine codes (addresses, TTOP, and COP bits) for C*RAM operations.

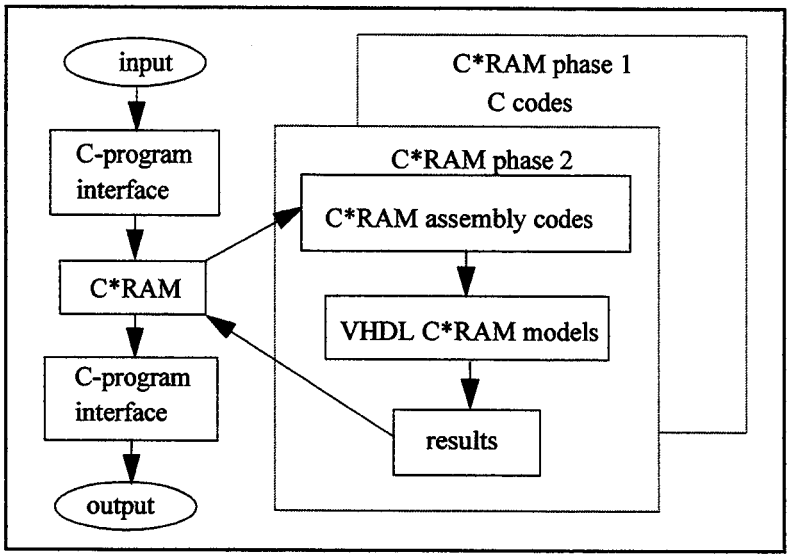


Figure 4.4 phase 1: C-program model; phase 2: C*RAM VHDL model

Phase one has been implemented using C codes. The performance of C*RAM is modeled by counting the number of C*RAM cycles required by each algorithm. This performance is compared against that of a conventional software implementation using a RISC processor, the Sun SuperSPARC system having 96MB of RAM and running at 50 MHz. Phase two has been implemented with actual C*RAM functions in assembly language. These instructions are subsequently translated into input stimuli for actual C*RAM operations. This translation process

emulates a controller for the study of C*RAM's functionalities for different image and video processing/compression algorithms. The input stimuli are stored as ASCII vectors in various text files which will be fed to the VHDL models for simulations.

Finally, all results are collected and interpreted using C for statistical and visual analysis.

4.4 C*RAM Assembler

4.4.1 Overview of C*RAM Cycles

In addition to TTOP and COP, other signals include: MCK - master clock strobe, CCK - C*RAM clock strobe, OPS - operation strobe, RD_WRB - read and write strobe, and R_CB - memory and compute signal. Unless otherwise stated, the baseline model is assumed.

There are two modes: memory mode and compute mode which are determined by the R_CB signal. In the memory mode, R_CB is set. For a memory Write, data is placed on the data bus and address is placed on the address bus before the rising-edge of MCK. For a memory Read, an address is placed on the address bus, and data are read out, also at the MCK rising-edge.

In the compute mode: R_CB signal is reset. The possible types of cycles are: Read, Read-Operate, Read-Operate-Write, Operate-Write, and Write. They are summarized below:

- The Read cycle (R) simply reads a memory bit M at the MCK strobe and leaves its value at the M input to the ALU;
- The Read-Operate cycle (R-O) reads a memory bit at the MCK strobe, combines it with possible values of either or both X and Y register values to produce a result at the ALU output following the OPS strobe. Write backs to registers are made possible using the immediately followed CCK rising edge on the COP bits;
- The Read-Operate-Write cycle (R-O-W) is similar to R-O cycle, except that the resulting bit is written back to the memory using the immediately followed MCK strobe; Fig. 4.5 shows an

example of a R-O-W cycle. During the Read cycle (RD_WRB=1, after t=100ns), the first rising-edge of MCK (t=105ns) strobes row-address <101>. Next, the rising-edge of OPS (t=115ns) evaluates the TTOP=0xF0 in the databus, and the first rising-edge of CCK (t=117ns) evaluates the now-COP-value address bus for any register write or bus-tie instruction. The second rising edge of CCK (t=137ns), while OPS is low, is when the actual writing to the register(s) is performed. Finally, when RD_WRB=0 (after t=140ns), the second rising-edge of MCK (t=145ns) strobes row-address <001> for writing back to memory. The time between the two CCK rising-edges is set long enough to ensure that the signal has time to propagate during bus-tie; Note that, the R-O cycle is similar to a memory-access cycle since it only uses one MCK cycle, while a R-O-W cycle requires two memory-access cycles since it uses two MCK cycles (one for reading, the other for writing with the OPS being sandwiched in between);

- The Operate-Write (O-W) and the Operate (O) cycles can be explained similarly;

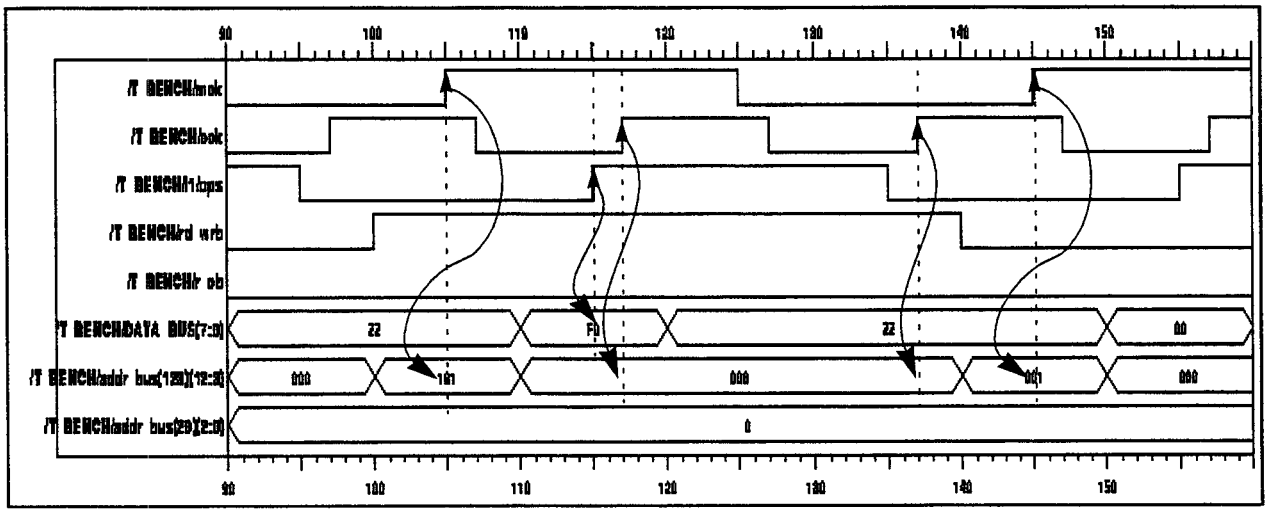


Figure 4.5 C*RAM Read-Operate-Write cycle

Since most of the cycles requires either a memory read and an operation on that operand, or an operation followed by a memory write, the cycle is, unless otherwise stated, assumed to be a Memory access followed by an Operation, called Memory-Operate cycle. Moreover, the limited

number of registers has forced the temporary result to be written back to a memory location and re-read in the following cycle.

Since a Memory-Operate cycle has the same width as a RAM cycle, it can, therefore, be scaled as the memory technology advances. Scalability of C*RAM cycles enable C*RAM programming to be kept in pace with the RAM technology. Test files for VHDL models are provided in Section C.2.

4.4.2 Data Format

C*RAM operations have been tailored for variable-length operands. For instance, the addition of two 8-b operands yields a 9-b result. In order to perform sign operations, the shorter operand must be sign-extended to the required length for proper alignment with the longer operand. Unless otherwise stated, pixels in an image are assumed to be quantized to 8 bits.

- Unsigned integer: Pixel value remains unchanged. Unsigned integer is sign-extended by appending 0's at its MSB positions. Sign extension is generally not required;
- Signed integer: Pixel value is level-shifted around 0 by inverting its MSB. Signed integer is extended by appending the appropriate sign bit at its MSB position;

Both unsigned and signed integers are expressed in fixed-point formats which have an integer part and a fractional part. The binary point is not shown, yet its position is remembered.

4.4.3 Instruction Derivation for an User-Defined Operation: An Example

When performing non-standard operations, user-defined instructions are often required. The following example demonstrates how an "A > B comparison" macro is derived. Assuming that A and B are *len* bits long. The LSB's of A and B are located at rows *src_addr1* and *src_addr2,* respectively, and the result of the comparison is stored in row *dst_addr*. Also, let the current and

previous bit positions be indexed (i) and (i-1), respectively. A truth-table of such an operation is summarized in Fig. 4.6.

A_i	B_i	G_{i-1}	G_i
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1
GRE =			0xBA

Figure 4.6 Derivation of GRE instruction

A is greater than B when:

- previously determined by the higher order bits, G(i-1);
- or currently evaluated by [A(i)=1 AND B(i)=0];

Therefore, instruction GRE(M,Y,X) is 0xBA in hexa-decimal representation. If G_i is assigned to X register, B_i to Y register, and A_i to the input M to the ALU, the macro CMPg(len, src_addr1, src_addr2, dst_addr) is derived as follows:

```
RST(X); /* initially, assuming A is not greater than B */
for i = len-1 downto 0
    Y = LD(B_src_addr1+i); /* load B_src_addr1+i to Y */
    X = GRE(M[A_src_addr2+i],Y,X); /* perform GRE and store result in X */
end;
STO(dst_addr,X); /* store the final result to dst_addr */
```

The number of Memory-Operate cycle is 2**len*+2.

Note that, there are cases where not all PEs participate in the same instruction. The macro, however, must remain generic to all PEs. Non-participating PEs may be masked out using their WE registers. This procedure can be used to develop many user-defined instructions and macros tailored to image and video processing.

4.4.4 C*RAM Instructions

There are two main sections: instructions for baseline PE and instructions for enhanced PE. Baseline operations include memory functions: external read (XDR) and write (XWR), and C*RAM compute functions. The C*RAM instructions can be classified into 2 categories: load operations (LD), logic operations (XOR, AND, OR, INV, SET, RST, etc.). In addition to standard instructions, 20 new user-defined C*RAM instructions have been developed. Among them, comparison (EQU, GRE, and LES), sign extension (SXT), sign reversal (SRV), and multiplexing (M1YM0X and Y1MY0X) instructions appear to be the most intuitive. For instance, instruction M1YM0X selects Y for output when M is '1', and selects X for output when M is '0'. Instruction Y1MY0X is defined similarly. C*RAM instructions are listed in Table 4.1.

Table 4.1 C*RAM standard and user-defined instructions

Instruction	TTOP	Notes
NO-OP	0x00	no operation
SET	0xFF	set registers
RST	0x00	reset registers
LD(M)	0xF0	load M
LD(~M)	0x0F	load ~M
LD(Y)	0xCC	load Y
LD(~Y)	0x33	load ~Y
LD(X)	0xAA	load X
LD(~X)	0x55	load ~X
XOR(M,Y,X)	0x96	generate SUM
CAR(M,Y,X)	0xE8	generate CARRY
XOR(~M,Y,X)	0x69	generate DIFFERENCE
BOR(~M,Y,X)	0x8E	generate BORROW

Table 4.1 C*RAM standard and user-defined instructions

Instruction	TTOP	Notes
XOR(~M,-,X)	0xA5	special instruction
AND(~M,-,X)	0x0A	special instruction
XOR(M,-,X)	0x5A	special instruction
AND(M,-,X)	0xA0	special instruction
AND(M,Y,-)	0xC0	special instruction
M1YM0X(M,Y,X)	0xCA	ALU = (M==1)?Y:X
Y1MY0X(M,Y,X)	0xE2	ALU = (Y==1)?M:X
EQU(M,Y,X)	0x82	M == Y
GRE(M,Y,X)	0xBA	M > Y
LES(M,Y,X)	0xAE	M < Y
SRV(M,Y,X)	0x96	ALU=(XOR(Y,X)==0)?M:~M; sign reversal
SXT(M,Y,X)	0xD4	sign extension for ADD, carry is in X
SXT2(M,Y,X)	0xB2	sign extension for ADD, carry is in Y
XOR(~M,Y,X)	0xB4	ALU=(Y==1)?XOR(~M,-,X):M;
CAR(~M,Y,X)	0x08	ALU=(Y==1)?AND(~M,-,X):0;

The last two entries in Table 4.1 show some complex instructions. In instruction results in 0xB4, the ALU output, based on the value of Y, is a function of XOR(~M,-,X) or M.

C*RAM operations can be performed using up to three inputs: M, X, and Y. The number of memory accesses and instructions can be reduced by maximizing the number of available inputs to the ALU.

4.5 C*RAM Macros

C*RAM macros are mainly unsigned and signed variable-length arithmetic operations. The signed operations are longer due to signed extension at the MSB. All C*RAM functions have left-shift and right-shift capability where operands on neighboring PEs can be operated on. C*RAM macros are listed in Table 4.2.

Table 4.2 C*RAM Macros

Operations	Instruction	No. of cycles	Notes
Memory	CLR, AST	b + 1	clear/assert b-bit word
	CLRm, ASTm	b + 3	clear/assert with a mask
Shifting	ASHR, ASHL	b*(2-pos) + pos	pos = no. of shiftings
	MOV	2*b	move to a different location
Comparison	CMPe, CMPg, CMPl	2*b + 2	equal, greater than, less than
	CMPe_R,CMPg_R	2*b + 2	comparison with shiftings
	CMPl_R	2*b + 2	less than comparison w. Rt.
	MIN, MAX	3*b + 2	min/max search
Unsigned	ADDu, SUBu	3*b + 2	unsigned add and subtract
	ADDu_L, ADDu_R	b*(2+pos) + 2	add to the left/right PE
	SUBu_L, SUBu_R	b*(2+pos) + 2	subtract fr. the left/right PE
	ACCu	3*S + 1	S = final sum length
	MULu	Lc*(3*b + 2)	Lc = length of the multiplier
Signed	ABS	2*b + 2	absolute difference
	ADD, SUB	3*b + 3	signed add/sub
	MUL	Lc*(1+3*b+3)+33+28+2	signed multiplication
Enhanced	PTRX	2*b + 6	parallel trans within a group
	PAS	3*b + 2	parallel add/subtract
	PMIN, PMAX	3*b + 3	parallel min/max. search

The absolute operation (ABS) macro is worth mentioning here. Absolute operation can normally be performed by 1's complementing the operand and adding 1 to the final result. This is similar to an addition which takes on the order of 3*b cycles. The absolute operation can be reduced to 2*b cycles by using the special XOR and AND instructions.

The enhanced C*RAM is equipped with the following macros:

- PTRX: allows parallel transfer within a segmented bus from a single source to one or many destinations;
- PAS: allows parallel additions and subtractions based on a sign mask; and

- PMIN/PMAX: minimum / maximum searches within a group of PEs whose boundary is defined by the PSB.

4.5.1 Design of the Assembler to Machine Code Translator

C programs, modeling the controller's functionalities of data format and transposing have been written to generate C*RAM machine codes for different memory and compute operations. There are two switches in each program: GENERATE and DEBUG. For each algorithm with GENERATE switch turned on, this program will generate the required row and column addresses for memory operations, the TTOP patterns as inputs to the ALU, the COP patterns as instructions for register writes, bus-tie, etc., and the row address (in the compute mode) common to all the PEs local memory. These addresses, TTOP, and COP are stored in different ASCII files and used as input stimuli to the VHDL models of the C*RAM. On the other hand, when being switched to DEBUG mode, similar files can be generated for visual inspection.

If a 4-b value A starting at row address 0 is to compared with another 4-b value B starting at row address 8, and the result is stored in row address 32, the macro CMPg(4,0,8,32) can be invoked. The corresponding codes are generated as follows:

```
1. RD_addr='1111111111000'; TTOP='00000000'; COP='0000000000010';
2. RD_addr='0000000011000'; TTOP='11110000'; COP='0000000000100';
3. RD_addr='0000001011000'; TTOP='10111010'; COP='0000000000010';
4. RD_addr='0000000010000'; TTOP='11110000'; COP='0000000000100';
5. RD_addr='0000001010000'; TTOP='10111010'; COP='0000000000010';
6. RD_addr='0000000001000'; TTOP='11110000'; COP='0000000000100';
7. RD_addr='0000001001000'; TTOP='10111010'; COP='0000000000010';
8. RD_addr='0000000000000'; TTOP='11110000'; COP='0000000000100';
9. RD_addr='0000001000000'; TTOP='10111010'; COP='0000000000010';
10.WR_addr='0000010000000'; TTOP='00000000'; COP='0000000000000';
```

The above codes can be explained as follows:

- Line1 reads a value from a dummy address[1], executes RST instruction, and writes to X by setting COP<1> to 1;
- Line2 reads value A(3) from row address 3, executes LD(M) instruction, and stores the result to register Y by setting COP<2> to 1;
- Line3 reads a value B(3) from row address 11, executes GRE(M,Y,X) instruction, and stores the result to register X;
- The pairs of lines (4 and 5), (6 and 7), and (8 and 9) repeat the same executions similar to the pair of lines (2 and 3); and
- Line10 writes the final result to row 32, and no operation or register write is required.

4.6 Conclusions

In this chapter, enhanced features and their advantages in image/video processing and compression have been presented. Detailed descriptions of the C*RAM assembler and translators have also been documented.

The special requirements for image and video processing have been outlined and addressed. Firstly, the frequent access to large memory space has been facilitated by the logic-in-memory C*RAM architecture. Secondly, variable word-length operations have been made possible by the bit-serial C*RAM's ALU and its instruction set. Finally, the first and foremost parallel addition/ subtraction circuits have been proposed to enable fast arithmetic operations such as MAE computations for ME and VQ.

With an addition of 18 transistors per PE for parallel addition/subtraction, one-to-many network connections, and variable word-length operations, C*RAM performance will be significantly improved as will be demonstrated in the following chapters.

1. The 3 least significant bits are bank address, and the 10 higher significant bits are row address. Bank address is ignored in C*RAM compute mode.

CHAPTER 5

C*RAM Implementations of Image Processing Algorithms

In this chapter, image and video processing algorithms will be mapped and implemented on the 1-D C*RAM using bit-serial baseline PE [Cojocaru93]. Data partition and arrangement will first be presented in Section 5.1. In Section 5.2, the implementations of general/morphological image processing algorithms will be presented, followed by the performance comparisons with a RISC processor, and the IMAP and HDPP. DCT will be implemented in Section 5.3, followed by the presentations of stages in entropy coding, especially DPCM and run-length coder, in Section 5.4. Finally, Sub-Codebook VQ (SCVQ) implementation will be presented in Section 5.5.

5.1 Image Data Arrangement

Image data arrangement has been a matter of significant interest to image processing programmers and application engineers. Questions such as whether to leave the image data in their row-by-column order or to rearrange them in a certain way, have always arisen with first

time programmers. Issues such as data padding and black (or white) bordering around an image is very critical in bi-level morphological image processing.

Image data captured by the camera light sensors are naturally ordered in the number of pixels in a column, followed by the number of pixels in a row. For most of the operations, the pixels are left in their natural order. The 2-D SIMD architectures are among those whose implementations respect this order. When implementing on the 1-D SIMD architectures, the PE array may be more efficiently used when processing data in 2-D blocks over 1-D vectors.

In this thesis, two main configurations are considered:

- The "1 x n^2 configuration": One n x n pixel block is stored in one PE. This configuration is commonly used when performing operations of Fast Fourier Transform (FFT) nature; and
- The "n x n configuration": One column of pixels is stored in each PE, and an n x n pixel block will be accommodated by n PEs. This configuration is used when performing pixel and local operations.

Throughout the thesis, either one or both configurations will be used to study the trade-offs in execution times, data transfer times among the PEs, and memory usage. A *computing unit* (CU) is defined as a PE in the 1 x n^2 configuration, and a group of n consecutive PEs in the n x n configuration.

To the C*RAM programmer, the bit-depth, hidden from readers, is analogous to the third dimension being mapped onto the same PE as shown in Fig. 5.1. It can be seen that host words and C*RAM words are orthogonal. Data transposing can be performed via array indexing at the host, or within C*RAM hardware. It has been proven that for images of size 256 x 256 (64KB) or larger, data transposing time in C*RAM is only 0.5% of total image loading time [Nyasulu99]. Therefore, data transposing time can be considered negligible in subsequent sections.

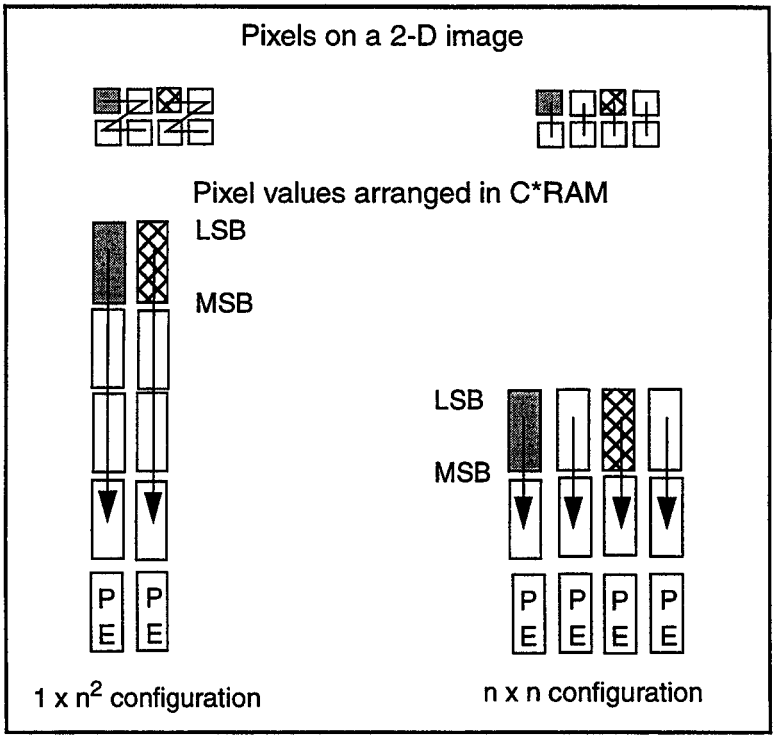


Figure 5.1 Two configurations for data arrangement in C*RAM

5.2 General/Morphological Image Processing Techniques

Let P(i,j) be the pixel at row i and column j on the input image, the corresponding values for this raw pixel and the processed pixel are I(i,j) and O(i,j), respectively[1]. When mapping onto C*RAM architecture, the first index, the memory row address, is usually implied for all C*RAM operations, while the second index, the PE address, is specified when left or right operations, such as shiftings and additions/subtractions with neighbouring pixels, are mentioned. Also, let $R_s(i,j)$ be the memory-map working registers which are used to store temporary results. Finally, let S(i,j) be the mask value stored at row(i) and PE(j), respectively; and W(j) be the content of the Writer-enable register at a certain time.

Recall that X, Y, and WE are registers, while M(i,j) is the memory bit at row(i) and PE(j). Since a b-bit pixel requires b rows of storage, the sub-pixel operations operate on a row-by-row basis. On

1. In most situations, P(i,j) and I(i,j) are the same. In JPEG, however, I(i,j) is the level_shifted value of P(i,j).

the other hand, the pixel operations operate on a pixel-by-pixel basis, indexed by the LSB of the word.

For image processing, unless otherwise stated, an image of size 256 x 256 pixels is used for analysis, and a 256-PE baseline C*RAM running at 40 ns Memory - Operate cycle is assumed. For video processing, the number of PEs will be derived based on the memory/processing requirements. The memory per PE, the number of PEs, and I/O considerations will be fully discussed in Chapter 7.

In terms of data arrangement, for sub-pixel and pixel operations, only the n x n configuration is assumed, while for DCT and ME[1], both configurations will be studied. In order to show the benefit of variable word-length operations, segments of assembly codes (previously introduced in Sections 4.4 and 4.5) will be presented.

5.2.1 Sub-pixel Operations

If each pixel is quantized to b bits, level-shifting can be performed by flipping the MSBs of the pixels as follows:

```
/* for each row in the image */
for(row=0;row<MAX_ROW;row++)
    O(b*row+b-1) = LD[~I(b*row+b-1)];
```

Bit flipping involves: reading the $\bar{b}$ value of M using the LD(~M) instruction, and storing the result. Therefore, two access-operate cycles are required per row of pixels. The frame execution time is therefore 0.020 ms.

Bit extraction: The pixels are naturally aligned and stored in their binary representations. As a result, the bit-planes can be easily extracted.

1. ME will be discussed in Chapter 6.

5.2.2 Row-wide Minimum / Maximum Search

Assuming that the image1 (I1) is stored in C*RAM starting from row address in1.

```
/* for each row in an image */
for(pixel=0;pixel<MAX_ROW;pixel++)
    O(in1 + pixel) = MIN[I1(in1 + pixel)];
```

The minimum search using the bus-tie circuit is performed one row at a time. For each row, 38 cycles are required for 12-bit search, and therefore, the frame execution time for minimum search is 0.389 ms.

5.2.3 Contrast Stretching, Clipping, and Thresholding

The contrast stretching algorithm is vectorized here: there are 3 regions of interest: [0,a], [b,L], and [a,b], where $0 \le a < b < L = 255$.

```
/* for each row in an image */
for (pixel=0;pixel<MAX_ROW;pixel++) {
    M(mask_cs,j) = '1';
    /* lower limit at a */
    WE(j) = CMPl[I(pixel,j),a] ? 1:0;
    O(pixel) = MULu[I(pixel),alpha];
    M(mask_cs,j) = '0';
    /* upper limit at b */
    WE(j) = CMPg[I(pixel,j),b] ? 1:0;
    O(pixel) = MULu[I(pixel),gamma];
    M(mask_cs,j) = '0';
    /* in between a and b */
    WE(j) = M(mask_cs,j);
    O(pixel) = MULu[I(pixel),beta];
}
```

When the fractional portion of the multiplier is significant, the precision can be preserved by multiplying by a factor of 2^q-1 and later right-shifting[1] by q bits. All multipliers in this algorithm

1. Right-shifting implies the conventional divide-by-two operation, when pixels are arranged in C*RAM, it is equivalent to moving the pointer up/down the row address.

are scaled by 8 bits.

For each row, there are two 8b-by-8b unsigned multiplications (for alpha and gamma less than 1) and one 8b-by-9b multiplication (for beta greater than 1), two 8b comparisons (using subtractions), and 3 Memory-Access cycles (for setting up). The total number of cycles taken per row is, therefore, 1058 cycles. The frame execution time is 10.834 ms.

Clipping, a special case of contrast stretching, can be obtained by making changes to the following lines:

```
O(pixel) = 0; /* set to zero for lower limit at point a */
O(pixel) = L; /* set to 255 for upper limit at point b */
```

Finally, thresholding can be obtained by setting a =b = threshold_level. A simpler implementation is as follows:

```
/* for each row in an image */
for(pixel=0;pixel<MAX_ROW;pixel++)
  O(pixel) = CMPg[I(pixel), threshold_level] ? L:0;
```

5.2.4 Image Subtraction

Assuming that the image1 (I1) and image2 (I2) are stored in C*RAM starting from row addresses in1 and in2, respectively.

```
/* for each row in an image */
for(pixel=0;pixel<MAX_ROW;pixel++) {
   R1(pixel) = SUBu[I1(in1 + pixel), I2(in2 + pixel)];
   O(pixel) = ABS[R1(pixel)];
}
```

For each row, there are one 8-b subtraction and one 9-b absolute operation. The number of cycles required per row is 47. The frame execution time for image subtraction is 0.481 ms.

A special case of image subtraction is grayscale reversal:

```
/* for each row in an image */
for (pixel=0;pixel<MAX_ROW;pixel++)
    O(pixel) = SUB[L, I(pixel)];
```

The frame execution time for grayscale reversal is 0.276 ms. (i.e. 27 cycles per row).

5.2.5 Filtering

This section includes typical filtering examples such as spatial averaging and a basic morphological dilation operation. The section will be concluded with typical edge detection algorithms.

5.2.5.1 Spatial Averaging

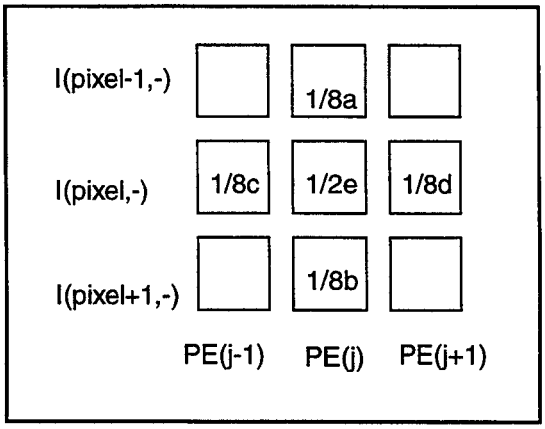


Figure 5.2 Spatial Averaging

A 3 x 3 window is used in the spatial averaging process with weights shown in Fig. 5.2. Divisions by 8 and 2, are equivalent to arithmetic right-shiftings by 3 and 1 binary positions, respectively. Therefore, the procedure is described as follows:

```
/* for each row in the image */
for(pixel=0;pixel<MAX_ROW;pixel++) {
    R1(pixel) = ADDu[I(pixel-1), I(pixel+1)];/* a + b */
    R2(pixel, j) = ADDu_L[R2(pixel, j), I(pixel, j-1)]; /* + c */
    R2(pixel, j) = ADDu_R[R2(pixel, j), I(pixel, j+1)]; /* c + d*/
```

```
    R1(pixel) = ADDu[R1(pixel), R2(pixel)];/* a + b + c+ d */
    R1(pixel) = ASHR[2,R1(pixel)];    /* 1/4(a + b + c+ d) */
    R1(pixel) = ADDu[R1(pixel), I(pixel)];/* 1/4(a + b + c+ d) + e */
    O(pixel) = ASHR[1,R1(pixel)];     /* 1/8(a + b + c+ d) + 1/2e */
}
```

Recall that ASHR(k, row_address) is the by-k arithmetic right-shift operation on the word starting at row_address. Due to the growing word sizes, only the final number of cycles are recorded. The number of cycles, for each row, are 168. The frame execution time is 1.720 ms. The gradient function for unsharp masking can be implemented in a similar fashion.

5.2.5.2 Morphological Dilation

The following SE's are allowed in the operations: square SE, plus-shaped SE, and linear SE with their origins being at the centers. Dilation operations are performed by replacing the center pixel of a n x n window by the maximum value of the containing pixels (Fig. 5.3).

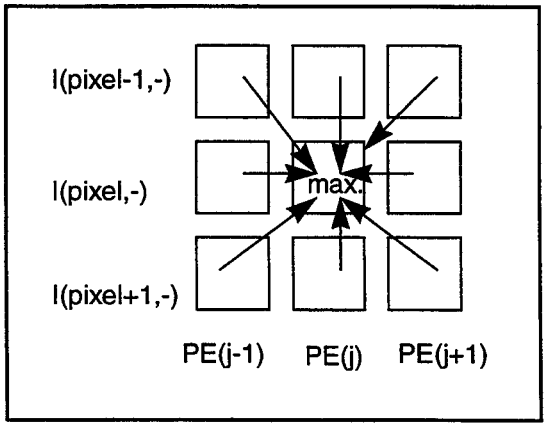


Figure 5.3 Morphological dilation

```
/* for each row of the image */
for(pixel=0;pixel<MAX_ROW;pixel++) {
    R1(pixel) = I(pixel);
    R1(pixel) = CMPl[I(pixel-1),R1(pixel)] ? R1(pixel): I(pixel-1);
    R1(pixel) = CMPl[I(pixel+1), R1(pixel)] ? R1(pixel): I(pixel+1);
    O(pixel,j) = CMPl_R[R1(pixel,j-1), R1(pixel,j)] ? R1(pixel,j):
                                        R1(pixel,j-1);
```

```
        O(pixel,j) = CMPg_R[R1(pixel,j+1), R1(pixel,j)] ? O(pixel,j):
                                                    R1(pixel,j+1);
    }
```

In this implementation, a 3 x 3 window is chosen. Similarly, a morphological erosion filter can be implemented using the minimum operation on the window. For each row, the number of required cycles is 165. The frame execution time is 1.697 ms.

5.2.5.3 Edge Detection Filters

In this section, two edge detection algorithms are presented. Focus will be placed on algorithms which use simple operations such as comparison, addition, subtraction, and arithmetic shift which are preferable for low-complexity ME algorithms. Airplane image of 256 x 256 pixels has been selected (Fig 5.4).

Figure 5.4 Original 256 x 256 Airplane image

The edge detection algorithms should be able to outline the airplane and the mountain over the white cloud and snow. Due to low complexity requirements, first order techniques such as spatial averaging and a morphological technique have been chosen for implementation.

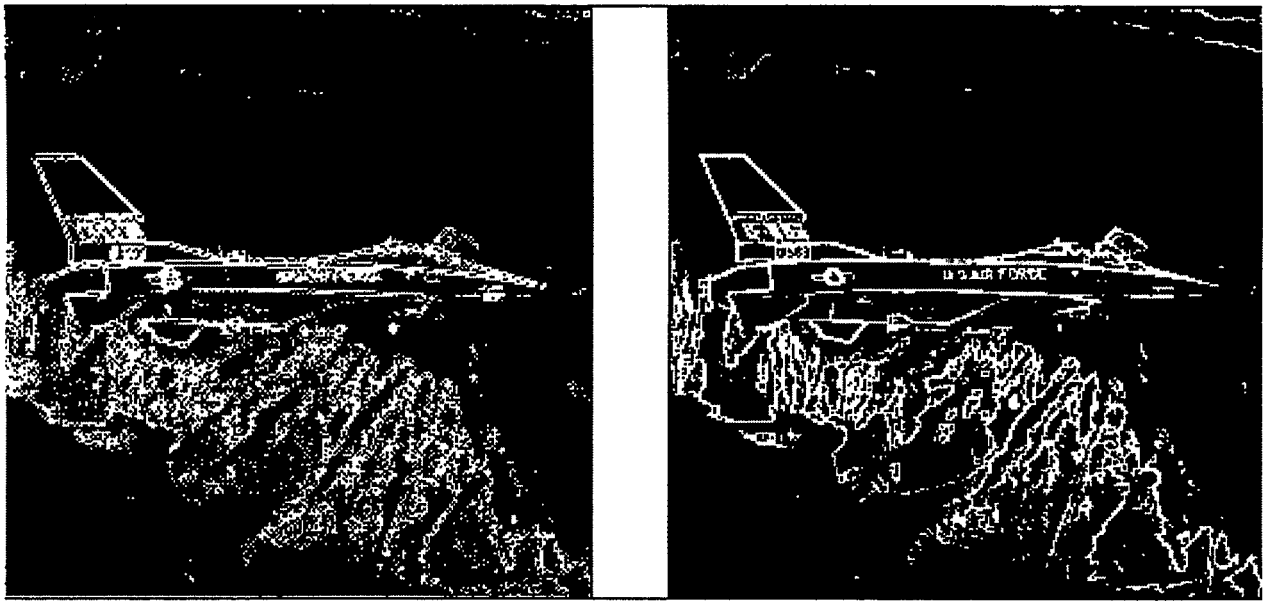


Figure 5.5 Edge detection using a) Spatial average filtering, and b) Morphological Ext. Gradient

In Fig. 5.5a, high-pass image is obtained by subtracting the original image from its low-pass version filtered by spatial averaging, and thresholding at gray level 5. Morphological external gradient (MEG) has been applied on the same image using a 3 x 3 SE. The high-pass image is obtained by subtracting the original image from its dilated version, and thresholding at gray level 5 (Fig. 5.5b). Even though the edges are detected in both techniques, the MEG produces clearer details compared to spatial averaging.

Frame execution times for edge detection using spatial average filter and MEG are 2.273 and 2.301 ms, respectively. It is noted that since the spatial average filter operation has been optimized at the bit-level by simple divisions by 2 and 8, it therefore, requires less time than MEG.

5.2.6 Performance Comparisons

5.2.6.1 Comparison with a RISC processor

A SUN SparcStation10, hosting a 32-bit SuperSPARC processor with 96MB of memory, running at 50MHz has been used to compare performances of different image processing algorithms. The term SuperSPARC and RISC (Reduced Instruction Set Computer) processor will henceforth, be

used interchangeably. The performance results are listed in Table 5.1. There is no performance difference in the enhanced C*RAM, and therefore, not listed.

Table 5.1 Performance comparison of image processing operations with a RISC (ms).

Operations	SuperSPARC 50MHz, 96MB	Baseline C*RAM	Speed-ups
Level shifting	116.677	0.020	5,833
Row-wide min/max search	166.663	0.389	428
Contrast stretching	203.030	10.834	18
Image subtraction	133.433	0.481	277
Gray-scale reversal	116.567	0.276	422
3 x 3 Spatial average filter	1,483.383	1.720	862
3 x 3 Erosion filter	1,466.267	1.697	864
Edge detection using spatial ave.	1,732.333	2.273	762
Edge detection using MEG	1,650.040	2.301	717

The smallest speed-up, of a few tens, has been obtained when contrast stretching is performed. When executing multiplication, the gain achieved by the massively parallel single-bit PEs cannot outperform the multiple-bit integer and floating point units of a RISC processor.

Greater speed-ups of a few hundreds have been obtained when performing pixel and local operations such as image subtraction, row-wide min/max searches, etc., where the algorithms are vectorized at the fine grain parallelism. Greatest speed-up is achieved with sub-pixel operations, where only a fraction of the bits are manipulated.

5.2.6.2 Comparisons with IMAP and HDPP

Performances of spatial average filter have also been reported on other logic-in-memory SIMD systems such as IMAP and HDPP. The same number of PEs is assumed among the systems under study. On the 8-b PE IMAP system running at 40MHz, a 512 x 512 image is spatial average filtered in 0.42ms [Okazaki95]. Compared to 25MHz 1-b PE C*RAM simulation on a 256 x 256 image, the *equivalent IMAP execution time* is: (40/25)*(512/256)*(8)*0.42ms = 2.69ms.

The first factor scales the IMAP PE-clock to 25Mhz, i.e. C*RAM PE-clock, while the third factor assumes IMAP has 1-b PEs. The second factor scales the image row to 256. It is noted that the image column does not contribute any coding delay since the numbers of PEs of the systems are assumed the same.

On the 2-D HDPP simulation, a 100-convolution spatial average operation takes 3.10 ms on a 10 MHz 1-b PEs [Gealow96]. The *equivalent HDPP execution time* is: (10/25)*(256/100)*(1)*3.10ms = 3.17ms. Similar to previous comparison, the first is a frequency scaling factor, the second operation scaling factor, and the third datapath scaling factor. The equivalent spatial average execution times of C*RAM, IMAP, and HDPP are summarized in Table 5.2.

Table 5.2 Comparisons in spatial average filter operation.

	Exe. time (ms)	Speed-ups
C*RAM	1.72	1.0
IMAP	2.69	1.5
HDPP	3.17	1.8

For 3 x 3 spatial average filter, it executes faster on the C*RAM eventhough the IMAP has larger PE (more than a factor of 8), and the HDPP has a 2-D network. These comparisons demonstrate that C*RAM is more efficient in terms of binary shiftings and left-right data transfers compared to IMAP and HDPP.

5.3 DCT

DCT has been chosen for spatial redundancy reduction due to its fast algorithm while maintaining a near optimal compression [Jain89]. Among many fast DCT algorithms which have been proposed in the literature, two algorithms are selected for implementations: Lee's algorithm [Lee84] for its regular structure, and Cho *et al.*'s algorithm [Cho91] for its fast execution time. The algorithms will be implemented on both the baseline and the enhanced C*RAM's.

Computation precisions will be studied with respect to those obtained by the SUN SparcStation10. Two configurations, namely, the 1 x n^2 and the n x n, will be considered and the corresponding execution times will be compared. Memory requirements will be discussed last.

5.3.1 General Considerations

DCT is generally applied to non-overlapping 8 x 8 pixel blocks of an image. Prior to processing, the value of a grayscale pixel is level-shifted from (0,255) to (-128,127) for a 8-b signed 2's complement representation. Recall from Section 5.2.1 that level-shifted operation is performed by inverting the MSB of a pixel value. Basic operations in DCT are addition, subtraction, multiplication, and possibly, data transfer. Addition/subtraction has briefly been discussed in Section 4.1.1. In this section, multiplication will be discussed. Multiplication is performed by repeated shiftings and additions. That is, if the multiplier bit is a '1', the multiplicand will be added to the right-shifted partial sum; otherwise, no operation is required. Therefore, the number of 1's in the multiplier determines the number of shift-and-additions required. Due to the bit-serial nature of the operations, the right-shifted operation on the partial sum can be implemented by aligning the LSB of the multiplicand to the second LSB of the partial sum. Booth's recoding algorithm[1] [Booth51] has also been considered. Booth's recoding multiplication can only be beneficial for special multiplicative constants where the number of alternating patterns is less than the number of 1's in the binary representation. For signed-multiplications, extra cycles are required for sign-extension operations.

5.3.2 Implementation of Lee's Algorithm [Lee84]

This algorithm is an optimization of Chen *et al.*'s algorithm [Chen77]. Chen's algorithm is based on the separability of the DCT. Basically, an n x n DCT can be decomposed into 2n 1-D DCT

1. Booth's recoding algorithm specifies that if a '10' pair of the multiplier is encountered, the multiplicand is subtracted from the partial sum. If a '01' pair is encountered, the multiplicand is added to the partial sum. Finally, when either '00' or '11' are encountered, no operation is required.

operations. First, the 1-D DCT is applied to n rows, then the same 1-D DCT operation is applied to the columns of the intermediate matrix. For 1-D DCT, Chen's algorithm requires 16 multiplications and 26 additions; Lee's algorithm, on the other hand, reduces the number of multiplications to 12, while slightly increasing the number of additions to 29. For 2-D DCT, the number of multiplications and additions for Chen's algorithm and Lee's algorithm are (256, 416), and (192, 464), respectively.

Multiplication with DCT coefficients is done by scaling the multiplicative constants by a factor of 255 for 8-b representations. A list of common multiplicative constants is shown in Table 5.3.

Table 5.3 Common multiplicative constants in scaled binary representations

i	dec[cos(i)]	bin[cos(i)]	Shift-Add	Booth's recoding
$\pi/16$	0.981	0.1111 1011	7 adds	*4 adds
$2\pi/16$	0.924	0.1110 1101	6 adds	6 adds
$3\pi/16$	0.831	0.1101 0101	5 adds	8adds
$4\pi/16$	0.707	0.1011 0101	5 adds	8 adds
$5\pi/16$	0.556	0.1000 1110	4 adds	4 adds
$6\pi/16$	0.383	0.0110 0010	3 adds	4 adds
$7\pi/16$	0.195	0.0011 0010	3 adds	4 adds

The first entry, $\cos(\pi/16)$ has seven 1's and four alternating pairs - going from right to left 01, 10, 01, and 10. The usual shift-add algorithm requires 7 additions while Booth's recoding algorithm requires only 4. For the rest of the entries, the shift-add algorithm results in fewer number of additions.

Quantization, often follows DCT, is performed by multiplying the resulting transformed coefficients by a scaled quantization matrix. An additional sixty-four 16-b x 8-b signed multiplications are needed.

5.3.2.1 Precision

In this section, the precisions obtained by computing on the bit-serial C*RAM will be studied. Let p.q be the fixed point format where p and q are the numbers of bits required to represent an integer part - including a sign bit, and a fractional part, respectively. An arbitrary 8 x 8 pixel input block, prior to level-shifting, is shown below:

142	148	143	154	154	166	164	167
144	140	144	147	149	155	159	163
135	141	140	148	138	152	152	165
135	143	140	147	142	155	153	165
137	139	141	143	150	152	155	153
136	139	141	143	144	151	148	151
137	135	144	141	145	144	152	147
131	137	136	138	138	146	150	144

Using a RISC processor, the resulting 8 1-D DCT vectors, expressed using 2 decimal points, are:

151.30	-50.60	2.07	4.89	-2.83	-2.68	-2.39	-15.11
125.14	-40.93	10.16	-0.19	3.54	4.27	4.21	-0.34
103.93	-43.29	13.32	-16.58	0.71	0.10	4.43	-19.53
110.29	-45.10	10.55	-11.06	-1.41	-5.63	3.29	-17.67
103.22	-36.47	-2.39	4.50	-2.83	-1.16	-2.07	3.49
91.20	-27.95	-1.91	-0.35	-3.54	-2.29	4.62	-5.26
85.55	-24.72	-2.23	-2.78	-3.54	7.79	0.16	11.42
67.87	-29.11	0.99	1.54	-12.73	3.58	-5.00	-3.63

while using Lee's algorithm with 10.2 signed 2's complement representation, the resulting vectors, after binary-to-decimal conversion, are:

151.25	-50.75	1.75	4.50	-3.00	-3.00	-2.50	-15.25
125.00	-41.25	9.75	-0.50	3.50	4.00	4.00	-0.50
103.75	-44.00	13.00	-17.25	0.50	-0.25	4.25	-19.75

110.25	-45.50	10.50	-11.50	-1.50	-5.75	3.25	-17.75
103.00	-37.00	-2.75	3.75	-3.00	-1.75	-2.25	3.25
91.00	-28.50	-2.25	-1.00	-3.75	-2.75	4.50	-5.50
85.50	-25.25	-2.50	-3.25	-3.75	7.50	0.00	11.25
67.75	-29.50	0.75	1.00	-12.75	3.25	-5.00	-3.25

It can be seen that the 1-D DCT's coefficients obtained by bit-serial C*RAM implementation are very closed to those obtained using the SuperSPARC processor. For 2-D DCT, the resulting transformed matrix using the SuperSPARC processor, expressed using 4 decimal points, is shown below:

592.8209	-210.8087	21.6003	-14.1673	-15.9989	2.8082	5.1201	-32.9671
123.1927	-44.7572	22.3508	-6.6099	18.2213	-8.6148	6.8729	-33.0883
11.1772	3.8452	-6.0355	17.3485	-9.3700	12.5546	-9.7425	9.5182
45.2333	5.1334	-23.6416	26.8492	1.9049	-4.3758	-1.4120	18.4931
18.9967	-17.2508	-5.7404	13.9786	-11.9999	-11.1485	-13.8581	-13.5911
15.8796	-6.2005	2.1721	-16.7691	0.5680	-3.2719	1.8969	-15.2303
-12.2060	-4.4403	1.2574	-7.9350	-6.9425	-10.2300	1.0357	-34.8810
-2.0729	0.5174	-6.6962	0.9791	0.1432	7.0969	-7.1499	13.1795

The 2-D DCT using Lee's algorithm with 12.4 signed 2's complement representation, also provides closed results.

592.1250	-213.3750	19.9375	-17.1875	-16.8125	0.8750	4.3750	-33.8750
122.8750	-44.3750	22.0000	-6.2500	18.0625	-8.5000	6.5000	-33.0000
11.2500	4.3125	-6.2500	17.7500	-9.3750	12.6875	-9.6875	9.5625
44.8750	5.3125	-24.0625	26.7500	1.5000	-4.8750	-1.7500	18.3750
19.0625	-16.8125	-5.5000	13.9375	-11.8750	-11.1875	-13.6250	-13.4375
15.9375	-6.2500	2.3750	-16.6250	0.2500	-3.0000	1.8125	-15.1250
-12.3750	-4.6875	1.1875	-8.3125	-7.1250	-10.4375	1.0625	-34.9375
-2.2500	0.1875	-7.0625	0.6250	-0.1250	6.6875	-7.4375	13.0625

After dividing by a typical quantization matrix [ISO/IEC 13818-2], the quantized DCT coefficients obtained by both approaches are identical as shown below:

74	-13	1	-1	-1	0	0	-1
8	-3	1	0	1	0	0	-1
1	0	0	1	0	0	0	0
2	0	-1	1	0	0	0	0
1	-1	0	0	0	0	0	0
1	0	0	-1	0	0	0	0
0	0	0	0	0	0	0	-1
0	0	0	0	0	0	0	0

It has been shown that using 16-bit fixed-point integer, the quantized 2-D DCT coefficients of an arbitrary block using C*RAM are identical to those obtained by 32-bit SuperSPARC processor. The coding quality is, therefore, assumed unaffected.

5.3.2.2 The n x n Configuration

Each 8 x 8 pixels block is assigned to a group of 8 PEs, 1 column per PE. Computations are performed as specified by the data-flow diagram, as shown in Fig. 2.5. In this configuration, not all PEs participate in global operations, thus, masks will be used to inhibit writes to non-participating PEs local memories. Matrix transposition is required in between row-wide and column-wide DCT computations.

5.3.2.3 The 1 x n^2 Configuration

Each 8 x 8 pixels block is stored in one PE as shown in Fig. 5.6.

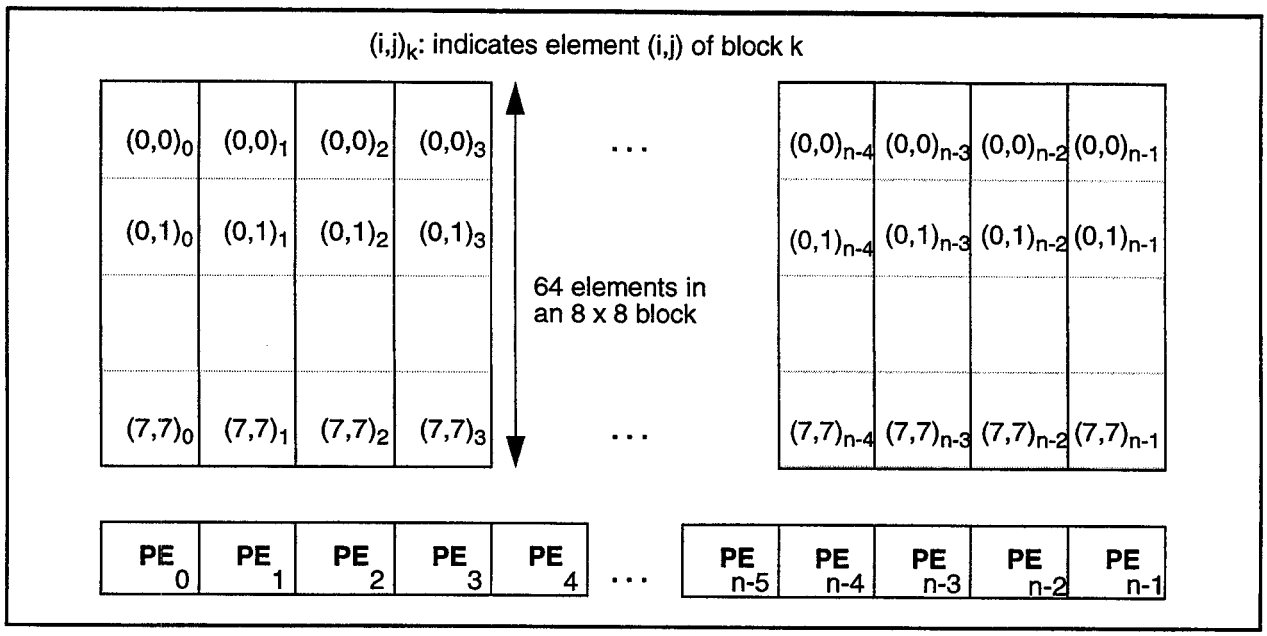


Figure 5.6 1 x n^2 block data arrangement for DCT computation

Unlike butterfly-FFT implementation (Fig.2.5), where computations are assigned groups of 8 PEs, this configuration unfolds each DCT computation and assigns it to a single PE. Each PE performs computations in serial while data transfers are contained within its own local memory. Parallelism is achieved by the large number of PEs operating on the blocks at the same time. Table 5.4 summarizes DCT and quantization times (without I/O times) in both configurations.

Table 5.4 Block DCT and quantization times using Lee's algorithm

n x n	1 x n^2
1.495 ms + 0.153 ms	5.488 ms + 1.224 ms

The DCT computation time is smaller for the n x n configuration because the number of PEs used is 8 times greater. The ratio in DCT computation times of the two configurations is, however, not 1:8. This is because the PEs in the n x n configuration have not been fully utilized. They are only 50% utilized when performing data transfers, and multiplications; and 100% utilized when performing additions and subtractions. On the other hand, the PEs are always 100% utilized in all operations using the 1 x n^2 configuration, and no matrix transposition is required.

Coefficient matrix quantization is performed in parallel using the n x n configuration, and in serial using the 1 x n^2 configuration. The ratio in execution times is exactly 1:8. All the PEs are 100% utilized during matrix quantization.

It appears that the 1 x n^2 configuration has some advantages compared to the n x n configuration. Depending on a particular application, it should be possible to decide if one configuration is more suitable than the other. If DCT and the subsequent matrix quantization are the only operations, as specified in JPEG algorithm, the 1 x n^2 configuration is chosen. If other operations, such as prefiltering and ME, are involved, the data arrangement problem needs to be analyzed for the entire application.

As a rule of thumb, when the number of available PEs is less than or equal to the number of 8 x 8 pixel blocks, the 1 x n^2 configuration is more appealing. When the number of PEs is significantly large, the n x n configuration is more suitable. Data arrangements of the preceding and subsequent operations are also important in deciding whether a particular configuration is more suitable.

5.3.3 Implementation of Cho's algorithm [Cho91]

Cho's algorithm is based on the simplification of the expanded 2-D algorithm. This algorithm reduces the number of multiplications and additions to 96 and 466, respectively. 1-D DCT operation is still applied at the beginning. The 1-D DCT Lee's algorithm is chosen for this step. Subsequent operations are optimized, and therefore, tend to be irregular.

The number of multiplications is greatly reduced in this implementation which results in faster operation. However, drawbacks are anticipated: the number of working memory-map registers increases by a factor of 2, and the address control becomes very complex and error prone. Similar to Lee's implementation, the coefficients are constant to all the PEs, and thus need not be stored in PEs memory. DCT and quantization times are listed in Table 5.5:

Table 5.5 Block DCT and quantization times using Cho's algorithm

n x n	1 x n^2
1.219 ms + 0.153 ms	3.443 ms + 1.226 ms

5.3.4 Implementations on the Enhanced C*RAM

When implemented on the enhanced C*RAM, additional savings are obtained by taking advantages of the added features, especially for the n x n configuration. No speed-up is obtained for the 1 x n^2 configuration.

Table 5.6 Block DCT and quantization times using n x n configuration

Algorithm	Baseline (ms)	Enhanced (ms)	% Reduction
Lee's	1.495 + 0.153	1.336 + 0.153	10.6 + 0.0
Cho's	1.219 + 0.153	1.108 + 0.153	9.1 + 0.0

A decrease in 10.6% and 9.1% have been obtained using Lee's and Cho's algorithms, respectively. These improvements can be attributed to:

- Parallel addition/subtraction circuits;
- Mask loadings can be minimized since the 8 x 8 groups are partitioned once at the beginning;
- Parallel transfers using the PSB;

5.3.5 Memory Requirements

Memory requirements vary over algorithms used as well as data arrangement configurations. Computation in Lee's algorithm is very contained within a row (column) and therefore, requires minimum amount of working memory - 0.5Kb for data and 1.5Kb for working space. In Cho's algorithm, on the other hand, computations spread out to many rows in an irregular manner. As a result, working memory is twice as many as that requires by Lee's.

In terms of storage for multiplicative constants, all PEs in the 1 x n^2 configuration operate on the same constant at a particular time. The set of multiplicative constants can be interpreted at the

controller as strings of 1's and 0's, and thus, there is no need to be stored in memory. On the other hand, PEs in the n x n configuration are partitioned into groups of 8. Constants used by one PE are different from constants used by another PE in the same group. Rows of memory must be allocated for these constants.

5.3.6 Summary of DCT Implementations

The block DCT (including quantization) times of different configurations and algorithms are summarized in Table 5.7 below:

Table 5.7 Block DCT (ms) using different algorithms on various configurations

Algorithm	Baseline n x n	Baseline 1 x n^2	Enhanced n x n
Lee	1.648	6.712	1.489
Cho	1.372	4.667	1.261

Table 5.7 shows that the enhanced n x n implementations are 10% better than their baseline n x n counterparts, and implementations using Cho's algorithm are 15 to 30% better than Lee's.

In order to compare with the performance of a RISC processor, I/O timings have been included. Knowing that the controller can be matched to provide data and instructions over the 8-b bus at the rate of 25MHz, the block DCT I/O time is: (8*8*8b) / (8b bus*25MHz) = 2,560 ns. Similar DCT operation has been run on the SuperSPARC for 10.712 ms. With adjustments to the I/O time, the speed-ups in block DCT times compared with the SuperSPARC are listed in Table 5.8:

Table 5.8 C*RAM speed-ups in block DCT over a RISC processor using different algorithms on various configurations

Algorithm	Baseline n x n	Baseline 1 x n^2	Enhanced n x n
Lee	6.5	1.6	7.2
Cho	7.8	2.3	8.5

Tables 5.7 and 5.8 provide some performance merits to different DCT algorithms implementing on various C*RAM designs and configurations. The speed-ups in block DCT ranges from 2 to 8.

No limitations are placed on any schemes and the criteria for selecting which schemes are made based the regularity of implementation, and overall timings and arrangement required. These speed-ups should be higher when the entire image is considered.

It is worth mentioning that it takes 4.95 ms to perform DCT on a 512 x 512 image using the 40 MHz IMAP system with 8-b PEs [Yamashita94]. After frequency scaling (by 40/25), operation scaling (by 1/4), and datapath scaling (by 8), the equivalent block DCT execution times is 15.84 ms. On the other hand, block DCT (including quantization) requires 6.71 ms on the 25MHz 1-b C*RAM. Therefore, it can be concluded that C*RAM performs better in FFT-based operations such as DCT.

In Chapter 7, block DCT implementation will be studied with regard to the image and video compression standard as a whole. The n x n configuration will be selected for the overall implementation for reasons discussed in Section 6.2.1. Worst-case block DCT is the Lee's algorithm implemented on the baseline C*RAM, while best-case block DCT is the Cho's algorithm implemented on the enhanced C*RAM.

5.4 Entropy Coding

In the entropy coder, the quantized DC and AC coefficients, and motion vectors are losslessly compressed. There are two stages: the run-length coder replaces consecutive zeros by their run length thus reducing the number of samples, while the VLC assigns shorter codewords to more probable source symbols so that the average number of bits per source symbol is minimized. Implementation of the entropy coder can be described by the following 3 operations:

5.4.1 Differential Coding of Quantized DC Coefficients

DC coefficients of adjacent blocks are differentially coded. This is performed by subtracting DC_i from DC_{i+1}. The differences in DC values are sent off-chip and variable-length coded.

5.4.2 Run-Length Coding of Zig-Zag Scanned, Quantized AC Coefficients

The next step is to determine the runs of zeros from the zig-zag scanned AC coefficients. Among the PEs, the number and order of zeros and non-zeros are not necessarily the same. However, in an SIMD architecture, at every instance, all the PEs must execute the same instruction. The following algorithm describes the run-length coding process on a SIMD architecture:

- Step 1: Check for zeros. This zero-checking operation results in group A with zero-value PEs, and group B with non-zero PEs.
- Step 2: The W registers of group A are masked out, and all the RUN counters and non-zero values of group B are stored. Then, all the RUN counters are reset. The masking operation ensures that only those of group B are affected by the subsequent operations.
- Step 3: The W registers of group B are masked out, and all the RUN counters in group A are incremented. Similar to the masking operation in step 2, only the RUN counters of group A are incremented, while those of group B are not affected.

Table 5.9 Block execution times (ms) for DC differential and AC run-length codings

	$1 \times n^2$	$n \times n$
Baseline	0.102	0.090

The baseline C*RAM has been used for implementation. The enhanced C*RAM offers no speed-up since operations of run-length coding does not make use of the enhanced features.

It is noted that the RUN counters can be represented using 6-bit numbers since there are 63 AC coefficients in each block, whereas the sizes of the AC coefficients can be represented using numbers of maximum 11 bits.

5.4.3 Variable-Length Coding for Run-Length Coded Coefficients

The RUN counters and non-zero coefficients can be represented by 6 and 11 bits, respectively. The 2 MSB's of the RUN counter represent the number of 16-runs of zeros. Therefore, these 2 bits are checked and coded first. The remaining 4 bits of the RUN counter and 11 bits of non-zero coefficient are grouped together and are subjected to a search operation. Many of the {RUN, non-zero AC coefficient} pairs are statistically improbable, and only 128 possible pairs are subsequently proposed for MPEG-2 implementation [ISO/IEC 13818-2].

Due to the irregular nature of the variable length codes, the {RUN, nonzero AC coefficient} pairs may be sent off-chip for variable-length coding. The on-chip alternative can be realized by storing a codebook of 128 representative {RUN, nonzero AC coefficient} pairs in C*RAM and table look-up operation may be performed to provide the index as the address to the best match pair. The indices are to be looked up in a ROM where actual variable-length codewords are stored.

There is no speed-up in the entropy coder because of data irregularity. The DPCM of DC coefficients and run-length coding of AC coefficients are, however, required to minimize I/O time for outputting similar DC coefficients and numerous zero-valued AC coefficients.

5.5 Sub-Codebook VQ for Image Compression

Vector Quantization is a low bit-rate compression technique where compression is achieved by representing a vector by its index (or address) to a codebook. The codebook, a set of representative vectors, had been generated from a set of standard images using a codebook generation algorithm. Even though VQ is not included as a component in any image and video compression standard, it remains attractive to applications where little processing power is anticipated at the low complexity receivers, and mobile communication devices [Kobayashi97, 98].

In this section, a Sub-Codebook VQ (SCVQ) implementation, where there are more codewords than the available PEs in a 1-D SIMD array processor, will be proposed. Different search strategies will be studied in details to achieve the most efficient implementation, while maintaining moderate reconstruction quality.

5.5.1 Implementation Procedure

VQ is popular for its low bit-rate operation and simple decoding at the receiver. Image quality, however, is greatly affected if the image does not have its representative[1] involved in the training process or the number of codewords is small. Therefore, a large codebook is preferred. VQ look-up operation can be tedious if the search is performed in serial. Due to data independency in the search algorithm itself, this look-up operation can be implemented on SIMD arrays [Le94, 95, 96], systolic arrays [Kung88, Yan90], and pipeline processors [Kolagotla93].

The following discussions are directly applied to any SIMD array processor including C*RAM. When the number of codewords is less than the number of PEs in a 1-D array processor, each PE can accommodate and represent one codeword. Therefore, the search for the best match codeword can be performed in parallel among the PEs.

When the number of codewords is greater than the number of PEs. They have to be arranged in such a way that allows for fast search while minimizing degradation in image reconstruction quality. This implementation provides mechanism for allocating large number of codewords into SIMD array processor with limited number of PEs, while achieving high reconstruction quality. The SCVQ implementation is described as follows.

Unlike TSVQ where zero-valued codewords exist in many leaves, the codewords of SBVQ are first re-arranged in order of increasing mean and variance to eliminate empty words. They are

1. The coding of a face image should use a codebook generated from a set of images which included one or more face images.

then grouped into a number of sub-codebooks. The codewords in each sub-codebook are stored in one level of hierarchy so that one sub-codebook may be searched at a time. At the top level of the hierarchy, the centroids (or representatives) of the sub-codebooks are stored.

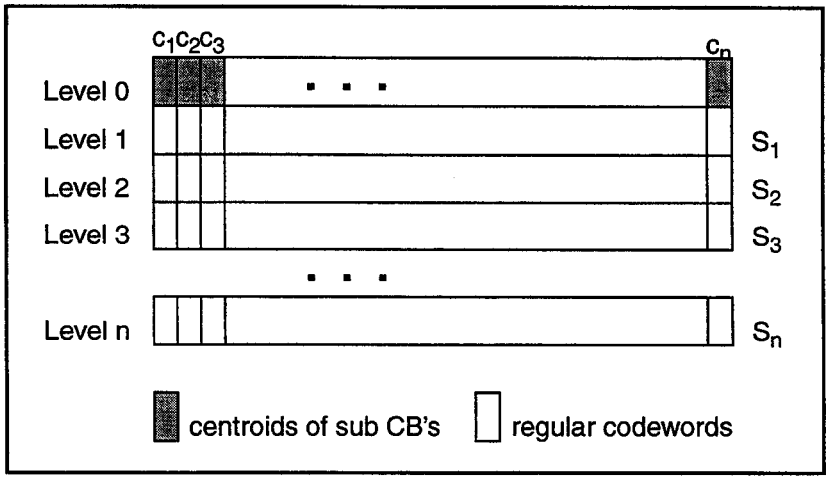


Figure 5.7 SCVQ for large codebook allocation

For instance, assuming there are 256 codewords in the original codebook (Fig 5.7), and the SIMD array processor has only 64 processors, the original codebook is first divided into 4 sub-codebooks S_1, S_2, S_3, and S_4. The corresponding centroids c_1, c_2, c_3, and c_4 of the sub-codebooks are calculated (at the host) and stored in level 0, while the sub-codebooks are stored in the subsequent levels 1, 2, 3, and 4.

Searches for the best match codeword are performed in at most 2 passes. During the first pass, the 16-D centroids of the sub-codebooks are compared with the 16-D input vector in parallel. The sub-codebook whose centroid best matches the input vector will be selected. The second pass will be performed on the entire selected sub-codebook for the final match.

5.5.2 Coding Performances

Simulations have been performed on 256 x 256 pixel images: Airplane - scenery image, Baboon - animal image, and Lena - face image, using their respective codebooks of 256 words. Table 5.10

lists the performance comparisons between FSVQ where all the codewords are available for search and SCVQ in order of decreasing NMSE.

Table 5.10 Performance of FSVQ vs. SCVQ

	FSVQ NMSE (%)	FSVQ PSNR (dB)	SCVQ NMSE (%)	SCVQ PSNR (dB)
Baboon	1.14	25.00	1.23	24.66
Lena	0.95	27.32	1.28	26.03
Airplane	0.25	28.88	0.34	27.58

In Table 5.10, Baboon image has the lowest reconstruction quality due to its complex features - animal's facial hairs require distinct codewords for good reconstruction (Fig. 5.8a), while Lena image has higher reconstruction quality - details in her hair and high contrast between her hat and the mirror frame (Fig. 5.8b). The Airplane image (Fig. 5.4) have the highest reconstruction quality because there are not a lot of details in the original images. The SCVQ is at most 1.3 dB lower in PSNR compared to FSVQ (0.33% in NMSE higher than the FSVQ)

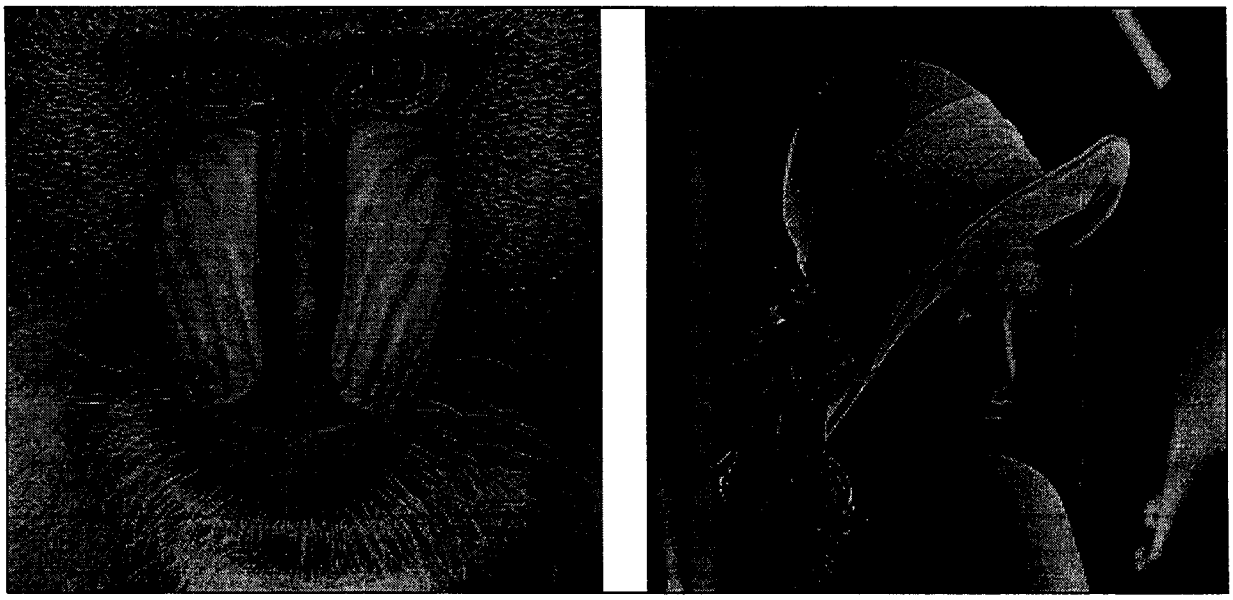


Figure 5.8 Original a) Baboon and b) Lena images of size 256 x 256 pixels

A closer look at the sub-codebooks reveals that codewords at the boundaries are very similar. In fact, if the best match codeword is not in the sub-codebook whose centroid is selected during the

first pass, then a similar codeword is selected and a wrong index is, in turn, chosen in the second pass. This causes degradation in image reconstruction.

A better arrangement is to store the codewords at the sub-codebook boundary into another level of hierarchy where they can be used to make fine adjustment. Assuming that the original codebook is divided into 4 sub-codebooks. Possible arrangement is shown in Fig 5.9.

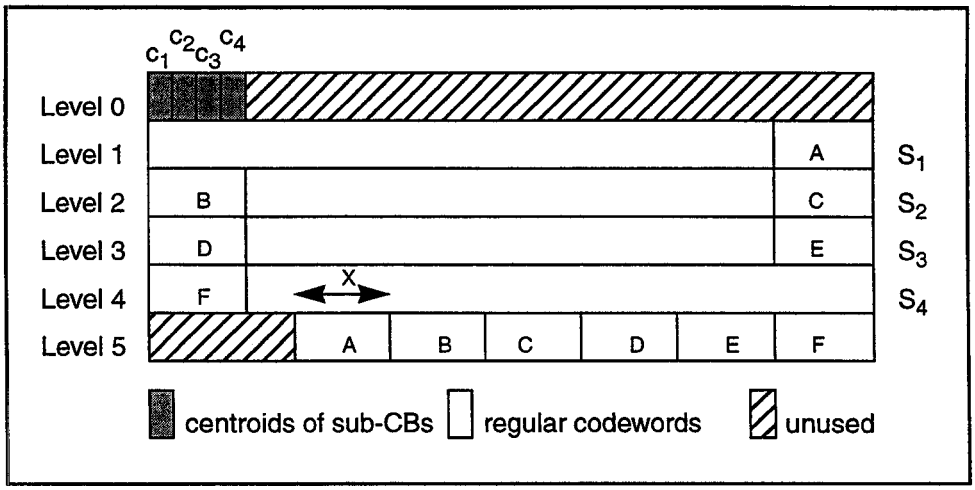


Figure 5.9 SCVQ with boundary consideration

Groups of x codewords at the sub-codebook boundaries: A, B, C, D, E, and F, are duplicated and stored in level 5. When there is a unique centroid match, say c_1, sub-codebook S_1 is search. When there are double matches in consecutive centroids, say c_1 and c_2, groups A and B will be searched. In case of triple matches, which is very unlikely, say c2, c3, and c4, the middle sub-codebook, in this case, S_3 will be searched. Simulation results are listed in Table 5.11.

Table 5.11 Performance comparison: SCVQ without versus with border consideration.

	SCVQ NMSE (%)	**SCVQ PSNR (dB)**	**SCVQb NMSE (%)**	**SCVQb PSNR (dB)**
Baboon	1.23	24.66	1.24	24.62
Lena	1.28	26.03	1.23	26.22
Airplane	0.34	27.58	0.34	27.57

The reconstruction qualities are unchanged for Airplane and Baboon images. For Lena image, reconstructions are improved by nearly 0.2 dB (or 0.05% in NMSE).

Finally, the SCVQb implementation has been extended to the encode images using universal codebook. Ten 256 x 256 standard images of different characteristics have been chosen as input vector pool to generate a 512-word universal codebook: Airplane and Sailboat - scenery images, Airport and Ptower - low contrast images, Baboon - high texture animal image, Ball and Visualmtf - graphics images, Chest - x-ray image, Lena - face image, and Moon - space image.

Table 5.12 Performance comparisons: FSVQ vs. TSVQ vs. SCVQb

	FSVQ NMSE (%)	**FSVQ PSNR (dB)**	**TSVQ NMSE (%)**	**TSVQ PSNR (dB)**	**SCVQb NMSE (%)**	**SCVQb PSNR (dB)**
Airport	6.67	24.42	7.56	23.91	7.28	24.03
Ptower	5.00	22.35	6.28	21.58	6.72	21.24
Baboon	1.43	24.00	1.72	23.23	1.50	23.74
Lena	1.25	26.14	1.54	25.27	1.65	24.83
Sailboat	0.82	26.05	1.01	25.18	0.97	25.19
Moon	0.42	29.60	0.51	28.78	0.41	29.39
Airplane	0.36	27.30	0.47	26.18	0.43	25.95
Ball	0.21	37.73	0.28	36.59	0.22	37.07
Visualmtf	0.19	33.02	0.57	28.50	0.65	27.77
Chest	0.05	36.37	0.09	33.81	0.14	31.79

FSVQ and Tree-Search VQ (TSVQ)[1] are also added to compare with SCVQb. For the implementations, the centroids of the sub-codebooks and the boundary codewords (level 0 and level 5 - Fig. 5.9) are merged into one level to reduce search time. There will be 8 levels of sub-codebooks and 1 level of centroids and boundary codewords. The distance x is chosen to be 4.

In Table 5.12, due to low contrast, Airport and Ptower images (Fig. 5.10a and b) - taken at night - have the highest NMSEs, regardless of coding techniques. Baboon and Lena images (Fig. 5.8a

1. Source code of TSVQ was obtained from [WStree96]

and b) have moderate reconstruction errors, while other images such as Sailboat and Moon images (Fig. 5a.11a and b) have low reconstruction errors. All reconstructions, except Visualmtf and Chest images, are within 1.5 dB of the FSVQ technique.

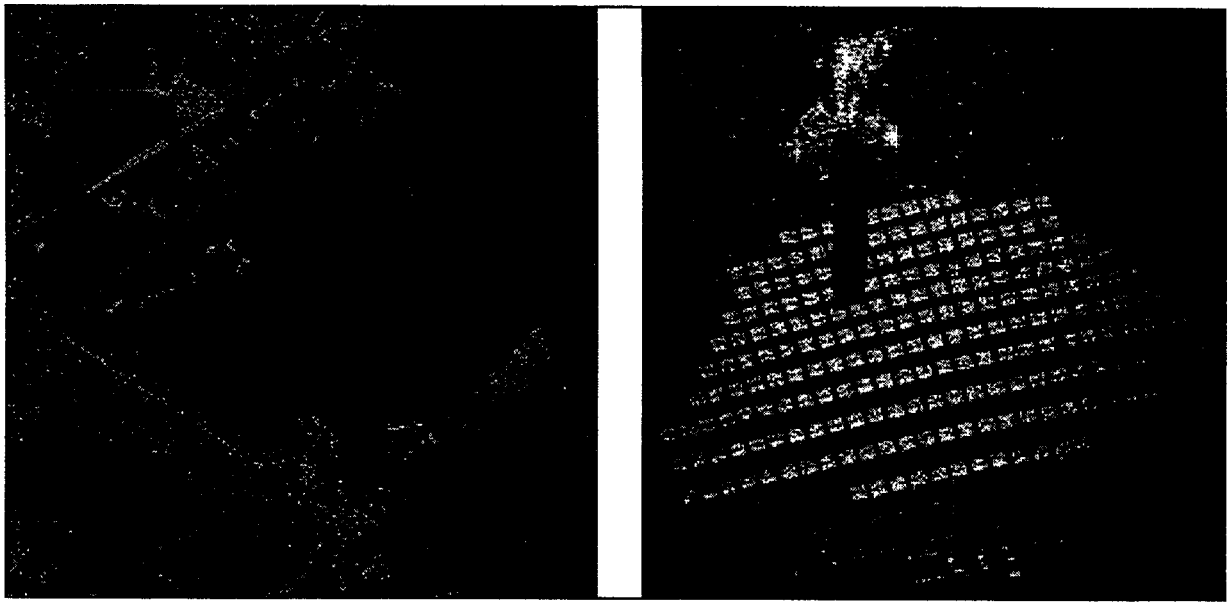


Figure 5.10 Original a) Airport and b) Ptower images of size 256 x 256 pixels

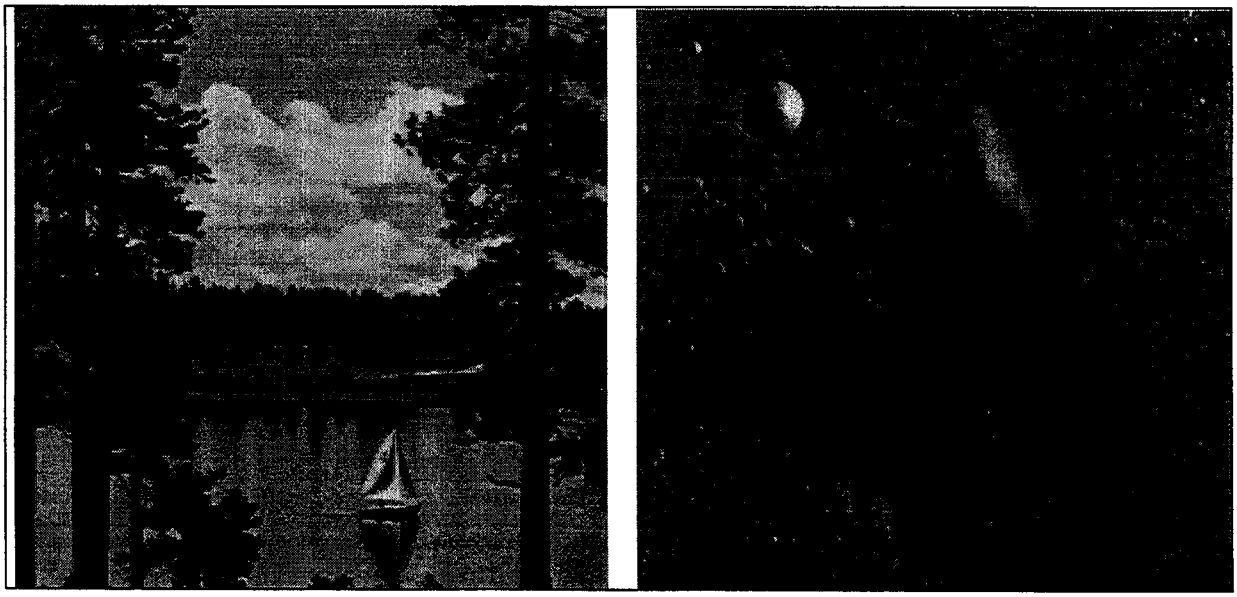


Figure 5.11 Original a) Sailboat and b) Moon images of size 256 x 256 pixels

The worst case reconstructions of Chest image will be visually examined. The reconstruction of the original Chest image (Fig. 5.12a) using 512-word FSVQ (Fig. 5.12b) will be displayed for ease of comparison with SCVQb (Fig. 5.13a) and TSVQ (Fig. 5.13b).

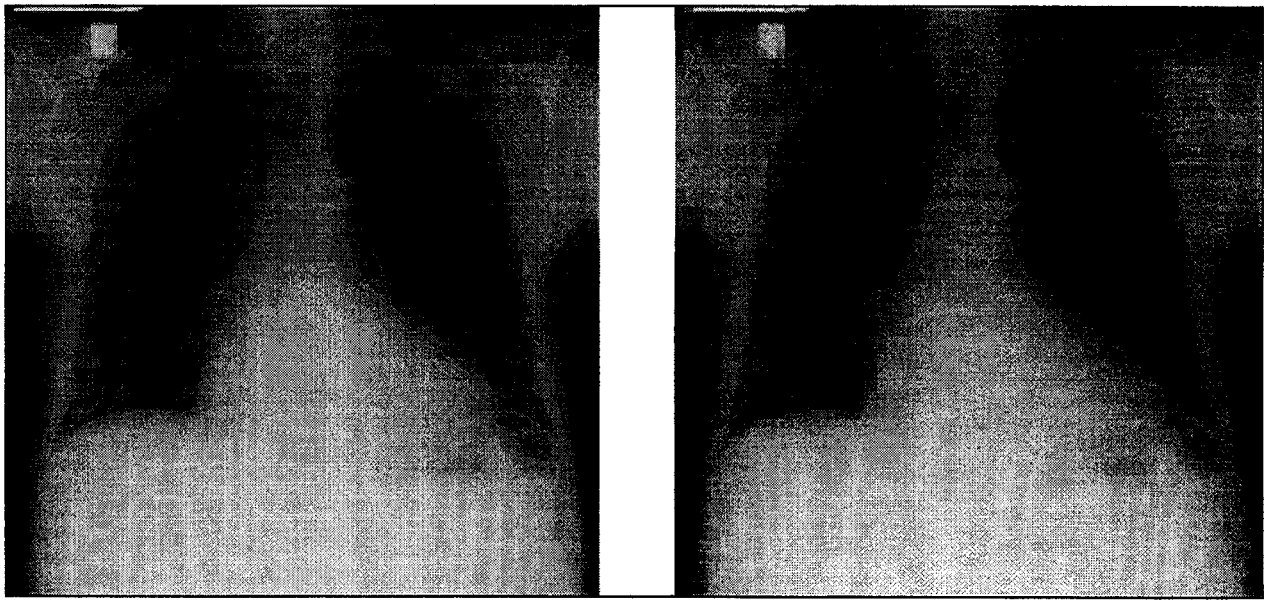


Figure 5.12 a) Original 256 x 256 Chest image and b) Its reconstruction using 512-word FSVQ.

Figure 5.13 Reconstructed Chest image using 512-word a) SCVQb and b) TSVQ.

It can be seen that recontruction using TSVQ is blockier compared to SCVQb. This is because TSVQ may end up with local minima, while SCVQb ends its search in sub-optimum global minima. Optimal minima may be found by increasing the number of boundary codewords.

In general, the coding performances of SCVQb are comparable to those of TSVQ in objective quality, while superior to TSVQ in subjective quality. The following section will be provided to study the speed-up achieved by SCVQb with respect to FSVQ.

5.5.3 Theoretical Speed-up Analysis

Let K, L, N be the number of input vectors, the codeword dimension, and the number of codewords in the universal codebook. Let s be codewords in the sub-codebooks.

It takes FSVQ K*L*N search operations to encode an image. On the other hand, it takes K*L*(N/s) operations to search for the best centroid, and another K*L*s to search for the best codeword in the selected sub-codebook. The resulting speed-up is (K*L*N) / [K*L*(N/s) + K*L*s] = N / [N/s + s]. Typically, N=512 and s = 64, the speed-up is thus 7.1.

On the generic 1-D SIMD machine with s PEs. It will take one search operation to determine the best match centroid, and another search operation to determine the best match codeword in the selected sub-codebook. The speed-up, due to parallel processing is, (K*L*N)/(K*L*2) = N/2.

Therefore, it can be said that SCVQb can achieve a speed-up of 7 and as high as N/2, compared to FSVQ.

5.5.4 Experimental Speed-ups

In order to compare with the performance of the Sun SuperSPARC processor, I/O timings have been included. Knowing that the controller can be matched to provide data and instructions over the 8-b bus at the rate of 25MHz, the block VQ I/O time is: (4*4*8b) / (8b bus*25MHz) = 640 ns.

With adjustment to I/O time, the speed-ups in block VQ times compared with the corresponding Sun SuperSPARC's values are listed in Table 5.13:

Table 5.13 Block VQ times and the corresponding speed-ups

	64 words	512 words
Sun SuperSPARC	7.229 ms	57.833 ms
64-PE Baseline C*RAM	57,400 ns (126)	114,160 ns (507)
64-PE Enhanced C*RAM	39,360 ns (184)	78,080 ns (741)

The speed-ups in the second column (126 and 184) can be attributed to on-chip parallel C*RAM implementations, while those in the third column (507 and 741) are the combined speed-ups of both on-chip implementations and special SCVQb arrangements. In fact, 507/126 = 741/184 = 4.0. This value of 4.0 is not near 7.1 as theoretically computed due to other factors such as I/O and C*RAM register settings.

Therefore, C*RAM implementation of VQ results in speed-up of 126 for baseline model, 184 for enhanced model. An additional speed-up of 4 is obtained by SCVQb arrangement.

5.6 Conclusions

In this chapter many image processing algorithms have been implemented. The smallest speed-up, of a few tens, has been obtained when contrast stretching is performed. When executing multiplication, the gain achieved by the massively parallel single-bit PEs cannot outperform the multiple-bit integer and floating point units of a RISC processor. Greater speed-ups (over RISC) of a few hundreds have been obtained when performing pixel and local operations such as image subtraction, row-wide min/max searches, etc., where the algorithms are readily vectorized. Greatest speed-up has been achieved with sub-pixel operations, where only a fraction of the bits are manipulated. Compared with other logic-in-memory SIMD systems, it has also been shown

that C*RAM is more efficient than IMAP and HDPP in terms of binary shiftings and left-right data transfers.

When mapping DCT and related quantization operations on C*RAM, the speed-ups over RISC in block DCT (including the subsequent matrix quantization) ranges from 2 to 8. They are larger when the entire image is considered. The criteria for selecting a particular scheme are made based the regularity of implementation (Lee's versus Cho's algorithms), and the overall timings and arrangements (1 x n^2 versus n x n configurations) required. It has also been concluded that C*RAM performs better than IMAP in FFT-based operations such as DCT.

Finally, C*RAM implementations of VQ results in speed-up (over RISC) of 126 for baseline model, and an additional speed-up of 4 is obtained by SCVQb arrangement. The speed-ups of different algorithms, especially VQ, have been further augmented by the C*RAM's enhanced features.

In Chapter 7, overall implementations of image/video processing components such as filterings, DCT, VLC, and ME will be put together for the realizations of image/video compression standards using C*RAM.

CHAPTER 6

A New Low-Complexity Motion Estimation Algorithm and Its C*RAM Implementation

In this chapter, a low bit-rate morphological-based ME algorithm (FEXOR) will be proposed. The motivation behind low-complexity ME is to achieve real-time video coding for applications which moderate quality is required. The algorithms will be presented in the following order: the algorithm, its performance, and algorithmic and architectural speed-ups. The second part of this chapter devotes to the implementations of various ME algorithms including: FBMA, PD, FEXOR, and BP_BPM.

6.1 Motion Estimation using Feature Extraction and XOR Operation

Motion estimation is a temporal image compression technique, where an n x n block of pixels in the current frame of a video sequence is represented by a motion vector with respect to the best matched block in a search area of the previous frame, and the DCT coefficients of the displaced block differences. A ME algorithm should have the following steps:

- Prefiltering: Noise is filtered out. In low-complexity algorithms, this step is to extract features from the image in terms of edges (step and ramp edges) and texture (non-ramp edges), necessary for motion detection;
- Distortion computation: every reference block is slid over a SA and the corresponding distortion is computed; and
- Motion vector determination: based on the distortions computed, the displacement vector corresponding to the minimum distortion is identified.

In FBMA, distortion computation and motion vector determination are the only steps. Pixels of block data are commonly 8 bits, and the entire search area is assumed. The MAE distortion measure gradually phases out MSE distortion measure due to its similar performance with simpler hardware realization. This FBMA algorithm is rather simple to implement on any uniprocessor.

It is generally accepted that ME using FBMA is computationally intense and its off-chip data transfer is overwhelming. These difficulties prevent ME-based applications from being operated in real-time. Therefore, algorithm developers have, over the years, attempted to reduce its complexity to a manageable level.

To reduce the number of search locations, TSS uses similar distortion computation method except that the search is performed on a coarse-to-fine basis. This algorithm works based on the assumption that the distortion is monotonically increasing as the search is moved away from the best match location. Since there are many local minima in a frame, algorithms based on this assumption do not necessarily yield the best results. When implemented on a uniprocessor, a speed-up of a factor of 8 to 9 is expected. When implemented on SIMD array processors, the control overhead and irregular data structure greatly offset the speed-up achieved by massively parallel feature of the SIMD architecture. As will be shown later, the TSS algorithm is only

feasible when the number of PEs is small. Therefore, TSS is considered inefficient when mapped onto the SIMD array processors.

PD uses similar distortion computation method except that only a fraction of the n x n pixels is used for computation. PD algorithm is also simple to implement on any uniprocessor. However, the mapping of PD to SIMD processors is neither efficient because no processing power is saved on the decimated data, nor accurate because proper alternating patterns required for accurate estimation may not be implemented. Similar arguments are applied to ME algorithm using Motion Field. The overall theoretical speed-up is from 8 to 16 times compared to FBMA.

Knowing that pixel operations such as filtering, and local operations such as block matching are suitable for SIMD array processors, low-complexity algorithms have been introduced. In this section, a new low-complexity technique for motion estimation is proposed. Unlike its predecessors, this algorithm is based on morphological image processing which is commonly used in machine vision and pattern recognition. A defective item on a conveyor belt can be detected by comparing its size and shape with the template characteristics of the standard item. In this concept, the color and shade of an item are not of importance.

The proposed technique attempts to fully vectorize the prefiltering step as much as possible, so that operations in subsequent steps can be avoided or reduced. First, it reduces the b-bit grayscale frames into 1-bit binary frames using morphological filters, and to determine the displacement of the edges and texture of the adjacent frames. While reduction in bit-depth requires a small percentage of computation at full pixel resolution, the search procedure is executed using simple XOR logic operations and 1-b distortion accumulations on the entire search area. Compared to other low complexity techniques, the proposed technique yields better frame reconstructions, operates using simpler arithmetic/logic operations, and possesses a higher degree of parallelism when implemented on a SIMD architecture [Le99].

6.1.1 Description of the Algorithm

According to BMA's assumptions, the edges of the object/region in consecutive frames should be similar in shape if conditions 1, 2, and 3 are met. There are three steps in FEXOR: 1-b transform, distortion computation, and motion vector determination. In the 1-b transform stage, a common morphological-based edge detection filter proposed by [Lee87] (briefly presented in Section A.4), and the proposed technique are simulated. In FEXOR, the grayscale images first undergo an opening-by-reconstruction operation using a 3 x 3 window to filter out any noise. The resulting image then undergoes EG, using a 5 x 5 window. Finally, the difference image, obtained by subtracting the original image from its processed version, is thresholded by a constant grayscale of 5 to generate a binary edge map. Dynamic thresholding can also be implemented.

In the subsequent distortion computation and motion vector determination stages, the FBMA technique is next applied on the entire SA of the 1-b edge maps. Since the edge maps are binary images, the single-cycle XOR operations are performed at each search location. At each pixel in the search block, the result of an XOR operation is a 0 if two pixels match, and 1, otherwise. The location resulting in the least number of 1's is the best match. Motion estimation using FEXOR generally results in a speed-up of nearly 8 (for 256 grayscale image) since the XOR operations and 1-b accumulations are simpler than the 8-b MAE operations.

6.1.2 Simulation Results

Simulations have been performed on the first 30 frames of 8 sequences: Football, Garden, and Table Tennis of sizes 720 x 480 pixels; Miss America and Salesman of sizes 352 x 288; Pingpong and Susie of sizes 352 x 240; Caltrain of size 512 x 400; and Trevor of size 256 x 256. In these simulations, the frames are divided into blocks of size 16 x 16 pixels, and the search range of (-16,+15) pixels in each direction has been chosen.

All simulation results are compared against the FBMA and the typical low-complexity BP_BPM algorithms. In Table 6.1, the sequences are listed in the order of decreasing NMSE in terms of reconstruction. The sport sequences such as Football, Table Tennis, and Pingpong have resulted in the largest NMSEs. The scenery sequences, such as Garden and Caltrain, have resulted in moderate NMSEs, while the head-over-background sequences, such as Salesman, Trevor, Miss America, and Susie, have lowest NMSEs.

Table 6.1 Average NMSE (%) of the tested sequences

Sequence	FBMA	BP_BPM	Lee87	FEXOR
Football	1.82	3.23	4.87	3.82
Table Tennis	0.99	1.15	1.48	1.26
Pingpong	0.82	0.95	1.25	1.06
Garden	0.62	0.74	1.03	0.79
Caltrain	0.32	0.41	0.43	0.44
Salesman	0.32	0.67	0.46	0.42
Trevor	0.28	0.33	0.47	0.33
Missa	0.26	0.45	0.49	0.40
Susie	0.12	0.17	0.20	0.16

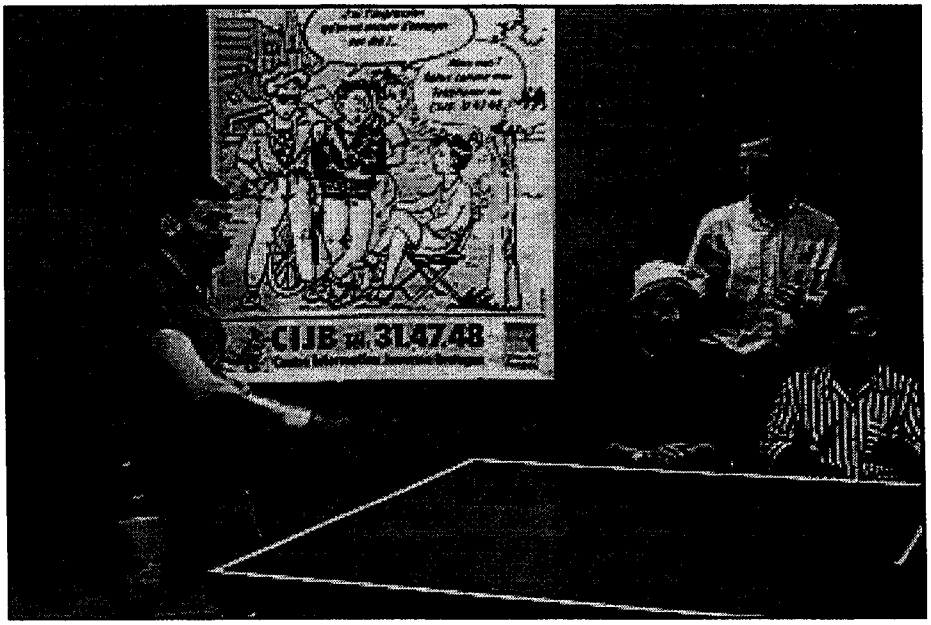


Figure 6.1 Original frame 12 of Pingpong sequence

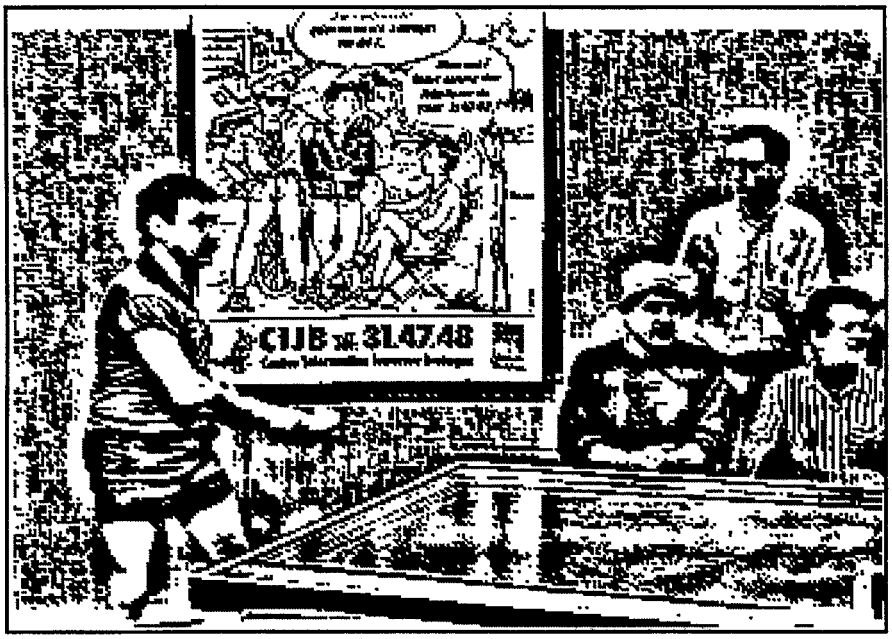


Figure 6.2 Binary frame 12 generated using BP_BPM

Fig. 6.1 shows frame 12 of the original Pingpong sequence. In Fig. 6.2, BP_BPM algorithm tends to keep all features. Edges are non-uniform in width, and objects tend to have shadows. In Fig. 6.3, ME algorithm using Lee's edge detection filter [Lee87] over-simplifies the details in the image. Therefore, the objects, large or small, necessary for ME have been removed.

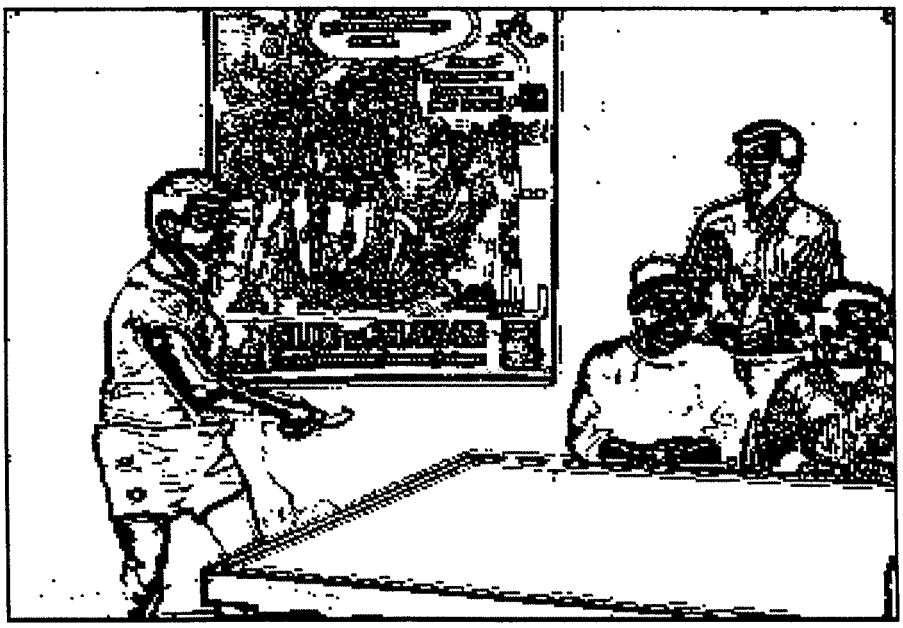


Figure 6.3 Binary frame 12 generated using Lee's filter

Figure 6.4 Binary frame 12 generated using FEXOR

In Fig. 6.4, the remaining details using FEXOR are blocky since the structuring element is a 3 x 3 square window. All changes in illumination have been eliminated.

Table 6.2 Average entropies (bpp) of the error frames

Sequence	FBMA	BP_BPM	Lee87	FEXOR
Football	4.49	4.81	5.02	4.84
Table Tennis	4.54	4.60	4.70	4.64
Pingpong	3.55	3.62	4.03	3.74
Garden	4.06	4.18	4.29	4.16
Caltrain	3.42	3.53	3.56	3.54
Salesman	3.12	3.23	3.30	3.23
Trevor	3.10	3.18	3.48	3.18
Missa	2.78	2.96	3.18	3.03
Susie	2.87	3.06	3.19	3.08

Table 6.2 lists the average entropies, in bits per pixel, of the error frames. The entropies of BP_BPM and FEXOR are within 0.3 bpp from FBMA. If the errors are compressed once more in the subsequent stage, the average entropies can be further reduced. This lends support to establish the benefit of low-complexity ME. In terms of entropy on the prediction error, BP_BPM and

FEXOR are comparable, while ME using Lee's filter results in large entropies. Therefore, low-complexity ME using Lee87's filter will not be discussed further.

6.1.3 Implementation of Dynamic Thresholding

In the above FEXOR algorithm, a fixed grayscale of 5 was selected for thresholding. This selection is rather rigid, and thus, does not adapt well with sequences with varying statistics. In order to address this problem, the dynamic range of the edge-enhanced images could be tracked using minimum and maximum (range) searches on local statistics of image data. The term 'local' can be block, row, or column of data. In particular, each row will be adaptively thresholded using a pixel dynamic range which can be determined at run-time. One possible implementation is described as follows: In each image row, conduct a parallel MIN search followed by a parallel MAX search. The difference in the MAX and MIN values is the dynamic range of the row pixels. This range is divided by 32 - for binary shift - and the zero vs. non-zero quotients determine the binary image.

Table 6.3 Average NMSE (%) over 30 frames using dynamic FEXOR

Sequence	FBMA	static FEXOR	dynamic FEXOR
Football	1.82	3.82	3.77
Table Tennis	0.99	1.26	1.37
Pingpong	0.82	1.06	1.09
Garden	0.62	0.79	0.81
Caltrain	0.32	0.44	0.47
Salesman	0.32	0.42	0.48
Trevor	0.28	0.33	0.37
Missa	0.26	0.40	0.48
Susie	0.12	0.16	0.19

From Table 6.3, the performance in terms of NMSEs are within 0.05%, yet no advanced knowledge in the threshold level is required. One exception is that dynamic FEXOR seems to perform better with fast changing sequences like Football.

Figure 6.5 Binary frame 12 generated using dynamic thresholding FEXOR

In general, dynamic FEXOR is superior when the sequence's statistics are unknown and change quickly with time.

6.1.4 Speed-up Analysis

Let N be the word length of a pixel. Also, let the following notations SUB, DIV, ABS, ACC, MIN, XOR, CMP, THR denote subtraction, division, absolute operation, accumulation, minimum search, XOR operation, comparison, and thresholding operation, respectively.

For FBMA, there is no prefiltering step. The distortion computation and motion vector determination steps are expressed below:

$$256*[256*(\text{N-b SUB} + \text{N-b ABS} + 2\text{N-b ACC}) + 2\text{N-b MIN}]$$

For a maximum displacement of 8, the corresponding number of search locations is 256. At each location, there are 256 MAE operations and one minimum search. The sequential approach requires:

$$256*[256*(3N + 3N + 3*2N) + 3*2N] = 811{,}008 \text{ N cycles}$$

If a block is stored as 16 rows of 16 pixels, then the total number of cycles, due to parallel processing, is reduced to:

$$16*[16*(3N + 3N + 3*2N) + 3*2N] = 3{,}168 \text{ N cycles}$$

For BP-BPM, the prefiltering step requires: 25 13-b ACC's and one 13b-by-8b DIV for each of the 256 pixels. The 13b-by-8b DIV can be considered as 13 8-b SUB's with signed extension. If 13-b is made equal to 1.5N, the pre-filtering step requires:

$$(256)*\{25*1.5N\text{-b ACC} + 13*1.5N\text{-b SUB} + 2*1.5N\text{-b (for signed extension)}\} =$$

$$(256)*\{25*3*1.5N + 13*3*1.5N + 2*1.5N\} = 44{,}544N \text{ cycles}$$

Due to parallel processing, the pre-filtering step of BP_BPM now only requires:

$$(256/16)*(5/16)*\{25*1.5N\text{-b ACC} + 13*1.5N\text{-b SUB} + 2*1.5N\text{-b (for signed extension)}\} =$$

$$5*\{25*3*(1.5N) + 13*3*(1.5N) + 2*1.5N\} = 870N \text{ cycles}$$

While the distortion computation and motion vector determination steps, without and with parallel processing, of the low-complexity algorithms require:

$$256*[256*(1/8N\text{ -b XOR} + N\text{-b ACC}) + N\text{-b MIN}] = 256*[256*(1/8N + 3N) + 3N] = 205{,}568N$$

$$\text{and } 16*[16*(1/8N + 3N) + 3N] = 864 \text{ N cycles, respectively.}$$

The total time complexities for BP_BPM without and with parallel processing are 250,112N and 1,734N cycles, respectively.

For FEXOR, the prefiltering stage without and with parallel processing requires:

$$(256)*\{4*4*N\text{-b CMP} + N\text{-b SUB} + N\text{-b THR}\} = 256*\{4*4*2N + 3N + 3N\} = 9728N$$

and $9728N/16 = 608N$ cycles, respectively.

The total time complexities for FEXOR without and with parallel processing are 226,816 N and 1,456N cycles, respectively.

Based on the above analysis, the relative execution times of FBMA and BP_BPM with respect to FEROX, without and with parallel processing, are 3.6: 1.1: 1.0 and 2.2: 1.2: 1.0, respectively. FEXOR is generally 10% to 20% less complex than BP_BPM in both cases. Since FBMA is more efficiently mapped onto the parallel architecture, its complexity is reduced significantly.

The low-complexity FEXOR results in a speed-up of 300% compared to FBMA using SIMD architecture. It should be noted that the proposed algorithms can be implemented on any comparable architecture and would result in similar orders of speed-up. The above (algorithmic) speed-ups increase moderately if implemented on the baseline C*RAM and significantly if implemented on the enhanced C*RAM.

6.2 C*RAM Implementation of ME Algorithms

ME has been chosen as part of the MPEG's and H.26x video compression standards. In each second, the ME module of an MPEG encoder has to perform 53.5 billion operations based on three {N=9, M=3} GOPs, typically of frame size 720 x 576 pixels, with a maximum displacement of (-16,+15). For this reason, ME is known to be the most time-consuming task in video compression.

The difficulty increases when data is brought off memory chips to the CPU for computation and later stored back in memory. Data bandwidth can be reduced by 3 to 4 orders of magnitude due to load capacity at the I/O paths, and bus speed and structure [Elliott97].

This problem can be alleviated by performing computation right inside the buffer where the frame data is stored. Moreover, due to the data parallel nature of the ME algorithm, parallel operations can be performed on the non-overlapping macro-blocks.

Recall from Section 2.6 that fast ME algorithms can be classified into 3 main streams:

- Reduction in the number of search locations;
- Reduction in the number of pixels involved; and
- Reduction in the bit-depth.

In this sections, 3 typical algorithms, namely FBMA, TSS, PD, and the proposed FEXOR algorithm will be considered for implementation, using two architectures: the baseline and the enhanced C*RAM's. Unless otherwise stated, the macroblock is referred to as block. The data arrangement and memory requirement will be discussed first.

6.2.1 Data Arrangement

6.2.1.1 Maximum Transfer Distance

Assuming that search range of d=16 or (-16, +15) is employed on a 256 x 256 frame operating on a 256-PE C*RAM (Fig. 6.6).

In the n x n configuration (Fig. 6.6a), each image column is assigned to a PE. Therefore, groups of 16 PEs accommodate blocks of 16 x 16 pixels. In the 1 x n^2 configuration (Fig. 6.6b), each PE accommodates one block, and each block must be stored in a 1-D format. Therefore, 256 PEs can be used to store all 256 blocks. In Fig 6.6a, block (1,1) is motion-estimated by matching it with blocks (0,0), (0,1), (0,2), (1,0), itself (1,1), (1,2), (2,0), (2,1), and (2,2). The maximum transfer distance of each datum is d=16 PEs.

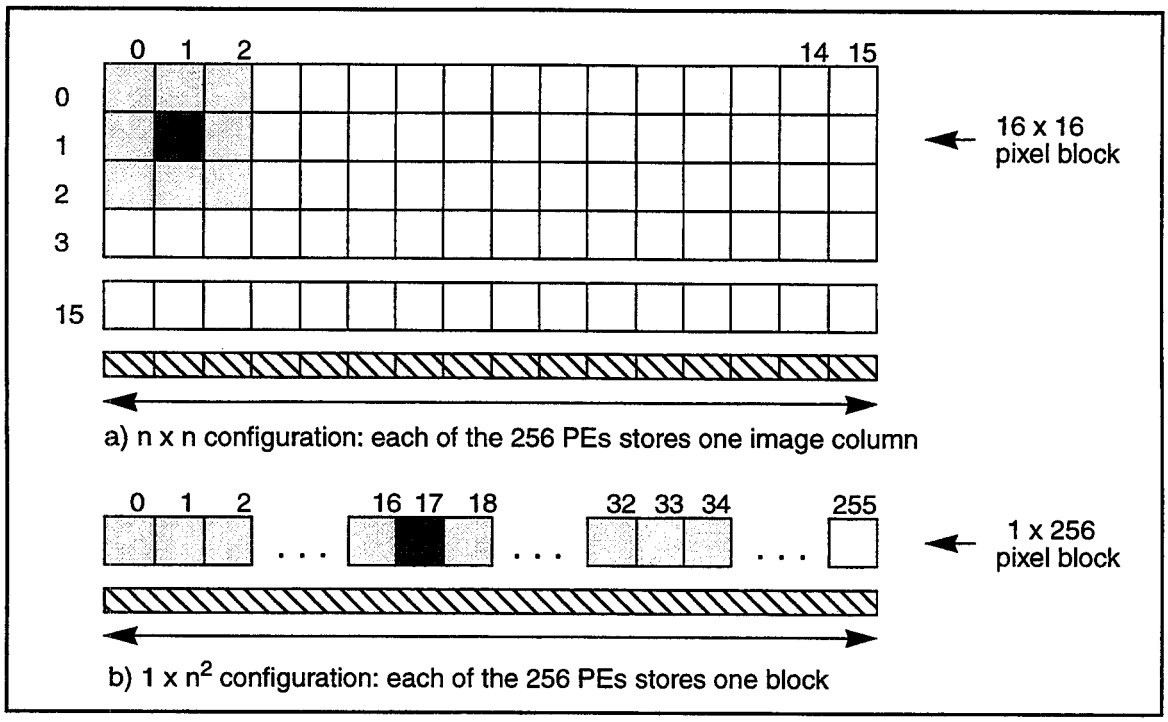


Figure 6.6 Image data arrangements

On the other hand, in Fig. 6.6b, block[1] (17) is motion-estimated by matching with blocks (0), (1), (2), (16), itself (17), (18), (32), (33), and (34). By visual inspection, the maximum transfer distance of each datum is d=17 PEs.

When the image is larger than 256 x 256, for instance 1920 x 1080, and the maximum search range is larger[2] than 16, i.e d = 48 or (-48,+47), the maximum distance that each datum is transferred is 16 and 204 PEs using configurations a and b, respectively.

In general, the maximum distance that each datum is transferred remains constant at 16 for the n x n configuration, and for the 1 x n^2 configuration, it is expressed as follows:

$$x_{max} = \lceil d/16 \rceil \times \{\lceil (columnsize)/16 \rceil + 1\} \tag{6.1}$$

1. Block (17) in Fig.6.6b is topologically similar to block (1,1) in Fig. 6.6a.
2. MPEG-2's High-level, High-profile

where column size is the horizontal dimension of an image, and $\lceil q \rceil$ is the ceiling operation which rounds up the quotien q to the nearest integer. Eq. 6.1 shows that x_{max} varies directly with both displacement d and the image column size. As either displacement d or image column size increase, x_{max} increases accordingly. Therefore, data transfer time (for distortion computation and motion vector determination) which is a function of x_{max} and the bit-depth of the pixels (and distortions) will become a parameter of concern.

6.2.1.2 Control Complexity

Control in the 1 x n^2 configuration is highly irregular, and thus, error prone.

6.2.1.3 PE Utilization

In the n x n configuration, the ME procedure, working on 16 rows of data, is repeated 16 times (over the 16 horizontal blocks). In the 1 x n^2 configuration, the ME procedure is, even though repeated once over one horizontal block, to operate on 16 times as many rows of data compared to the n x n configuration. The PE utilizations are thus the same and 100% in both configurations.

In summary, both configurations allow high PE utilization. The n x n configuration is, however, more regular and requires much smaller and constant data transfer distance among the PEs. Therefore, the n x n configuration is preferred.

6.2.2 Memory Requirement

For bi-directional ME, at least 3 frames of image data are required at a particular time instance: the current frame, and the future and past reference frames. Depending on the size of the displacement d, extra memory will have to be set aside for temporary results. A detailed description of memory required for data and padding will be presented in Section 7.2.1.

Recall from Section 6.1 that there are 3 steps to motion estimation: Prefiltering, distortion computation, and motion vector determinations. The low-complexity algorithms focus on the

prefiltering step, while others emphasize the distortion computation step. The motion vector determination step is common to all algorithms.

6.2.3 C*RAM's Implementation of a Single MAE Computation

Conventionally, an MAE computation involves 3 operations: subtraction, absolute operation, and summation. Due to the bit-serial nature of C*RAM along with its enhanced features, three implementations have been performed and summarized as follows:

- Approach 1: involves the normal distortion computation where each search includes: one subtraction, one absolute operation, and one accumulation per pixel, and one minimum search per vector;
- Approach 2: the accumulation is performed using a variable bit-length as opposed to a fixed 12-b distortion; and
- Approach 3: takes into consideration the proposed parallel addition/subtraction feature.

Simulations have been performed on a 256-D input stimuli. The results are shown in Table 6.4:

Table 6.4 Block MAE time per 256-D vector.

	Time (ms)	% Reduction	% Speed-up
Approach 1	1.341	0	0
Approach 2	0.908	32	48
Approach 3	0.620	(32) 54	116

Approach 1, a straight forward translation of the MAE operation, requires the longest computing time, and is chosen as a reference for other approaches. Approach 2 takes for granted the bit-serial nature and uses variable-length operations for distortion accumulation. As a result, the coding time is reduced by 32% (or a speed-up of 48%). Approach 3 achieves a reduction of 32% compared to approach 2 due to the parallel addition/subtraction enhancements. With the combination of variable-length operations and the enhancements, a reduction of 54% (or a speed-

up of 116%) per MAE operation, can be achieved. It is noted that similar MAE computation can be applied to a 16-D vector quantizer.

6.2.4 Implementation of FBMA

In FBMA, the search algorithm is performed exhaustedly for every location, using all the pixels in the block at their full resolution. The motion estimation algorithm is performed in 2 steps: distortion computation and motion vector determination.

Distortion computation involves aligning the reference block with the candidate block, subtracting the respective pixel values, performing an absolute operation on the differences (when the baseline C*RAM is used), and accumulating all distortions.

For identification purpose, the block is addressed by its upper-left coordinates. As simplified in Fig. 6.7, the MAE distortion at co-ordinate (k,l) can be computed as follows:

$$D(k, l) = \sum_{i=0}^{3} MAE(R_i, V_{i+k, l}) \qquad k, l = -4, \ldots, 3 \tag{6.2}$$

In Eq. 6.2, R_i and $V_{i+k,l}$ are themselves vectors of n elements, where n is the block size. For FBMA, each element is an 8-b pixel; for low-complexity ME algorithms, each element is an 1-b pixel. The same distortion computation step can be applied for both algorithms. Due to parallel nature of FBMA, the same operations will be applied to all other blocks across the PE array. Fig. 6.7 shows full operations on the even reference block, the same set of operations can also be applied to odd reference blocks in parallel.

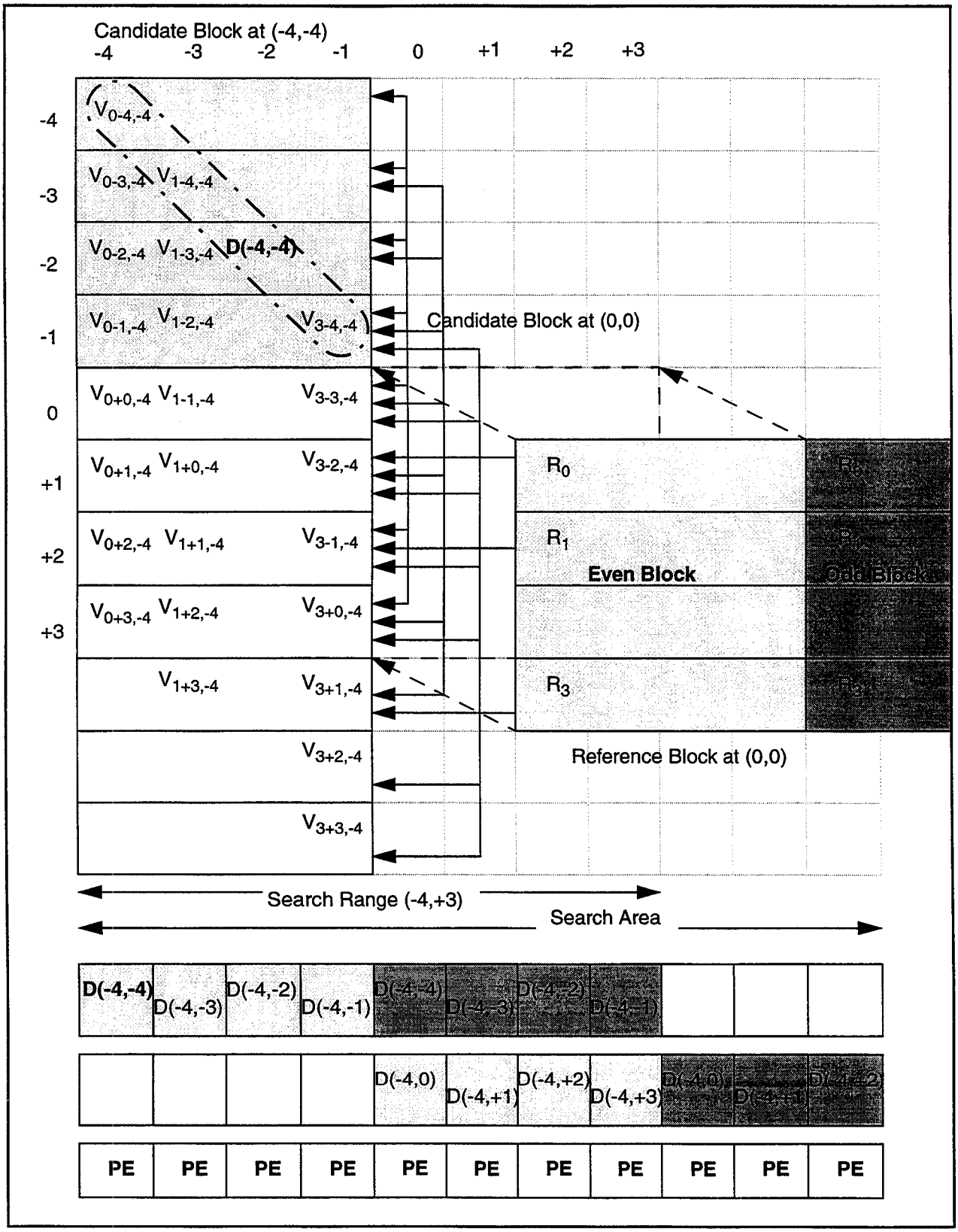


Figure 6.7 C*RAM implementation of FBMA

Parallelism can be exploited by computing all the $MAE(R_i,V_{i+k,l})$ distortions, say $MAE(R_0,V_{0+k,l})$ at once to save data transfer time (by left- and right-shiftings), and, later, accumulate the $MAE(R_0,V_{0+k,l})$, $MAE(R_1,V_{1+k,l})$, ..., and $MAE(R_3,V_{3+k,l})$ for a particular search point (k,l), say (-4,-4). The drawback of this parallelism is that extra memory is needed to store the individual $MAE(R_i,V_{i+k,l})$ for each D(k,l). The accumulation of $MAE(R_0,V_{0+k,l})$, $MAE(R_1,V_{1+k,l})$, ..., and $MAE(R_3,V_{3+k,l})$ is performed horizontally across the column, and the final distortion is stored in one location D(-4,-4), using a pre-stored mask. Similar distortion computation of the neighbouring reference block can also be performed at the same time. The respective distortions of the even and odd reference blocks are shown by differently shaded boxes.

It can be seen that due to data parallel nature, the distortion computation can be performed in parallel across the PE array. Improvement can be achieved by implementing the parallel addition/ subtraction capability at the PE, which results in a 32% reduction in distortion computation (Section 6.2.3). With the enhanced features, the absolute operations are avoided.

In the motion vector determination stage, the distortions are first row-wide (row by row) minimum searched within each PE. The resulting minimum in each column will "bubble down" to the bottom row where the minima of all the columns are stored. These minima will then be either column-wide (column-by-column) minimum searched using a mask with the bus-tie function, or nearest neighbour searched. The former requires simpler control, loading of masks - one per reference block - and thus, varies linearly with the number of reference blocks. The later requires elaborated control including data-shiftings and comparisons, but is independent of the number of reference blocks. Two execution times using different algorithms are recorded in Table 6.5 below:

Table 6.5 Minimum search time (ns)

Operations	column-wide using masks	nearest neighbour search
MV Determination	112,480 + k* 2,000	301,760

where k is the number of reference blocks that can be arranged across the PE array. By equating the two expressions, it can be derived that when the number of reference blocks is greater than 94, nearest neighbour search is preferable over column-wide search using masks. Therefore, when the frame size is small, such as QCIF and sub-QCIF, column-wide minimum search is used. Otherwise, nearest neighbour minimum search is employed.

Execution time for motion vector determination can be further reduced by applying the column-wide minimum search using the PSB. This additional feature of the enhanced C*RAM acts like a divider, separating groups of minima searches. First, the distortions of each even (or odd) blocks are rearranged onto the same rows, and column-wide minimum searches among blocks are conducted in parallel (Fig. 6.8).

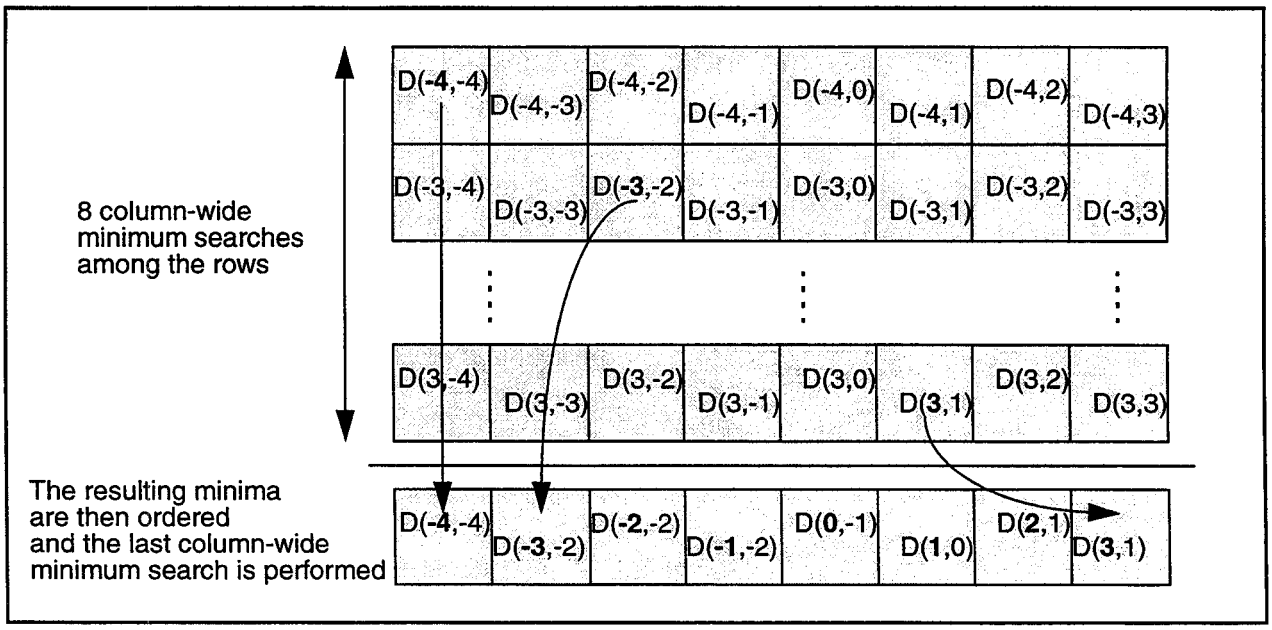


Figure 6.8 Column-wide minimum search using PSB

At any row, a '1' at a particular PE indicates that this PE has the minimum distortion among those who shared the same segmented bus. In cases where there are two or more minima (usually of the same value), any PE position, and thus, coordinate can be used. The minimum distortion of each

row will then be transferred to a column-ordered position of a new row, and parallel column-wide minimum search is performed one last time.

Table 6.6 shows an example of motion vector map after minimum search using a search range of (-4,+3). Minimum distortions, shown by the table, are found at search points: (-4,-4), (-3,-2), (-2,-2), (-1,-2), (0,-1), (1,0), (2,1), and (3,1).

The final result of the minimum search is shown in the last row. This row indicates that distortion is minimum at the 4th column, which corresponds to row k=-1 (due to the previous row-column distortion transfer). From row k=-1, it can be found that the minimum is located at the 3rd column which is column l=-2. Therefore, the motion vector is MV(-1,-2)[1].

Table 6.6 Motion vector map: an example

D(k,l)	l=-4	-3	-2	-1	0	+1	+2	+3
k=-4	1	0	0	0	0	0	0	0
-3	0	0	1	0	0	0	0	0
-2	0	0	1	0	0	0	0	0
-1	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0
1	0	0	0	0	1	0	0	0
2	0	0	0	0	0	1	0	0
3	0	0	0	0	0	1	0	0
D(-1,-2) min	0	0	0	1	0	0	0	0

With regard to the motion vector map, the determination of the DFP errors - without loss of generality, is a sequential process. Adjacent blocks may have the same motion vectors, whereas distant blocks may not. Parallel subtractions may be performed after the best candidate blocks and their corresponding reference blocks have been aligned.

1. The interpretation of motion vector has to be performed sequentially at the C*RAM controller.

Simulations have been performed on the 16 x 16 pixel blocks with search ranges of (-8,+7). Execution times are recorded per search of blocks which are arranged on the same row. Searches on different rows will be integer multiple of the recorded times. The results, using the baseline and the enhanced C*RAM's, are listed in Table 6.7:

Table 6.7 Block ME times using FBMA (ms)

Operations	Baseline	Enhanced	% Reduction
Distortion computation	19.446	16.896	13.1
MV determination	0.302	0.115	61.9

Due to the parallel addition/subtraction and PSB circuits, the distortion computation time has been reduced by 13.1%, while the motion vector determination time reduced by 61.9%. The reduction in computation time is not as high as 54% (Section 6.2.3, Table 6.4) because there involves data transfer and alignment in addition to MAE computation. These block ME times will later be used as a parameter for determining frame execution times based on frame sizes, mode of predictions, maximum displacement, and the number of available PEs.

6.2.5 The TSS Algorithm

Assuming that the search range is now (-7,+7) for the sake of presentation. According to [Koga81], the SA is first divided into nine smaller non-overlapping search areas. Thus, there are nine searches in the 1^{st} steps. Subsequently, there are only eight searches in the 2^{nd} and 3^{rd} steps since one search has already been performed in the previous step.

Let the search complexity of FBMA on a 16 x 16 pixel block be expressed as:

256 MAE computations + MV[1] determination + control complexity

Since the same set of operations can be applied across the reference blocks on the PE array, this complexity measure is a constant. On the other hand, the search complexity of TSS is:

1. MV stands for motion vector.

- Step1: 9 MAE computations are applied in parallel across the reference blocks;
- Step2: Since not all reference blocks have the same motion vector, MAE computations are performed sequentially;
- Step3: similar to step 2.

If the number of reference blocks is k, then the search complexity of TSS becomes:

9 MAE computations + k*16 MAE computations + MV determination + control complexity

The MV determination of the two approaches should be the same. If the control complexity of the sequential TSS is made comparable to that of the computational intensive, yet fully parallel FBMA, the TSS is more advantageous over FBMA only if:

$$9 + k*16 <= 256 \text{ or } k <= 15$$

In practice, the number of PEs is very large and so is k. Therefore, TSS is not advantageous over FBMA when implemented on the SIMD architecture like C*RAM.

6.2.6 Implementation of Pixel Decimation

On a RISC processor, the PD algorithm is expected to result in a speed up factor of 4. However, due to the SIMD nature of the C*RAM, the decimation by 2 in the horizontal direction cannot be saved because all the PEs are working whether on useful data or not. Alternating patterns, suggested by Liu [Liu93], is implemented by alternating the rows involved in the decimation process.

Time saving is achieved due to the reduction, by a factor of 2, in the distortion computation in the vertical (row-wide) direction, and the shorter word-length distortions which result in faster minimum search.

Simulations have been performed on the 16 x 16 pixel blocks with search ranges of (-8,+7). Execution times are recorded per search of blocks which are arranged on the same row. Searches on different rows will be integer multiples of the recorded times. The results, using the baseline and the enhanced C*RAM's are listed in Table 6.8:

Table 6.8 Block ME times using PD (ms)

Operations	Baseline	Enhanced	% Reduction
Distortion computation	10.086	9.267	8.1
MV determination	0.280	0.108	61.4

Using the enhanced C*RAM, the distortion computation time has been reduced by 8.1%, while the motion vector determination time reduced by 61.4%. Compared to FBMA, the reduction in distortion computation time is not the same since the number of rows in a block reduces.

In the following sections, implementations of the low complexity ME algorithms will be investigated. The FEXOR and BP-BPM differ only by the way frames are prefiltered, or the 8-bit to 1-bit transformation. The distortion computations, and motion vector determination, are similar to FBMA and PD, except that the working data have shorter word length. Therefore, only the prefiltering stage of each algorithm is discussed.

6.2.7 Implementation of FEXOR

As mentioned in Section 5.3, there are many morphological filters which can effectively detect the edges and textures of an image. Simulations have shown that the algorithm proposed by [Le98a] is the most efficient low-complexity ME algorithm while yielding highest quality among its class.

The image, stored in C*RAM, is first noise-reduced using opening-by-reconstruction with a filter of size 3 x 3. The resulting frame is next dilated using a 5 x 5 window[1] to generate a 2-pixel wide

1. Dilation using a 5 x 5 SE can be implemented as dilations using a 3 x 3 SE twice.

edge. The edge frame is finally obtained by subtracting the processed frame from its original and thresholding at grayscale level of 5.

The distortion computation is performed by sliding the binary reference block over the SA of the previous binary frame. Instead of performing MAE computation, the search operation is a simple XOR operation, where a result of 1 represents a mismatch, and 0 represents a match. The coordinates of the candidate block whose displacement results in a smallest distortion sum will be recorded as the motion vector of the reference block.

Simulations have been performed on the 16 x 16 pixel blocks with search ranges (-8,+7). The results, using the baseline and the enhanced C*RAM's, are listed in Table 6.9:

Table 6.9 Block ME times using FEXOR (ms)

Operations	Baseline	Enhanced	% Reduction
Prefiltering	0.490	0.490	0.0
Distortion computation	3.233	3.233	0.0
MV determination	0.215	0.090	58.1

There is no time savings in the prefiltering and distortion computation stages since there is no need for parallel addition and subtraction. Saving in MV determination time is achieved by the PSB on the enhanced C*RAM.

6.2.8 Implementation of BP-BPM

Common to all low-complexity ME algorithms, the pre-filtering stage is the one that differentiates one algorithm from another. The 1-b transform of BP-BPM is implemented on the C*RAM as follows:

```
/* for each row in the frame */
for(pixel=0;pixel<MAX_ROW;pixel++) {
    /* row wide */
    R1 = ADDu[I(pixel-4), I(pixel)];
    R2 = ADDu[I(pixel+8), R1];
    R3 = ADDu[I(pixel-8), R2];
    /* column wide */
    R1(pixel, j) = ADDu[R3(pixel, j+4), R3(pixel, j)];
    R2(pixel, j) = ADDu[R3(pixel, j+8), R1(pixel, j)];
    R4(pixel, j) = ADDu[R3(pixel, j-4), R2(pixel, j)];
    /* division: Mul PE(j) with c = [00001010] and scaled by 8 bits. */
    O(pixel) = MULu[R4(pixel), c];
}
```

Simulations have been performed on the 16 x 16 pixel blocks with search ranges (-8,+7). The results, using the baseline and the enhanced C*RAM's are listed in Table 6.10:

Table 6.10 Block ME time using BP_BPM (ms)

Operations	Baseline	Enhanced	% Reduction
Prefiltering	0.526	0.526	0.0
Distortion computation	3.233	3.233	0.0
MV determination	0.215	0.090	58.1

Similar to FEXOR, there is no time savings in the prefiltering and distortion computation stages since there is no need for parallel addition and subtraction. In the MV determination stage, the PSB provides a 58.1% reduction in execution time.

6.2.9 Comments on Different ME Algorithms

The block execution times of the 4 algorithms using baseline C*RAM are listed below.

Table 6.11 Block ME times on the baseline C*RAM (ms)

Operations	FBMA	PD	BP_BPM	FEXOR
Prefiltering	0.000	0.000	0.526	0.490
Distortion computation	19.446	10.086	3.233	3.233
MV determination	0.302	0.280	0.215	0.215
Total execution time	19.748	10.366	3.974	3.938

The FBMA is, as expected, the most time consuming algorithm. FBMA is, however, more efficient being implemented on the SIMD architecture like C*RAM. The PD is nearly twice as fast. PD will not performed as well in terms of reconstruction quality because the alternating patterns suggested by Lu cannot be realized efficiently in C*RAM. The decimation by a factor of 2 in the vertical direction (row-wide) fails when being applied to frames with horizontal patterns. Had the suggested patterns been implemented, execution time of PD will be as lengthy as the FBMA. PD is, therefore, not chosen for any further discussion.

The execution time of FEXOR is shorter than BP_BPM in terms of prefiltering. Moreover, the 3 x 3 window is flexible, and a 1-pixel pad is considerably small for all filter operations. On the other hand, the 16 x 16 kernel proposed in BP_BPM algorithm is rather fixed, and thus does not work well with algorithms requiring variable block sizes. An additional pad of 8 pixel wide is needed to get the correct filtered values for the boundary pixels.

In Section 6.1.4, the ratios of block ME times, assuming parallel processing, of FBMA, BP-BPM, and FEXOR were 2.2:1.2:1.0. The ratios are now 5.0:1.0:1.0, because more realistic models with data transfer have been used. The total execution times of both low-complexity algorithms are 5 times faster than the FBMA.

Table 6.12 Block ME times among C*RAM designs and data configurations

Algorithm	Baseline (ms)	Enhanced (ms)	% Reduction
FBMA	19.748	17.011	13.9
FEXOR	3.938	3.813	3.2

Table 6.12 shows the comparisons in overall reduction in block execution times using the enhanced C*RAM. The block ME using FBMA is reduced by 13.9% when mapped onto the enhanced C*RAM, while its FEXOR is reduced by only 3.2%. The earlier has a larger reduction factor because it uses MAE more frequently than the later.

It is worth mentioning that the block ME time using FBMA implemented on the 60-ns HDPP (reviewed in Section 3.8) requires 30.1 ms when performing a 256-location search using a 16 x 16 block [Gealow97]. Had this algorithm been run using a 40 ns cycle, block FBMA time of the 2-D HDPP would have been 20 ms. Similar block FBMA time of 1-D baseline C*RAM also remains at 19.7ms.

Recall that the 3-input ALU of the HDPP's PE is similar to the 3-input ALU of C*RAM's PE, and the array is arranged in a 2-D fashion as opposed to the 1-D C*RAM PEs. The equal block FBMA time achieved by C*RAM has shown that the 1-D baseline C*RAM design is indeed more efficient than the 2-D HDPP.

In order to compare with the performance of a RISC processor, I/O timings have been included. Knowing that the controller can be matched to provide data and instructions over the 8-b bus at the rate of 25MHz, the block ME I/O time is: (16 x 16 x 8b) / (8b bus * 25MHz) = 10,240 ns. Similar FBMA operation has been run on the Sun SuperSPARC for 192.098 ms. With the adjustments to the I/O time, the speed-ups in block ME times compared to the corresponding SuperSPARC's values are listed in Table 6.13:

Table 6.13 C*RAM speed-ups in block ME times over a RISC processor using FBMA on both C*RAM versions

Algorithm	Baseline (ms)	Enhanced (ms)
FBMA	9.7	11.3

The following observations have been drawn:

- From Table 6.12, the algorithmic speed-up of FEXOR over FBMA is 5;
- From Table 6.13, the speed-up obtained when performing FBMA in C*RAM (as opposed to the SuperSPARC's) ranges from 9.7 to 11.3. This value can be considered as architectural speed-up;

- Therefore, the overall speed-up for block ME ranges from 5*9.7 to 5*11.3, or from 48.5 to 56.5.

When the number of CUs is scaled up to the number of macroblocks in an applications, the resulting speed-up should be the product of the algorithmic speed-up, the architecture speed-up, and the number of macroblocks.

In Sections 5.4, 5.5 and 6.2, only the block DCT, VLC, and ME times as opposed to the frame statistics in C*RAM have been recorded. The data loading and unloading times will be studied thoroughly in Chapter 7 where overall applications such as JPEG, MPEG's and H.261 compression standards are implemented.

6.3 Conclusions

The speed-ups of different algorithms have been augmented by the C*RAM concept itself along with its proposed enhanced features. For block ME time, unlike algorithmic speed-ups, which is in the order of a few times, the combined algorithmic and architectural values range in the order of a few tens (from 49 to 57). The equal block ME execution times obtained by both C*RAM and HDPP have also demonstrated that the 1-D array C*RAM is more efficiently designed and programmed than the 2-D HDPP.

In the next chapter, overall implementation of image processing components such as filterings, DCT, VLC, and ME will be put together for the realizations of image/video compression standards using C*RAM.

CHAPTER 7

C*RAM Implementations of Image and Video Compression Standards

Building-block compression operations have been investigated in Chapters 5 and 6. In this chapter, these operations will be put together on a larger context for the implementations of the image and video compression standards. Section 7.1 investigates the real-time possibility of realizing JPEG image compression standard on C*RAM in two modes: baseline sequential and lossless. Section 7.2 focuses on a generic approach to C*RAM design targeting video compression standards: H.261/3, and MPEG-2, given the memory specifications and implementation requirements. The goal is to achieve fast, if not real-time compression with minimum degradation, or equivalently, a small increase in the compressed bit stream. Finally in Section 7.3, modifications to the C*RAM interface will be proposed.

7.1 Implementations of the JPEG Image Compression Standard

Implementations will be presented from simple grayscale to more complex color image compressions. For grayscale image compression, there are three modules in baseline sequential mode: DCT, quantization, and VLC. The image data are first DCT transformed and quantized in C*RAM (Section 5.4). DC coefficients are differentially coded, while AC coefficients are run-length coded (Section 5.5). As a result, the output data stream sent back to the controller are the differences in DC values and the {RUN, nonzero AC coefficient} pairs. This fraction of data, referred to as *compression factor*[1] δ, is image dependent, typically ranges from 2 to 10% for lossy compression, and 30 to 50% for lossless compression.

It has been noted that the algorithms and data configurations selected depend on the number of available PEs (and their local memories). If the number of PEs is less than or equal to the number of 8 x 8 blocks, the 1 x n^2 configuration is chosen. If the number of PEs is significantly greater (by a factor of 8) than the number of 8 x 8 blocks, the n x n configuration is preferred.

7.1.1 JPEG Baseline Compression

In intraframe compression, the CU can be a PE, or a group of 8 PEs. If the number of CUs is matched with the number of 8 x 8 blocks in the image, single block coding time is achieved. The data loading and unloading times, however, are proportional to the image size and compression method used. No parallelism may be exploited at the memory port.

Let p be the number of b-bit pixels in an image, and assuming that there are enough PEs to perform compression in one block coding time, the total intraframe compression time, for grayscale baseline JPEG compression, is:

$$T_{intra} = T_{IO} + T_{preprocessing} + T_{dct} + T_{vlc} \tag{7.1}$$

1. Compression factor is the reciprocal of compression ratio.

where $T_{IO} = p \times (1 + \delta) \times 40ns$ corresponds to the I/O time for data loading and unloading using the 8-b 25 MHz C*RAM controller. δ is chosen to be 10%. $T_{preprocessing}$ includes the level-shifting time.

Since DCT is the dominant operation in intraframe compression, the worst compression time is experienced when applying Lee's algorithm to baseline C*RAM arranged in 1 x n^2 configuration, while the best compression time is achieved when applying Cho's algorithm to enhanced C*RAM arranged in n x n configuration. Table 7.1 shows various C*RAM intraframe execution times for various frame sizes.

Table 7.1 C*RAM intraframe execution times for grayscale frames (ms)

Frame size	Worst case	Best case
512 x 512	11.534 + 6.814	11.534 + 1.351
256 x 256	2.884 + 6.814	2.884 + 1.351
128 x 128	0.721 + 6.814	0.721 + 1.351

In Table 7.1, each intraframe execution time is composed of two terms: I/O time and processing time. I/O times are the same for images of the same sizes, while processing times are equal within each case (The number of CUs is made equal to the number of blocks in an image). Table 7.1 shows that, for large frame size, most of the execution time is indeed I/O time. I/O time can be reduced by enlarging bus width via memory interleaving [Patterson94] and increasing bus speed.

In the following analysis, C*RAM will be considered first, as a video processor chip where I/O overhead is included; and secondly, as a computer RAM module where the I/O overhead is excluded. These performance measures will be compared against the corresponding 50 MHz SuperSPARC simulation result[1]. On the SuperSPARC, a typical baseline JPEG compression on a 512 x 512 pixel grayscale Fruits image takes 3.18 seconds.

1. Source codes of JPEG baseline sequential and lossless modes are obtained from [WSjpeg].

Let *relative speed-up* be the RISC execution time over C*RAM execution time, while *effective speed-up* be the relative speed-up over the number of PEs per CU, Table 7.2 summarizes the relative and effective (in brackets) speed-ups of two C*RAM systems over two scenarios:

Table 7.2 Grayscale baseline JPEG: C*RAM speed-ups over RISC.

	Worst case	Best case
C*RAM with I/O overhead	178.3	246.8 (30.9)
C*RAM without I/O overhead	466.7	2,353.8 (294.2)

Speed-ups are generally from 2 to 3 orders of magnitudes. When C*RAM is employed as a video processor, speed-up ranges from 178 to 246. Since best case speed-up is based on the n x n configuration with n = 8, the effective speed-up is only 30 times. These effective speed-ups imply that fastest execution time is achieved when using best case scenario, while most efficient implementations are realized when using worst case scenario. For time-critical application, the best case, but less efficient, scenario is traded off for smaller compression time.

Speed-up increases significantly when using C*RAM as computer RAM module. It increases four-fold for worst case, while increases ten-fold for best case scenario.

Since execution times in both cases under various frame sizes remain below 33.3 ms, grayscale baseline JPEG compression can, therefore, be achieved in real-time.

Color Image Compression: As mentioned in Section 2.8.1, the gamma corrected RGB components are color-space converted, subsampled, and level-shifted before being processed. I/O and processing times are to be adjusted as follows.

- Three frames of data, corresponding to the three components, will be stored in C*RAM;
- Assuming 4:2:0 subsampling, the computation of every other C_b and C_r pixel in both horizontal and vertical directions can be avoided. C_b and C_r pixels can be stored alternatively on a single line. Also, at least one 8 x 8 pixel block of C_b and one 8 x 8 pixel block C_r should be

contained in one CU. This necessitates the containment of three 16 x 16 pixel blocks of RGB components in one CU.

In each CU, before color conversion, there are three 16 x 16 pixel blocks corresponding to the gamma corrected R'G'B' components. After color conversion, there will be one 16 x 16 pixel Y-block, one 8 x 8 pixel C_b-block, and one 8 x 8 pixel C_r-block.

Color-space conversion is basically matrix multiplication (Section B.1), which can be effectively performed in parallel across the PEs. The intraframe execution time becomes:

$$T_{intra} = T_{IO} + T_{preprocessing} + 3T_{dct} + 3T_{vlc} \tag{7.2}$$

where $T_{IO} = 3k \times (1 + \delta) \times 40ns$ corresponds to the I/O time for data loading and unloading using the 8-b 25 MHz C*RAM controller. δ is chosen to be 30% since color image has better compression compared to grayscale. $T_{preprocessing}$ includes color-conversion time in addition to level-shifting time. The JPEG intraframe execution times for color image are summarized below:

Table 7.3 JPEG intraframe execution times for color frames (ms)

Frame size	Worst case	Best case
512 x 512	32.506 + 21.043	32.506 + 4.654
256 x 256	8.126 + 21.043	8.126 + 4.654
128 x 128	2.032 + 21.043	2.032 + 4.654

On the SuperSPARC, a typical JPEG compression in the baseline sequential mode on a 512 x 512 pixel color Fruits image takes 5.63 seconds. Table 7.4 summarizes the relative and effective (in brackets) speed-ups of two C*RAM systems over two scenarios:

Table 7.4 Color baseline JPEG: C*RAM speed-ups over RISC.

	Worst case	Best case
C*RAM with I/O overhead	105.1	151.5 (18.9)
C*RAM without I/O overhead	267.5	1,209.7 (151.2)

Due to color-conversion, a group of 16 PEs is required to accommodate a 16 x 16 Y-block and 8 x 8 C_b- and C_r-blocks. This arrangement requires that DCT and VLC are performed three times as opposed to one time in grayscale scheme. Therefore, color baseline JPEG compression generally results in lower speed-ups compared to its grayscale implementation.

7.1.2 JPEG Lossless Compression

In lossless mode, compression is performed on pixel-based rather than block-based. A prediction value y of pixel X is predicted based on one of eight linear combinations of the previously encoded 3 pixels: A, B, and C.

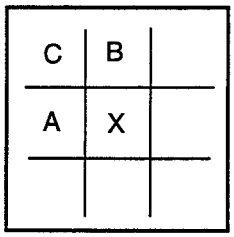


Figure 7.1 Prediction window in lossless mode

In this implementation, the prediction residual is defined as r = y - X, where y = A + B - C as shown in Fig. 7.1. The JPEG lossless execution time is summarized in Eq. 7.3:

$$T_{lossless} = T_{IO} + T_{preprocessing} + T_{predictive} + T_{vlc} \tag{7.3}$$

where compression factor is chosen to be 50%. Similar to baseline sequential mode, $T_{preprocessing}$ is the level-shifting time. T_{dct} is replaced by $T_{predictive}$ in lossless mode. Typical JPEG lossless execution times, composed of I/O time and processing time, are listed below:

Table 7.5 JPEG lossless execution times (ms)

Frame size	Nominal case
512 x 512	15.729 + 1.928
256 x 256	3.932 + 1.119
128 x 128	0.983 + 0.821

A typical JPEG lossless compression on a 512 x 512 grayscale Fruits image takes 9.65 seconds on the SuperSPARC [WSjpeg]. Speed-up comparisons are tabulated below:

Table 7.6 Lossless JPEG: C*RAM speed-ups over RISC.

	Nominal case
C*RAM with I/O overhead	76.5
C*RAM without I/O overhead	700.2

Speed-ups, in general, are 1 to 2 orders of magnitudes. Lossless compression has higher speed-ups because it involves pixel operations rather than local operations. Therefore, higher degree of parallelism is achieved.

There is no subsampling in color image lossless compression. The speed-up for color lossless compression should be similar to grayscale lossless compression.

Architectural parameters such as memory per PE, the number of PEs, the C*RAM Memory-Operate cycle, and the bus width and speed, the frame sizes, and maximum displacement (search range) are generally technology and application dependent. In the following section, a generic approach to the implementation of video compression standards will be presented.

7.2 C*RAM Model for Video Compression

Fast implementation requires data arrangement configuration which best matches the involved operations. Among the operations, ME is the most time consuming operation in video compression. Sections 5.4, 5.5 and 6.2 have been provided for single macro-block timing analysis which belong to many algorithms under the n x n configuration. This analysis did not include I/O times for loading actual frame data, and outputting the motion vector maps, the differences in DC values and the {RUN, nonzero AC coefficient} pairs. Moreover, extra time required to pad data for motion estimation on the sub-image boundaries must also be accounted for.

This section is devoted to a generic C*RAM design for video compression. The section starts with the design equations for local memory requirement per PE, the number of PEs on a C*RAM chip, and various I/O and execution times for I-, P-, and B-frames. Simulation models have been developed in accordance with the design considerations. The performance analysis is provided at the end with 2 video compression schemes: H.261, MPEG-2 Main Profile at Main Level (MP@ML). In order to facilitate the derivation process, specifications outlined in MPEG-2 video part have been chosen as design guideline.

7.2.1 Design Equations

Ideally, the number of CUs should be made equal to the number of blocks in a compression scheme. The memory per PE, in turn, should be allowed large enough to hold a number of frames at a certain time as well as extra working space for constants, intermediate results, and memory-map working registers.

The I/O path on the other hand, has contributed a significant delay to compression time. For instance, using a 8-b 25MHz bus, it takes 49.8 ms to store a 720 x 576 pixel MPEG-2 MP@ML color frame, directly from the video capture device. The compressed data also need to be sent off-chip. Assuming that the intended compression factor is 10%, an additional 5.0 ms is required to output the compressed data (in terms of motion vector maps, the differences in DC values, and the {RUN, nonzero AC coefficient} pairs). Thus, total frame I/O time is 54.8 ms which is nearly three times the block ME time.

In interframe compression, the CU is a group of 16 PEs. If the number of CUs is matched with the number of 16 x 16 pixel blocks in the frame, single block coding time can be achieved. The data loading and unloading times, however, are proportional to the image size and compression method used.

From Section 6.2, it takes an average of 19.7 ms to perform a full search ME operation using the (-8,+7) search range. If the proposed FEXOR algorithm is implemented, the execution time is reduced to 3.8 ms per block ME operation. For bi-directional ME required by B-frames, twice amount the block ME execution time is anticipated. Therefore, block ME time ranges from 7.6 ms to 39.4ms. It can be seen that block ME alone would exceed the real-time requirement (33.3 ms/ frame). This value can get larger if the search range is increased. Fortunately, not all frames are B-frames. There are I-frames where there is no ME computation, and P-frames where the ME computation is performed only once.

For a typical GOP, where N=9 and M=3, the intensive block ME time effectively reduces to 19.7ms*(1*0 + 2*1 + 6*2)/(1+2+6) = 30.6 ms for full search ME.

In the following sections, architectural parameters such as memory requirement per PE, the number of PEs in a C*RAM, and different I/O and coding times will be studied.

7.2.1.1 Memory Requirement and the Number of PEs in a C*RAM

In order to avoid transfers of frame data and intermediate results, reference frames should be retained in C*RAM until they are no longer needed. The above (N=9, M=3) GOP is sequenced below:

$I_1 \Rightarrow B_1 \Rightarrow B_2 \Rightarrow P_1 \Rightarrow B_3 \Rightarrow B_4 \Rightarrow P_2 \Rightarrow B_5 \Rightarrow B_6 \Rightarrow I_2$

For proper processing, this GOP is re-ordered as follows:

$I_1 \Rightarrow P_1 \Rightarrow B_1 \Rightarrow B_2 \Rightarrow P_2 \Rightarrow B_3 \Rightarrow B_4 \Rightarrow I_2 \Rightarrow B_5 \Rightarrow B_6$

In this arrangement, I_1 is encoded first. Then P_1 is encoded based on reference frame I_1. Next, B_1 is encoded based on reference frames I_1 and P_1. Similarly, B_2 is encoded based on reference frames I_1 and P_1. It is essential that reconstructed I_1 and P_1 frames remain in C*RAM until the encoding of B_2 has been complete. After B_2 is encoded, I_1 is no longer required. It will be

replaced by P_2. P_2 is encoded based on reference frame P_1, while P_1 and P_2 are referenced by B_3 and B_4, and so on. Therefore, no more than 3 frames are required to be in the C*RAM at any instant in time. Fig. 7.2 shows a typical frame arrangement.

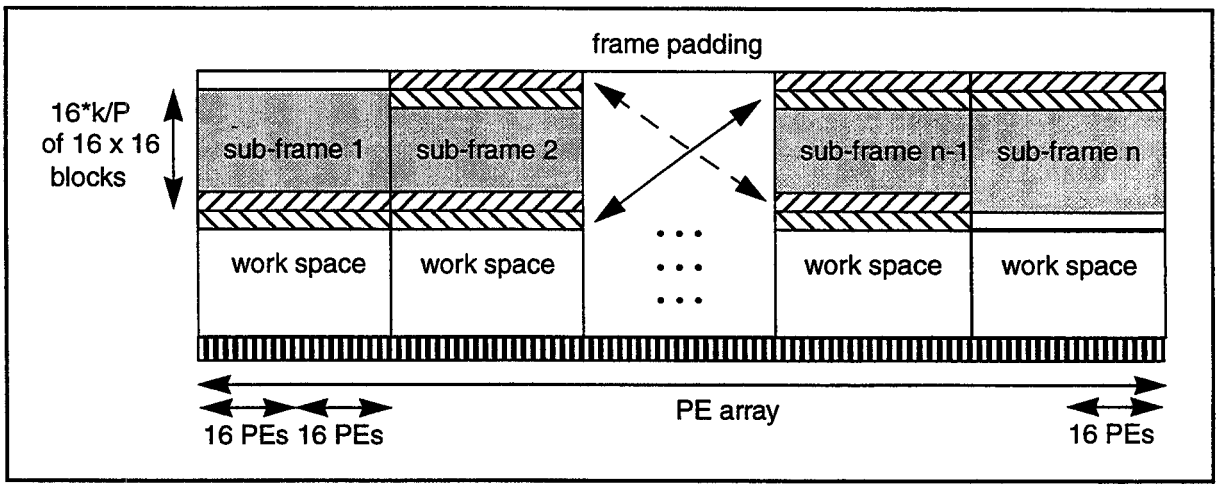


Figure 7.2 Partition of frame data for C*RAM processing

Frame data is first partitioned into horizontal sub-frames (or stripes) so that the maximum number of PEs can be utilized, and as a result, maximum parallelism can be achieved. Each 16 x 16 pixel block should be assigned to a CU of 16 PEs as specified by the n x n configuration. After color-space conversion and level-shifting, partial frame padding is performed by copying the upper-pad of the (i) luminance sub-frame, and appending it to the lower boundary of the (i-1) luminance sub-frame. In this manner, padding can be performed in parallel. Similarly, the lower pad of the (i+1) luminance sub-frame is copied and added to the upper boundary of the (i) luminance sub-frame. Some design parameters will be defined as follows.

Let Mem be the memory capacity of a RAM chip (expressed in bits),
P: the number of PEs, usually a power of 2,
c: the C*RAM Memory-Operate cycle, (expressed in ns),
bus_w, bus_s: the bus width and speed, respectively, at the C*RAM interface,
ρ : the intended compression factor,
row: number of lines in a frame,
col: number of pixels in a frame,

k: the number of macro-blocks = (row/16)*(column/16), as opposed to 1Kb =1024 bits,

d: the maximum displacement,

mode: unidirectional (mode=1) vs. bidirectional (mode=2)

S_o: classic C*RAM running at 25MHz;

S_n: C*RAM running at speed other than 25MHz;

T_{dct}: the block DCT execution time (Section 5.4)

T_{vlc}: the block VLC execution time (Section 5.5), and

T_{me}: the block ME execution time (Section 6.2).

The memory allocated to each PE should satisfy the following inequality:

$$\text{Mem/P} >= 24*[16k/(P/16)] + (\text{mode}+1)*2d*8 + 3\text{Kb}, \text{ where } P>=16*k \quad (7.4)$$

In Ineq. 7.4, the first term is the 24-b frame data which is partitioned into k blocks of 16 x 16 pixels, the second term is the 8-b padded data[1] required for non-overlapping ME, and the third term is allocated for working memory obtained from simulations in Chapters 5 and 6. Frame data required for DCT and ME are the same, and therefore, need to be stored only once, and they can be overwritten by the compressed data prior to VLC.

For H.261 using QCIF frame where d=8, mode=1, k=(176*144)/(16*16)=99, Inequality 7.4, after simplification, becomes Mem >= 608,256 + 3,328*P.

Since P, a power of two, should be made equal to or greater than 16*k, P is 2,048. Mem is, therefore, 5,879,808 bits. The closest memory module is 8Mb or 8,388,608 bits. For efficient RAM design, the next higher module is 16Mb[2] is chosen, and the number of PEs becomes P'=2*2,048.

1. ME is performed on the luminance component only.
2. RAM design specifications require equal number of row and column address lines [Prince91]. Therefore, the 8Mb (23-b address line) cannot be efficiently realized.

The amount of memory per PE is 16Mb/4096PE = 4Kb. When substituting the chosen values of P and Mem back to Ineq. 7.4, the relationship holds. Therefore, the most suitable processor-memory configuration for H.261 using QCIF is P=4,096 and Mem=16Mb.

For MPEG-2 MP@ML where d=16, mode=2, k=(720*576)/(16*16)=1620, P and Mem are similarly determined to be 64K and 256Mb, respectively. For HDTV where d=32, mode=2, k=(1920*1080)/(16*16)=8160, the most suitable configuration for HDTV implementation is P=128K and Mem=1Gb.

Processor-memory configurations for various video compression profiles are listed in Table 7.7:

Table 7.7 Optimal no. of PEs and the sizes of their local memories for various video compression profiles.

	Frame size	P	Mem/P	Mem
H.261	176 x 144	4K	4Kb	16Mb (2MB)
MPEG-2	720 x 576	64K	4Kb	256Mb (32MB)
HDTV	1920 x 1080	128K	8Kb	1Gb (128MB)

It can be seen that the memory capacity for the most demanding application such as HDTV remains 128MB.

It is also noted that an increase in 94 transistors for PE real-estate (Section 4.1) in the 4Kb/PE and 8Kb/PE versions result in 1 to 2% increase in final C*RAM area, respectively. This marginal increase drastically improves image/video processing operations as have been demonstrated.

In the following paragraphs, adjustments to block timings will be made. The block is 8 x 8 pixels for DCT and VLC, and 16 x 16 pixels for ME.

7.2.1.2 Computation of I/O Time

I/O time includes the followings: 24-b actual frame data loading time, time for outputting run-length coded (compressed) data, and time for outputting 1-b motion vector maps (one map per

block per direction). I/O time is inversely proportional to the bus width and bus speed, and is expressed below:

$$T_{I/O} = \{(1+\rho)*row*col*24 + k(4d^2+2d)*1\} / (bus\_w*bus\_s) \quad (7.5)$$

7.2.1.3 Computation of Padding Time

Padded data are required for ME at the sub-image boundaries. An extension of d-pixel pad is attached to the top and bottom of each sub-frame, where d is the maximum displacement in ME. Padding can be performed in parallel, described earlier in Fig. 7.2, is expressed below.

$$T_{padding} = 2*d*8*col*Operate\text{-}cycle*(S_o/S_n) \quad (7.6)$$

Padding is best performed using C*RAM's Operate-cycle, which is half the Memory-Operate cycle. Padding time may also be scaled with RAM with other speed S_n.

7.2.1.4 Adjustment to Block DCT Time

Due to the dominating ME operation, data is arranged in 16 x 16 pixel block. DCT, on the other hand, operates on 8 x 8 pixel block. Therefore, 2 block DCTs are, in effect, required. An additional T_{dct} is, however, needed for the transformation of the chrominance block.

$$T_{adct} = (2+1)*T_{dct}*(S_o/S_n) \quad (7.7)$$

7.2.1.5 Adjustment to Block VLC Time

Similar to T_{dct}, T_{vlc} is adjusted by a factor of (2+1) due to n x n arrangement of the 8 x 8 pixel block and the existing of the chrominance block.

$$T_{avlc} = (2+1)*T_{vlc}*(S_o/S_n) \quad (7.8)$$

7.2.1.6 Adjustment to Block ME Time

The obtained block ME time has been based on a maximum displacement of d=8. For other maximum displacements, block ME can be adjusted by $(2d/16)^2$. The T_{dfp} has been added to account for the computation of DFP (Section 6.2.4).

$$T_{ame} = [(2d/16)^2 * T_{me} + T_{pfd}] * (S_o/S_n) \tag{7.9}$$

7.2.2 Simulation Results and Performance Analysis

The general equations for different frame types are shown below:

$$T_{I\text{-}frame} = T_{I/O} + T_{preprocessing} + 2*T_{adct} + T_{avlc} \tag{7.10}$$

$$T_{P\text{-}frame} = T_{I/O} + T_{preprocessing} + 2*T_{adct} + T_{avlc} + T_{ame} \tag{7.11}$$

$$T_{B\text{-}frame} = T_{I/O} + T_{preprocessing} + T_{adct} + T_{avlc} + 2*T_{ame} \tag{7.12}$$

In Eq. 7.10, 7.11, and 7.12, the $T_{preprocessing}$ is composed of color-conversion, level-shifting, and padding times.

Recall that simulations on C*RAM using the n x n configuration result mainly in two scenarios for each of the DCT and ME operations. DCT is best (time wide) implemented using Cho's algorithm on the enhanced C*RAM, and worst implemented using Lee's algorithm on the baseline C*RAM. Likewise, ME is best implemented using FEXOR on the enhanced C*RAM, and worst implemented using FBMA on the baseline C*RAM. These parameters will be used to determine best and worst scenarios for the subsequent analysis.

7.2.2.1 Implementation of H.261

Simulations have been performed on the first 40 QCIF frames of Carphone sequence, already in YC_bC_r format, using d=8 and a compression factor of 10%. Using the source codes provided by [WSh263], the SuperSPARC takes, on the average, 1.435 seconds to compress a frame. The worst

and best processing times using C*RAM, excluding constant I/O time of 3.480 ms, are shown in Table 7.8:

Table 7.8 H.261 frame processing time (ms).

	Worst	Best
$T_{I\text{-frame}}$	10.609	8.287
$T_{P\text{-frame}}$	30.755	14.498

Worst case P-frame compression time of 30.755 + 3.480 = 34.235ms, which is near the 33.3ms real-time requirement, is quite comfortable for low 10 to 15 frame rate of H.261. Therefore, no additional efford is required if the derived 4K PE by 4Kb C*RAM version (Table 7.7) is used.

The H.261 source codes offer no facility for repeating I-frame as desired, I-frame is encoded only once at the beginning. Therefore, except for the first frame, the rest of the 39 frames are P-frames. Speed-ups are, therefore, computed with respect to P-frame compression time. Table 7.9 lists the C*RAM speed-ups over RISC counterpart with and without I/O considerations.

Table 7.9 H.261 simulations: C*RAM speed-ups over RISC.

	Worst	Best
$T_{P\text{-frame}}$ with I/O	42	90
$T_{P\text{-frame}}$ without I/O	47	99

Speed-ups are low because of first, there is no color-conversion which contributes a significant reduction if processed in C*RAM. Secondly, early-exit conditions for ME have been implemented in software. In fact, as specified in [WSh.263], search operation at a particular location is aborted if the intermediate distortion exceeds the distortion of an earlier motion vector in the same full search. This is most efficient if vectors of potentially low distortions are evaluated first.

The H.263 standard reduces bit rate with better quality mostly via SW implementations of the 4 optional modes: unrestricted ME, arithmetic coding, advance prediction, and PB frame.

Therefore, for H.263, the parallelizable operations such as DCT, RLC, and ME are performed on the C*RAM and the rest on the host processor.

7.2.2.2 Implementation of MPEG-2 MP@ML

Simulations have been performed on the first 28 frames of Football sequence of size 720 x 576 pixels, in Portable Pixel Map (PPM) format. Using source code provided by [WSmpeg], the SuperSPARC compresses 28 frames in 36 minutes and 3.84 seconds. The worst and best processing times using C*RAM, excluding constant I/O time of 55.32 ms, are shown below:

Table 7.10 MPEG-2 MP@ML frame processing times

	Worst	Best
$T_{I\text{-frame}}$	14.444	11.522
$T_{P\text{-frame}}$	98.637	29.455
$T_{B\text{-frrame}}$	59.396	26.065

Under this implementation, 3 complete GOP's in which 4 I-frames, 6 P-frames, and 18 B-frames are used. There is a need for color-conversion from PPM to YC_bC_r format. The maximum displacement d's are 16 and 8 for P- and B-frame, respectively. A complete 28-frame timing analysis is provided below:

Table 7.11 28-frame timing window using C*RAM (ms)

	Worst	Best
C*RAM with I/O	3,268	2,241
C*RAM without I/O	1,718	692

Table 7.11 shows that under worst case with I/O implementation, it takes more than 3 seconds to compress 28 frames using MPEG-2 MP@ML. Therefore, if real-time compression is targeted, modifications to the bus interface and computational load balancing must be addressed. Section 7.3 will provide some suggestions to performance improvement at the architecture level.

On the other hand, it takes less than a second to compress the same number of frames under best case without I/O consideration. This necessitates the implementation of the proposed FEXOR algorithm plus the parallel add/subtract circuits and one-to-many network enhancement. In such case, no additional efford is necessary if the derived 64K PE by 4Kb C*RAM version (Table 7.7) is used.

Table 7.12 compiles the C*RAM speed-ups over RISC counterpart with and without I/O considerations. Speed-ups are, due to the GOP requirement, computed with respect to the 28 frame period.

Table 7.12 MPEG-2 MP@MP simulations: C*RAM speed-ups over RISC

	Worst	Best
C*RAM with I/O	662.1	965.5
C*RAM without I/O	1,259.0	3,127.0

Compared to the speed-up values obtained in the H.261 simulations, the ones in this MPEG-2 simulations are, in general, one order of magnitude higher. There are also early-exist conditions for ME implemented by [WSmpeg], but they are not dominant contributing factors.

Recall that, there is no color-conversion in H.261 simulations, and there is in MPEG-2 simulations. Color-conversion, in this case, is unoptimized matrix multiplication[1] and is very inefficient on a RISC processor.

For a RISC processor, the amount of off-memory matrix multiplication time is directly proportional to the frame size. On the other hand, C*RAM, under the mentioned assumptions, can perform similar operations in parallel at the pixel level. Therefore, higher speed-ups are obtained.

1. DCT is optimized matrix multiplication.

7.3 Improvements to existing C*RAM architecture for better performance

Before discussing possible improvement to existing C*RAM, a minor observation may be made. If the SuperSPARC has 96MB of memory which helps speed-up its computation, the C*RAM system in consideration should be using the same amount of memory. In fact, C*RAM system in Section 7.2.2.2 uses only 256Mb or 32 MB of RAM. Therefore, if 3 modules of C*RAM are used and without any modification to the existing architecture and interface, the amount of speed-up should, theoretically, be increased by a factor of 3.

The C*RAM 8-b interface is another bottleneck regarding loading (unloading) data. This bus can be widen to 32-b while retaining its 8-b structure by interleaving four 8-b buses, and offsetting the output by one-quarter of a memory cycle. Another speed-up factor of 4 can be realized.

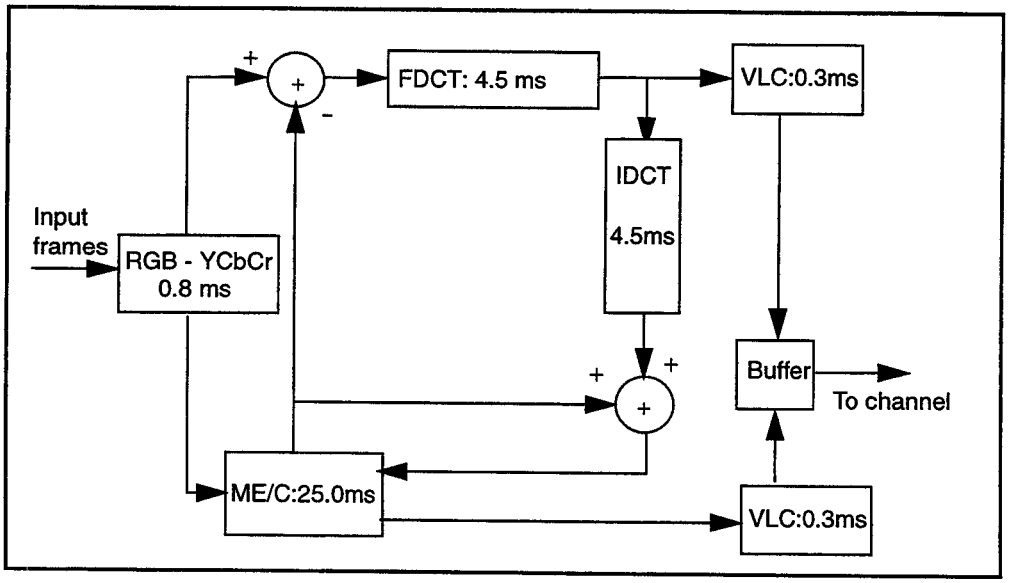


Figure 7.3 Computing load of a C*RAM-based video encoder

Finally, by balancing the computing load of a video encoder shown by Fig. 7.3, possible reduction in compression time can be achieved. It is known that ME, especially with large displacement vector d, is very time-consuming. Therefore, if ME/C is separated from the rest of the operations, compression time can be saved. This requires independent executions in two paths, which in turn, requires two PE arrays operating in parallel. Subsequently, this realization necessitates the dual-

port C*RAM: one port to accept uncompressed data and perform ME, the other to perform the rest of the operations and output compressed data.

Dual-port, particularly SRAM, results in 50% increase in silicon area [Silburt92]. For MPEG-2 standard, compression time will reduce by 30 to 40% depending on the compression profiles. From the C*RAM assembly development viewpoint, extra address lines can be used for a second set of COP bits. If 32-bit interleaved bus is implemented, there are also enough room for a second set of TTOP bits. The challenge relies on the synchronization of the two parallel operations on separate PE arrays, which is a decent topic for further research in C*RAM.

7.4 Conclusions

In Section 7.1, it has been shown that grayscale baseline and lossless JPEG compressions can be achieved in real-time. Baseline and lossless compression of color image of large size, i.e. 512 x 512, are limited by the 8-b interface to C*RAM, compression times are well under requirement, otherwise. Due to the smaller grain size of operations, pixel-based in lossless compression versus block-based in baseline lossy compression, lossless compression achieves higher speed-ups.

In Section 7.2.1, design parameters for different compression applications have been derived. For H.261 using QCIF format, a small 4K PEs by 4Kb C*RAM module is adequate. For MPEG-2 MP@ML, C*RAM module of larger number of PEs, 128K, and larger memory, 8Kb, is needed.

It is also noted that an increase in 94 transistors for PE real-estate (Section 4.1) in the 4Kb/PE and 8Kb/PE versions result in 1 to 2% increase in final C*RAM area, respectively. This marginal increase drastically improves image/video processing operations as have been demonstrated.

Performance analysis for H.261 and MPEG-2 MP@ML have been provided in Section 7.2.2. H.261, due to its small frame size and low processing requirement, can be achieved within timing

specifications. MPEG-2 compression, unless the proposed FEXOR algorithm plus the parallel add/subtract and one-to-many network enhancements are implemented, cannot make it in real-time.

Finally in Section 7.3, suggested modification to the C*RAM interface can provide an additional speed-up of 4, while balancing the computing load may reduce from 30 to 40% in compression time. As a result, extra effort will be required to add another PE array to the dual-port C*RAM, and synchronize its operations with its twin.

CHAPTER 8

Conclusions and Future Work

Various image and video processing algorithms and compression standards have been successfully implemented on the VHDL-based C*RAM models. The goals of executing visual computing and communications in C*RAM have been demonstrated so that:

- Memory bandwidth is fully utilized which is beneficial to memory- bound operations such as motion estimation;
- The emulation of arrays of SIMD processors allows natural mappings of image and video processing/compression algorithms onto the C*RAM;
- Variable word lengths are readily achievable through bit-serial operations. Furthermore, arithmetic and logic operations can be optimized at the bit level;
- Applications can be programmed at the controller level which alleviates the needs for specialized hardware;

8.1 Conclusions

Factors that allow improved CPU performance have been considered including:

- Increase in clock speed: The clock speed is technology dependent and, therefore, is dealt with by the device physicists and silicon engineers. With that in mind, the design of C*RAM hardware and C*RAM cycles are scalable (Sections 4.2 to 4.4). As the technology improves, C*RAM memory cycles get smaller and performance increases accordingly.
- Improvement of processor organization: C*RAM's have been designed and fabricated by the C*RAM design team. Issues such as operating power, operating frequency, and data bandwidth have extensively been discussed. The parallel add/sub circuits, proposed in this thesis (Section 4.1), allows improvement in MAE calculation by reducing computation time to more than 30%. This circuit is the first of its kind within the logic-in-memory SIMD community.
- Enhancement of its compiler: In software terms, frequently used instructions and macros have been optimized to reduce the number of cycles (Section 4.4). Even though manually parsed and interpreted, the proposed FEXOR algorithm (Section 6.1) has flattened the conventional block matching algorithm into three stages. With FEXOR, smart compilers can assign filter operations to comparison units, and distortion computations to integer/floating point units. The bus bottleneck in ME has been smoothen.

It has been demonstrated that *Visual Communications on a Memory-Embedded Array Processor: The Computational*RAM* is:

- Efficient: The 1-D array of 1-b PE of C*RAM outperforms the 2-D array of 1-b PEs of HDPP and the 1-D array of 8-b PEs of IMAP in filter, DCT, and ME operations (Sections 5.2.6, 5.3.6, and 6.2.9). Another example includes the real-time MPEG-2 MP@ML compression using only one 32MB C*RAM module (Section 7.2).

- Practical: The C*RAM design takes into consideration of existing DRAM modules. Additional logics form an array of PEs at the RAM's boundary without altering the compact and well-designed memory cells. Moreover, a moderate increase from 76 to 94 transistors (23.6% increase) in PE size (Section 4.1) has provided tremendous computing power to C*RAM and enlarged its range of applications.
- Economical: The C*RAM's concept, once recognized, will set a new stage for visual computing and communications. In fact, an increase in 94 transistors for PE real-estate (Section 4.1) in the 4Kb/PE and 8Kb/PE versions result in 1 to 2% increase in final C*RAM area, respectively. This marginal increase drastically improves image/video processing operations as have been demonstrated in Chapters 5, 6, and 7. Therefore, C*RAM is a low-cost solution to a media co-processor.

8.2 Future Works

Future works on C*RAM include the followings, but does not limit to:

- Since visual effects such as contrast stretching, edge detection, etc. are performed in real-time, image processing for office automation can be realized by embedding C*RAMs onto photocopiers and scanners;
- Standard compression techniques such as ME and DCT are best realized in C*RAM due to on-chip computing and massively parallel processing. These attractive realizations allow for the production of C*RAM-based MPEG cards for real-time compression;
- Enhancement to C*RAM interface is possible and the idea of load balancing necessitates the implementation of dual PE arrays, possibly, on the dual-port C*RAM; and finally
- Implementation of MPEG-4 video compression: C*RAM is able to segment the frame into objects of arbitrary shapes (as opposed to block-based) using the data independent morphological image processing operations.

Bibliography

STANDARDS

[ISO/IEC 10918-1] ISO/IEC JTC1, "Digital Compression and Coding of Continous Tone Still Images", Part 1, Requirements and Guidelines. Draft International Standard 10918, Nov. 1991.

[ISO/IEC 11172] ISO/IEC 11172, "Coding of Moving Pictures and Associated Audio - for Digital Storage Media at up to about 1.5Mb/s", 1991.

[ISO/IEC 13818-2] ISO/IEC 13818-2, "Generic Coding of Moving Pictures and Associated Audio", Video, Nov. 1993.

[ITU-T Rec. H.261] ITU-T Rec. H.261, "Video Codec for Audio-Visual Services at p x 64kbits/s", 1993.

[ITU-T Rec. H.263] ITU-T Rec. H.263, "Video Codec for Low-Bit Rate Communications", 1996.

WEBSITES

[WSh263] Sources for H.263, "ftp://bonde.nta.no: /pub/tmn/software".

[WSjpeg] Sources for JPEG, "ftp://ftp.uu.net: /graphics/jpeg/".

[WSmpeg] Sources for MPEG, "ftp://www.mpeg.org".

[WStree96] Sources for Tree-Search VQ, "ftp://isdl.ee.washington.edu: /pub/VQ/code/".

BOOKS and PAPERS

A

[Ahmed74] N. Ahmed, T. Natajan, and K. R. Rao, "Discrete Cosine Transform", *IEEE Transactions on Computers*, pp. 90-93, Jan. 1974.

[Aimoto96] Y. Aimoto, T. Kimura, Y. Yabe, "A 7.68GIPS 3.84GB/s 1W Parallel Image Processing RAM Integrating a 16Mb DRAM and 128 Processors", *ISSCC'96*, pp.372-373, 1996.

[Aizawa87] K. Aizawa, H. Harashima, and T. Saito, "Model-Based Synthetic Image Coding - Construction of a 3-D Model of a Person's Face", *Picture Coding Symposium '87*, Stockholm, pp.3-11, 1987.

[Athanas95] P. Athanas and A. Abbott, "Real-Time Image Processing on a Custom Computing Platform", *IEEE Computer*, pp.16-24, Feb. 1995.

B

[Balsara91] P.T Balsara and M.J Irwin, "Image Processing on a Memory Array Architecture", *Journal of VLSI Signal Processing*, pp.313-324, 1991.

[Berger71] T. Berger, *Rate Distortion Theory*, Prentice-Hall, Engelwood Cliffs, 1971.

[Booth51] A. D. Booth, "A Signed Binary Multiplication Technique", *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2), pp.236-240, 1951.

[Buzo80] A. Buzo, A. H. Gray, R. M. Gray, and J. D. Markel, "Speech Coding Based upon Vector Quantization", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-28, pp.562-574, Oct. 1980.

C

[Canny86] J. Canny, "A Computational Approach to Edge Detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8, pp. 679-698, Nov. 1986.

[Chan92] C.K. Chan and C. K. Ma, "A Fast Image Codevector Generation Method", *IEEE VSPC '92*, pp.68-73, 1992.

[Chan96] Y.L. Chan and W.C. Siu, "New Adaptive Pixel Decimation for Block Motion Estimation", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 1, Feb. 1996.

[Chen90] D. Chen and A. C. Bovik, "Visual Pattern Image Coding", *IEEE Transactions on Communications*, vol. 38, no. 12, pp.1662-1671, Dec. 1990.

[Chen66] G.L. Chen, J.S. Pan, and J.L. Wang, "Video Encoder Architecture for MPEG-2 Real Time Encoding", *IEEE Transactions on Consumer Electronics*, vol. 42, no. 3, pp.290-299, Aug. 1996.

[Chen77] W. H. Chen, C. H. Smith, and S. C. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform", *IEEE Transactions on Communications*, vol. COM-25, pp.1004-1009, Sept. 1977.

[Cho91] N. I. Cho and S. U. Lee, "Fast Algorithm and Implementation of 2-D DCT", *IEEE Transactions on Circuits and Systems*, 38(3), pp. 297-305, Mar. 1991.

[Chou89] C. H. Chou and Y. C. Chen, "A VLSI Architecture for Real-Time and Flexible Image Template Matching", *IEEE Transactions on Circuits and Systems*, vol. 36, no.10, pp.1336-1342, Oct. 1989.

[Cole93] B. C. Cole, "PC Makers Opt for Vector Quantization", *Electronic Engineering Times*, Nov. 1993.

[Cojocaru93] C. Cojocaru, M. Snelgrove, and D. G. Elliott, "A BATMOS Processing Element for 256Kb Computational RAM", *Internal Report*, Department of Electronics, Carleton University, 1993.

[Cojocaru94] C. Cojocaru, *Computational RAM: Implementation and Bit-Parallel Architecture, Master's Dissertation*, Department of Electronics, Carleton University, Dec. 1994.

D

[DeVos95] L. De Vos and M. Schobinger, "VLSI Architecture for a Flexible Block Matching Processor", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, no. 5, pp.417-428, Oct. 1995.

[Dinstein90] I. Dinstein, K. Rose, and A. Heiman, "Variable Block Size Transform Image Coder", *IEEE Transactions on Communications*, vol. 38, no. 11, pp.2073-2078, Nov. 1990.

[Dougherty92] E.R. Dougherty, *An Introduction to Morphological Image Processing*, SPIE Optical Engineering Press, 1992.

E

[Elliott92] D. G. Elliott, M. Snelgrove, and M. Stumm, "Computational-RAM: A Memory-SIMD Hybrid and Its Applications to DSP", *IEEE Custom Integrated Circuits Conference*, pp.30.6.1-30.6.4, Boston, May 1992.

[Equitz89] W. H. Equitz, "A New Vector Quantization Clustering Algorithm", *IEEE Transactions on Acoustics, Speech, and Signal Processing'89*, vol. 37, no. 10, pp.1568-1575, Oct. 1989.

[Elliott97] D.G. Elliott, M. Snelgrove, C. Cojocaru, and M. Stumm, "Computing RAM's for Media Processing", *SPIE Proceedings*, vol. 3021, pp.66-77, 1997.

F

[Falk76] H. Falk, "Reaching for the Gigaflop", *IEEE Spectrum*, pp.65-70, Oct. 1976.

[Farmwald92] M. Farmwald *et al.*, "A Fast Path to One Memory", *IEEE Spectrum*, pp. 50-51, Oct. 1992.

[Feig92] E. Feig and S. Winograd, "Fast Algorithms for the Discrete Cosine Transform", *IEEE Transactions on Signal Processing*, vol. 40, no. 9, pp. 2174-2193, Sept. 1992.

[Flanagan89] J. K. Flanagan, R. D. Morrell, R. L. Frost, C. J. Read, and B. E. Nelson, "Vector Quantization Codebook Generation Using Simulated Annealing", *Proceedings of ICASSP'89*, pp.1759-1762, 1989.

[Flynn72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, vol. 21, no. 9, pp. 940-960, 1972.

[Fok95] Y.H. Fok, O.C. Au, and R.D.Murch, "Novel Fast Block Motion Estimation in Feature Subspace", *Proceedings of ICIP'95*, vol. 1, pp.209-212, 1995.

[Fountain87] Fountain, *Processor Arrays: Architectures and Applications*, Academic Press, Harcourt Brace Jovanovich, Publisher, 1987.

[Foss92] R. Foss *et al.*, "Fast Interfaces for DRAMs", *IEEE Spectrum*, pp. 54-57, Oct. 1992.

[Foss96] R. Foss, "Implementing Application Specific Memory", *IEEE International Solid-State Circuits Conference'96*, Paper FP 16.1, 1996.

G

[Gealow96] J.C. Gealow, F. Herrmann, L. Hsu, and C. Sodini, "System Design for Pixel-Parallel Image Processing", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 1, pp.32-41, Mar. 1996.

[Gealow97] J.C. Gealow, *An Integrate Computing Structure for Pixel-Parallel Image Processing*, Ph.D. Dissertation, Massachusetts Institute of Technology, pp. 83, Jun. 1997.

[Gersho82a] A. Gersho, "On the Structure of Vector Quantizer", *IEEE Transactions on Information Theory*, vol. IT-28, pp.157-166, Mar. 1982.

[Gersho82b] A. Gersho and B. Ramamurthi, "Image Coding Using Vector Quantization", *IEEE International Conference on Acoustics, Speech, and Signal Processing'82*, pp.428-431, May 1982.

[Gjessing92] S. Gjessing *et al.*, "A RAM Link for High Speed", *IEEE Spectrum*, pp.52-53, Oct. 1992.

[Goldberg86a] M. Goldberg and H. Sun, "Frame Adaptive Vector Quantization for Image Sequence Coding", *IEEE Transactions on Communications*, vol. 36, no. 5, pp. 629-635, May 1986.

[Goldberg86b] M. Goldberg and H. Sun, "Image Sequence Coding Using Vector Quantization", *IEEE Transactions on Communications*, vol. COM-34, pp. 703-710, Jul. 1986.

[Gray84] R. M. Gray, "Vector Quantization", *IEEE ASSP Magazine*, pp. 4-29, Apr. 1984.

[Gray90] R. M. Gray, *Source Coding Theory*, Kluwer Academic Publishers, 1990.

[Gupta91] 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.

[Guo92] Q.Guo, N. M. Nasrabadi, and N. Mohesenian, "An Interframe Dynamic FSVQ Codec for Video Sequence Coding", *IEEE International Conference on Acoustics, Speech, and Signal Processing '92*, pp. III-501 - III-504, Mar. 1992.

H

[Habibi77] A. Habibi, "Survey of Adaptive Image Coding Techniques", *IEEE Transactions on Communications*, vol. COM-25, no. 11, pp.1275-1284, Nov. 1977.

[Hammerstrom96] D. Hammerstrom and D. Lulich, "Image Processing Using One-Dimensional Processor Arrays", *Proceedings of the IEEE*, vol. 84, no. 7, pp.1005-1018, Jul. 1996.

[Hardin99] R. W. Hardin, "Telemedicine Creates a New Kind of House Call", *SPIE OE - Reports*, Jan. 1999.

[Haskell98] B.G. Haskell, P.G. Howard, Y.A. LeCun, A. Puri, J. Ostermann, M. R. Civalar, L. Rabiner, L. Bottou, and P. Haffner, "Image and Video Coding - Emerging Standards and Beyond", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 7, Nov. 1998.

[He90] Y. He, Q. Zhang, Y. Ye, and Z. Li, "Vector Quantization by Neural Networks", *SPIE Medical Imaging IV: Image Capture and Display*, vol. 1232, pp.359-362, 1990.

[Hennessy94] J. Hennessy and D. Patterson, *Computer Organization and Design: The Hardware/ Software interface*, Morgan Kaufmann Publisher, 1994.

[Hertz92] L.Hertz and R.W. Schafer, "On the Use of Morphological Operators in a Class of Edge Detectors", *Computer Vision and Image Processing*, pp. 25-54, Academic Press, 1992.

[Higgins90] R. J. Higgins, *Digital Signal Processing in VLSI*, Prentice Hall, 1990.

[Huffman52] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", *Proceedings of IRE*, vol. 40, Sept. 1952.

[Huh94] 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 Communications and Image Processing '95*, vol. 1, pp.207-217, 1994.

[Hussain91] Z. Hussain, *Digital Image Processing: Practical Applications of Parallel Processing Techniques*, Ellis Horwood, 1991.

[Hwang84] K. Hwang, F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[Hwang93] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, and Programmability*, McGraw-Hill, 1993.

J

[Jain81a] A. K. Jain, "Image Data Compression: A Review", *Proceedings of the IEEE*, vol. 69, no. 3, pp.349-389, Mar. 1981.

[Jain81b] J. R. Jain and A. K. Jain, "Displacement Measurement and Its Application in Interframe Image Coding", *IEEE Transactions on Communications*, vol. COM-29, no. 12, pp.1799-1808, Dec. 1981.

[Jain89] A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, 1989.

[Jayant84] N. S. Jayant and P. Noll, *Digital Coding of Waveforms - Principles and Applications to Speech and Video*, Prentice-Hall, 1984.

[Jehng93] Y.S. Jehng, L.G. Chen, and T.D. Chiueh, "An Efficient and Simple VLSI Tree Architecture for Motion Estimation Algorithms", *IEEE Transactions of Signal Processing*, vol. 41, no.2, Feb. 1993.

[Johnson90] R.P. Johnson, "Contrast-based Edge Detection", *Pattern Recognition*, vol. 23, no.3-4, pp. 311-314, 1990.

[Jones92] F. Jones *et al.*, "A New Era of Fast Dynamic RAMs", *IEEE Spectrum*, pp.43-49, Oct. 1992.

[Jong94] H. M. Jong, L. G. Chen, and T. D. Chiueh, "Parallel Architectures for 3-Step Hierarchical Search Block-Matching Algorithm", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 4, no. 4, Aug. 1994.

K

[Kim91a] D. S. Kim and S. U. Lee, "Image Vector Quantization Based on Classification in the DCT domain", *IEEE Transactions on Communications*, vol. 39, no. 4, pp.549-556, Apr. 1991.

[Kim91b] D. S. Kim and S. U. Lee, "Classified Vector Quantizer Based on Minimum Distance Partitioning", *SPIE Visual Communications and Image Processing*, vol. 1605, pp. 190-201, 1991.

[Kim92] J. W. Kim and S. U. Lee, "A Transform Domain Classified Vector Quantization for Image Coding", *IEEE Transactions on Circuits and Systems*, vol. 2, no. 1, pp.549-556, Mar. 1992.

[Kim95] Y. Kim, C.S. Rim, and B. Min, "A Block Matching Algorithm with 16:1 Subsampling and Its Hardware Design", *ISCAS'95*, pp.613-616, 1995.

[Kobayashi97] K. Kobayashi *et al.*, "A Memory-Based Parallel Processor for Vector Quantization: FMPP-VQ", *IEICE Transactions on Electron*, vol. E80-C, no. 7, Jul. 1997.

[Kobayashi98] K. Kobayashi *et al.*, "An LSI for Low Bit-Rate Image Compression Using Vector Quantization", *IEICE Transactions on Electron*, vol. E81-C, no.5, May 1998.

[Koenen99] R. Koenen, "MPEG-4: Multimedia for Our Time", *IEEE Spectrum*, Feb. 1999.

[Koga81] T. Koga, K. Iinuma, A. Hirano, Y. Iijima, and T. Ishiguro, "Motion Compensated Interframe Coding for Video Conferencing", *Proc. Nat. Telecom. Conf.*, pp-G5.3.1-5.3.5, Nov. 1981.

[Kolagotla93], R. Kolagotla, S. S. Yu, and J. JaJa, "VLSI Implementation of a Tree Searched Vector Quantizer", *IEEE Transactions on Signal Processing*, vol. 41, no. 2, Feb. 1993.

[Krikelis96] A. Krikelis and T. Tawiah, "An Associative Massively Parallel Processor for Video Processing", *SPIE* vol. 2668, pp.222-232, 1996.

[Kudora98], I. Kudora and T. Nishitani, "Multimedia Processors", *Proceedings of the IEEE*, vol. 86, no. 6, Jun. 1998.

[Kung88] S. Y. Kung, *VLSI Array Processors*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

L

[Le94] T. M. Le, S. Panchanathan, and M. Snelgrove, "Computational*RAM Implementation of Vector Quantization for Image Compression", *Proceedings of IEEE Workshop on Visual Signal Processing and Communications '94*, pp.157-162, Sep. 1994.

[Le95] T. M. Le and S. Panchanathan, "Computational*RAM Implementation of an Adaptive Vector Quantization for Video Compression", *IEEE Transactions on Consummer Electronics* '95, vol. 41, no. 3, pp.738-747, Aug. 1995.

[Le96] T. M. Le, S. Panchanathan, and M. Snelgrove, "Computational*RAM Implementation of Mean-Average Scalable Vector Quantization for Real-Time Progressive Image Transmission", *Proceedings of the 1996 Canadian Conference on Electrical and Computer Engineering*, vol.1, pp.442-445, May 1996.

[Le97] T. M. Le, S. Panchanathan, and M. Snelgrove, "Computational*RAM Implementation of MPEG-2 for real-time encoding", *SPIE Proceedings of Multimedia Hardware Architectures'97*, vol. 3021, pp.182-193, Feb. 1997.

[Le98a] T. M. Le, M. Snelgrove, and S. Panchanathan, "Fast Motion Estimation Using Feature Extraction and XOR Operations", *SPIE Proceedings of Multimedia Hardware Architectures '98*, vol. 3311, pp.108-118, Mar. 1998.

[Le98b] T. M. Le, M. Snelgrove, and S. Panchanathan, "SIMD Processor Arrays for Image and Video Processing: A Review", *SPIE Proceeding of Multimedia Hardware Architectures '98*, vol. 3311, pp.30-41, Mar. 1998.

[Le99] T. M. Le and S. Panchanathan, "Low Complexity Block Motion Estimation Using Morphological Image Processing and XOR Operations", to be published in SPIE *Journal of Electronic Imaging*, 2000.

[Lee84] B. G. Lee, "A New Algorithm to Compute Discrete Cosine Transform", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-32, no. 6, pp.1243-1245, Dec. 1984.

[Lee87] J. Lee, R. Haralick, and L. Shapiro, "Morphologic Edge Detection", *Journal on Robotics Automation*, Ra-3, pp.142-156, Apr. 1987.

[Lee97] F. Lee, "A Parallel Implementation of the Sea Ice Tracking Algorithm for Coarse-Grained Multicomputers", *The Carleton Journal of Computer Science*, no 1, pp24-28, 1997.

[LeGall91] D. LeGall, "MPEG: A Video Compression Standard for Multimedia Applications", *Communications of the ACM*, vol. 34, pp.46-58, Apr. 1991.

[Li89] 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, Nov. 1989.

[Li94] W. Li and E. Salari, "Hybrid Vector Quantization of Video Sequences using Quadtree Motion Segmentation", *Proceedings of Visual Communications and Image Processing '94*, vol. 2308, pp.1383-1390, Sept. 1994.

[Linde80] Y. Linde, A. Buzo, and R. M. Gray, "An Algorithm for Vector Quantizer Design", *IEEE Transactions on Communications*, vol. COM-28, pp. 84-95, Jan. 1980.

[Linzer91] E. Linzer and E. Feig, "New Scaled DCT Algorithms for Fused Multiply/Add Architectures", *Proceedings ICASPP'91*, vol. 3, pp. 2201-2204, May 1991.

[Liu93] B. Liu and A. Zaccarin, "New Fast Algorithms for the estimation of block motion vectors", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 3, no. 2, pp.148-157, Apr. 1993.

[Loeffler89] C. Loeffler, A.Ligtenberg, and G. S. Moschytz, "Practical Fast 1-D DCT Algorithm with 11 Multiplications", *Proceedings ICASPP'89*, vol. 2, pp.988-991, Feb. 1989.

[Lu97] J. Lu and M.L. Liou, "A Simple and Efficient Search Algorithm for Block-Matching Motion Estimation", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 7, no. 2, Apr. 1997.

M

[Mallat89] S.G. Mallat, "A Theory for Multi-Resolution Signal Representation: the Wavelet Decomposition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol.11, no.7, pp.674-693, Jul. 1989.

[MacCalla80] J. R. MacCalla and M. V. Chang, "Multi-spectral Data Compression Using Hybrid Cluster Coding", *Proceedings of the AIAA Communication Satellite Systems Conference*, pp.404-407, 1980.

[McKenzie97] R. N. McKenzie, *A DRAM-based Parallel Processor for Real-Time Video*, Master's dissertation, Department of Electronics, Carleton University, Dec. 1997.

[Murakami82] T. Murakami, K. Asai, and E. Yamazaki, "Image Sequence Coding", *Electronics Letters*, vol. 18, no. 23, pp.1005-1006, Nov. 1982.

N

[Nam91] J. Y. Nam and K. R. Rao, "Image Coding Using a Classified DCT/VQ Based on Two-Channel Conjugate Vector Quantization", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 1, no. 4, pp.325-336, Dec. 1991.

[Nasrabadi88] N.M. Nasrabadi and R. A. King, "Image Coding Using Vector Quantization: A Review," *IEEE Transactions on Communications*, vol. COM-36, no. 8, pp. 957-971, Aug. 1988.

[Nasrabadi90] N. M. Nasrabadi and Y. Feng, "A Dynamic Finite State Vector Quantization Scheme", *IEEE*, pp. 2261-2264, 1990.

[Netravali79] A. N. Netravali and J. D. Robbins, "Motion Compensated Television Coding: Part I", *Bell Syst. Technical Journal*, vol. 58, pp. 631-670, Mar. 1979.

[Netravali80] A. N. Netravali and J. O. Limb, "Transform Picture Coding", *Proceedings of the IEEE*, vol. 68, no. 3, pp.366-407, Mar. 1980.

[Netravali89] A. N. Netravali and B. G. Haskell, *Digital Pictures: Representation and Compression*, Plenum Press, 1989.

[Nyasulu98] P. M. Nyasulu and W. M. Snelgrove, "Architecture and Implementation of a Computational RAM Controller", *IEEE International Conference on Massively Parallel Computing Systems*, Apr. 1998.

[Nyasulu99a] P. M. Nyasulu, R. Mason, W. M. Snelgrove, and D. G. Elliott, "Minimizing the Effect of the Host Bus on the Performance of a Computational RAM Logic-in-Memory Parallel Processing System", *IEEE 1999 Custom Integrated Circuits Conference*, pp. 631-634, May 1999.

[Nyasulu99b] P. M. Nyasulu, *System Design for Computational RAM Logic-in-Memory Parallel Processing Machine*, Ph.D. Dissertation, Carleton University, May 1995.

O

[Okazaki95] S. Okazaki, Y. Fujita, and N. Yamashita, "A Compact Real-Time Vision System Using Integrated Memory Array Processor Architecture", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, no. 5, Oct. 1995.

P

[Patterson96] D.A. Patterson, T. Anderson, and K. Yelick, "A Case for Intelligent RAM: IRAM", Hot chip 8, 1996.

[Patterson98] D.A. Patterson and J. L. Hennessy, *Computer Organization and Design: the Hardware / Software Interface*, 1998.

[Petajan95] E. Petajan, "The HDTV Grand Alliance System", *Proceedings of the IEEE*, vol. 83, no. 7, Jul. 1995.

[Pirsch93] P. Pirsch, *VLSI Implementations for Image Communications*, Elsevier, 1993.

[Pratt90] W. K. Pratt, *Digital Image Processing*, Academic Press, 1990.

[Price91] B. Prince, *Semiconductor Design - A Handbook of Design, Manufacture, and Application*, 2nd edition, John Wiley & Sons Ltd., 1991.

[Prince94] B. Prince, "Memory in the Fast Lane", *IEEE Spectrum*, pp.38-41, Feb. 1994.

R

[Ramamurthi86] B. Ramamurthi and A. Gersho, "Classified Vector Quantization of Images", *IEEE Transactions on Communications*, vol. COM-34, no. 11, pp. 1105-1115, Nov. 1986.

[Ramasubramanian92] 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, Mar. 1992.

[Rao90] K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, and Applications*, Academic Press, 1990.

[Riskin91] 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, Nov. 1991.

[Roese77] J. A. Roese, W. K. Pratt, and G. S. Robinson, "Interframe Cosine Transform Image Coding", *IEEE Transactions on Communications*, vol. COM-25, no. 11, pp.1329-1338, Nov. 1977.

[Rutledge87] C. W. Rutledge, "Variable Block DPCM: Vector Predictive of Color Images", *Proceedings of the IEEE*, vol. 1, pp. 130-135, Jun. 1987.

S

[Serra82] J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press, London, 1982.

[Silburt92] A. L. Silburt *et al.*, "A 200MHz 0.8um BiCMOS modular memory family of DRAM and multi-port SRAM's", *Proceedings of the IEEE Custom Integrated Circuits Conference 1992*, pp. 7.2.1-7.2.4, May 1992.

[Srinivasan85] R. Srinivasan and K. R Rao, "Predictive Coding Based on Efficient Motion Estimation", *IEEE Transactions on Communications*, vol. COM-33, pp. 888-896, Aug. 1985.

[Sunaga96] T. Sunaga, H. Miyatake, K. Kitamura, P. Kogge, and Eric Retter, "A Parallel Processing Chip with Embedded DRAM Macros", *IEEE Journal of Solid-State Circuits*, pp.1556-1559, vol. 31, no. 10, Oct.1996.

T

[Torrance98] R. Torrance *et al.*, "A 33GB/s 13.4Mb Integrated Graphics Accelerator and Frame Buffer", *ISSCC'98*, paper SA 21.5, 1998.

[Tou74] J. Tou and R. C. Gonzalez, *Pattern Recognition Principles*, Reading, MA: Addition-Wesley, 1974.

V

[Vaisey88] J. Vaisey and A. Gersho, "Simulated Annealing and Codebook Design", *IEEE Transactions on Acoustics, Speech, and Signal Processing '88*, pp.1176-1179, Apr. 1988.

[Vasudev95] B. Vasudev and Constantine, *Image and Video Compression Standards*, 1995.

[Verbeek88] P. W. Verbeek, H. A. Vrooman, and L. J. van Vliet, "Low-Level Image Processing by Max-Min Filters", *Signal Processing*, vol. 15, pp.249-258, 1988.

W

[Wallace91] G. K. Wallace, "The JPEG Still Picture Compression Standard", *Communications of the ACM*, vol. 34, no. 4, pp.30-44, Apr. 1991.

[Wang96] D. Wang, "A Multiscale Gradient Algorithm for Image Segmentation Using Watershreds", *IEEE Transactions on Pattern Recognition*, 1996.

[Weste94] N. Weste and K. Eshaghian, *Principles of CMOS VLSI Design*, 2nd edition, 1994.

[Witten87] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression", *Communications of the ACM*, vol. 30, no. 6, pp.520-540, Jun. 1987.

X

[Xie91] Z. Xie and T. G. Stockham, "Previsualized Image Vector Quantization with Optimized Pre and Postprocessors", *IEEE Transactions on Communications*, vol. 39, no. 11, pp .1662-1671, Nov. 1991.

Y

[Yan90] 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.

[Yamashita94] N. Yamashita, T. Kimura, Y. Fujita, and Y. Aimoto, "A 3.84 GIPS Integrated Memory Array Processor LSI with 64 Processing Elements and 2Mb SRAM", *ISSCC'94*, pp.202-203, 1994.

[Yeh87] C. L. Yeh, "Color Image Sequence Compression Using Adaptive Binary-Tree Vector Quantization with Codebook Replenishment", *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp.1059-1062, Apr. 1987.

APPENDIX A

Other Image Compression Issues

A.1 Data Compression Techniques

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

A.1.1 Lossless Compression

Lossless compression minimizes the average number of bits per pixel (bpp) without any loss in objective image quality. That is, the decoder reconstructs the exact original image from the encoded bit stream. Information theory states that the source can be exactly encoded with H bpp, where H is the *source entropy* [Jayant84]. For a source having 2^b possible independent symbols with probabilities p_i, $i=2^0, 2^1,..., 2^{b-1}$, the entropy is given by:

$$H = -\sum_{i=0}^{2^{b-1}} p_i \log_2 p_i \qquad \text{(A.1)}$$

In light of information theory, *variable length coders* (VLC), where shorter bit patterns are assigned to more probable source symbols and vice versa, have been established. *Huffman coding* [Huffman52] and *arithmetic coding* [Witten87] are among the most popular entropy coding techniques. While arithmetic coding achieves higher compression ratio compared to Huffman coding, it is more difficult to implement. Another lossless compression technique, applicable especially to binary image, is *run length coding* (RLC) [Jayant84] in which reduction in bit rate is achieved by grouping a non-zero value with a run of preceding zeros.

A.1.2 Lossy Compression

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

$$D = E\{d(X, \hat{X})\} \qquad \text{(A.2)}$$

where $d(X, \hat{X})$ is a distortion measure between the source X and its reproduction $\hat{X}$, and $d(X, \hat{X})$ can be of first-order as defined by *Mean Absolute Error* (MAE), or second-order as *Mean Square Error* (MSE). Clearly, the design of such an encoding scheme depends on the statistics of the source X and the characteristics of the distortion measure d.

It can be shown that R(D) is a monotonically non-increasing function of D (Fig. A.1). The minimum rate required for distortion-free compression of the source is the value of R at D = 0, and is equal to or greater than the source entropy, depending on the distortion measure.

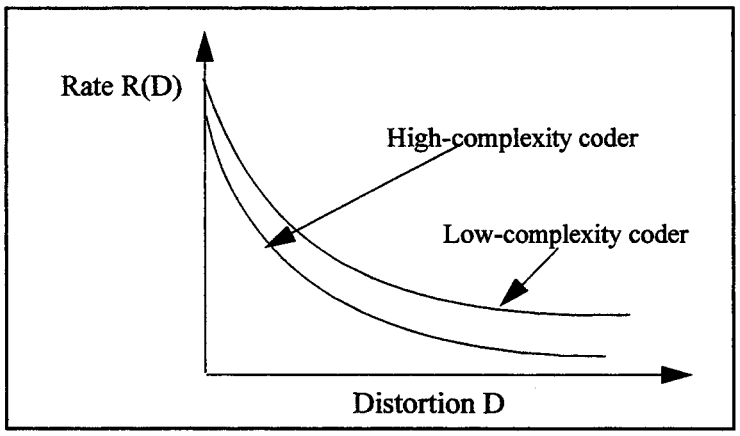


Figure A.1 Rate distortion curves and typical encoder performance

A.2 Objective Measures

Objective measures are sometimes referred to as distortion measures. The most commonly used distortion measures are Mean Square Error (MSE), Normalized MSE (NMSE), Mean Absolute Error (MAE), Signal-to-Noise ratio (SNR), and Peak Signal-to-Noise ratio (PSNR).

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

$$MSE = \frac{1}{XY}\sum_{k=1}^{X}\sum_{l=1}^{Y}(x_{kl}-\hat{x}_{kl})^2 \tag{A.3}$$

where x_{kl} and $\hat{x}_{kl}$ denote the original and reconstructed image pixels, respectively.

- Mean Absolute Error (MAE):

$$MAE = \frac{1}{XY}\sum_{k=1}^{X}\sum_{l=1}^{Y}|x_{kl}-\hat{x}_{kl}| \tag{A.4}$$

- Normalized Mean Square Error (NMSE):

$$NMSE = \left(\sum_{k=1}^{X}\sum_{l=1}^{Y}(x_{kl}-\hat{x}_{kl})^2\right) / \left(\sum_{k=1}^{X}\sum_{l=1}^{Y}x_{kl}^2\right) \tag{A.5}$$

- Signal-to-Noise Ratio (SNR):

$$SNR = 10 \cdot \log \frac{1}{NMSE} \tag{A.6}$$

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

$$PSNR = 10 \cdot \log \frac{(Peak-signal-value)^2}{MSE} \tag{A.7}$$

If the pixel is quantized to 8-b or 12-b, the peak signal value is 255 or 4095, respectively.

- PSNR': Sometimes, the coding performance is evaluated using PSNR' which is defined as:

$$PSNR' = 20 \cdot \log \frac{Peak-signal-value}{MAE} \tag{A.8}$$

NMSE generally provides a more precise judgement to a particular coding algorithm. On the other hand, many use PSNR as the performance indicator. PSNR is, however, very image dependent. Images with little details tend to have higher PSNR's, compared to those with many details, assuming the same coder is applied.

A.3 Codebook Generation for Vector Quantization

The key element in VQ is the design of a codebook which is a good representation of the image vectors. An optimal codebook, using the MSE criterion, must simultaneously satisfy two necessary conditions for optimality [Gersho82a].

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

$$R_i = \{v, |v - w_i| \le |v - w_m|, for \quad i \ne m\} \tag{A.9}$$

- The corresponding codewords are defined by:

$$w_i = E\{v|(v \in R_i)\} \tag{A.10}$$

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 means of the corresponding Voronoi regions under the MSE distortion measure. The K-mean [Gersho82a] or the closely related generalized Lloyd clustering algorithm proposed by Linde, Buzo, and Gray (LBG) [Linde80] is 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 techniques have been developed to obtain the initial codebook. In the first [Tou74], 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 2^n 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 are generated by preprocessing the training sequence. Two examples of this approach are parametric or diagonal seeding [Buzo80] where seeds are located along the diagonals of the Euclidean hyperspace, and mode-seeking seeding [MacCalla80] where seeds are located at the modes of the histogram.

Other techniques include: Simulated Annealing [Vaisey88, Flannagan89] which generates codebooks of better performance compared to LBG, but computation time is drastically increase;

Pairwise Nearest Neighbor (PNN) [Equitz89] which converges to an asymtotic value faster than LBG; and Maximum Descent (MD) [Chan92] which is 1 to 2 orders of magnitude faster and produces nearly 1dB better image quality compared to the LBG algorithm.

A.4 Edge Detection/Enhancement Techniques

Many edge detection/enhancement techniques have been proposed. Those proposed by [Canny86] and [Lee87] will be reviewed here as a supplement to the discussion in the thesis body.

[Canny86]: the preprocessing step consists of blurring the image with a Gaussian filter. The next step computes gradients in different directions for each pixel. The edge strength image is created by determining the maximum of these gradients at each point. The thresholding of this edge strength image is accomplished by selecting a single value at a predetermined percentage of the edge strength histogram. Finally, the edges are connected by selecting a lower threshold and iteratively adding pixels to the edge image.

In [Lee87], the original image is blurred by local averaging over a small square region of support. The blurred image is then eroded and dilated using a square structuring element. For each pixel (i,j), the resultant edge strength is the minimum of $(b_{ij}-e_{ij})$ and $(d_{ij}-b_{ij})$, where b_{ij}, d_{ij}, and e_{ij} are the grayscale values at pixel (i,j) in the blurred, dilated, and eroded images, respectively. The edge strength image is then thresholded to obtain the edge image.

A.5 Discrete Wavelet Transform (DWT)

Wavelet transform [Mallat89] offers a wide variety of useful features, in contrast to other transforms, such as Fourier transform or Cosine transform. Some of these features include no blocking effect[1], lower aliasing distortion for signal processing applications, and inherent scalability.

1. Due to this property, Wavelet has been proposed for an emerging standard called JPEG-2000.

DWT represents an arbitrary square integrable function[1] as superposition of a family of basis functions called *wavelets*. A family of wavelet basis functions can be generated by translating and dilating the mother wavelet corresponding to the family. The DWT coefficients can be obtained by taking the inner product between the input signal and the wavelet functions. Since the basis functions are translated and dilated versions of each other, a simpler algorithm, known as *Mallat's tree algorithm* or pyramid algorithm, has been proposed [Mallat89]. In this algorithm, the DWT coefficients of one stage can be calculated from the DWT coefficients of the previous stage, which is expressed as follows:

$$W_H(n,j) = \sum_m W_L(m,j-1)g(m-2n) \tag{A.11}$$

$$W_L(n,j) = \sum_m W_L(m,j-1)h(m-2n) \tag{A.12}$$

where $W_L(p,q)$ is the p^{th} scaling coefficient at the q^{th} stage, $W_H(p,q)$ is the p^{th} wavelet coefficient at the q^{th} stage, h(n) and g(n) are the dilation coefficients corresponding to the scaling and wavelet functions, respectively (Fig. A.2).

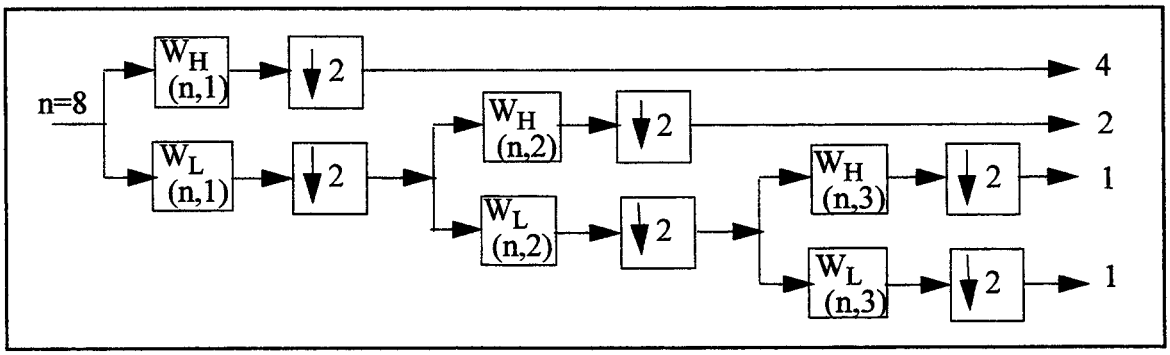


Figure A.2 Three-stage 1- D DWT decomposition using pyramid algorithm [Mallat89]

When computing the DWT coefficients of the discrete-time data, it is assumed that the input data represent the DWT coefficients of a high-resolution stage. Equations (A.9) and (A.10) can then be

1. A function s(t) is square integrable if the following expression exists: $\int s^2(t)dt < \infty$

used to obtain DWT coefficients of subsequent stages. In practice, this decomposition is performed only for a few stages. The dilation coefficients h(n) also represent a low-pass filter, while those of g(n) represent a high-pass filter. Therefore, DWT extracts information from the signal at different scales. The first level of wavelet decomposition extracts the details of the signal (high-frequency components) while the second and subsequent wavelet decompositions extract progressively coarser information (lower-frequency components).

In order to reconstruct the original data, the DWT coefficients are upsampled and passed through another set of low- and high-pass filters, which is expressed as:

$$W_L(n,j) = \sum_k W_L(k,j+1)h'(n-2k) + \sum_l W_H(l,j+1)g'(n-2l) \qquad \text{(A.13)}$$

where h'(n) and g'(n) are respectively the low- and high-pass synthesis filter corresponding to the mother wavelet. It is observed from (A.11) that the j^{th} level DWT coefficients can be obtained from $(j+1)^{th}$ level DWT coefficients.

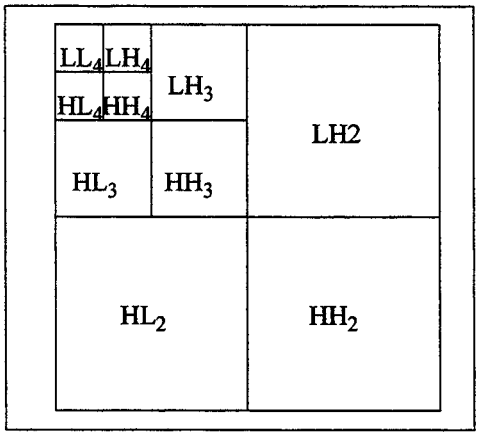


Figure A.3 Three-level 2-D wavelet decomposition

Similar to 2-D DCT, the 2-D DWT is commonly computed using row-column decomposition method. In the first level of decomposition (Fig. A.3), one low-pass subimage (LL_2) and three orientation selective high-pass subimages (LH_2, HL_2, and HH_2) are created. In the second level of decomposition, the low-pass subimage is further decomposed into one low- (LL_3) and three high-

pass subimages (LH_3, HL_3, and HH_3). This process is repeated on the low-pass subimage to derive the higher level of decompositions. In other words, DWT decomposes an image into pyramid structure of subimages with various resolutions corresponding to the different scales. The inverse wavelet transform is calculated in the reverse manner, i.e., starting from the lowest resolution subimages, the higher resolution images are calculated recursively.

Unlike 2-D DCT where operations are performed on the non-overlapping n x n blocks where n is usually small, 2-D DWT requires operations on the entire image. Therefore, 2-D DWT can be classified as global operation. To efficiently implement 2-D DWT, either a 1-D architecture having a global data transposer or a 2-D architecture is required.

APPENDIX B

Matrix Multiplication Using C*RAM

In this appendix, matrix multiplications on any SIMD architecture, especially C*RAM, will be presented via 2-D image transformation. An image transformation can be expressed as the products of matrix multiplications as follows:

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

where U and V are the input and output image blocks, respectively; A and B are the row and column operators, respectively.

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

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

By expanding the left hand size, we obtain:

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

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 trans-from coefficients is known in advance, they can also be ordered in any predetermined order. For 2-D DCT, this arrangement is called zig-zag order. The operation can be shown in Figure B.1 below:

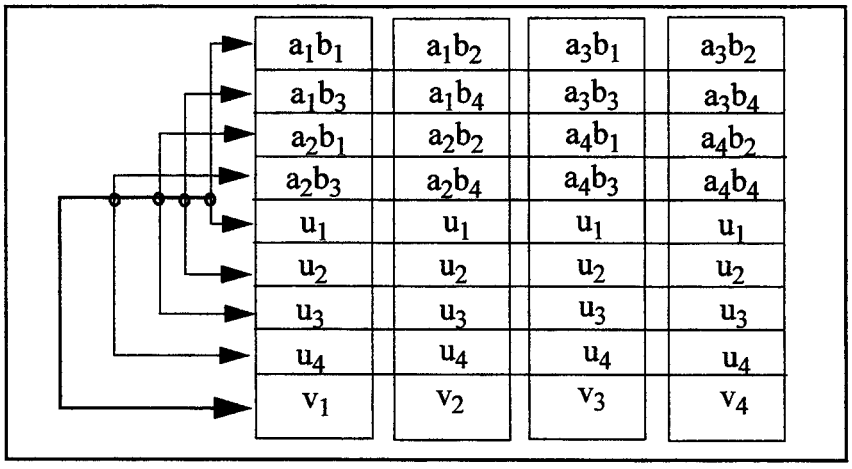


FIGURE B.1 Image Transformation implemented on a SIMD architecture

Matrix multiplications are applied in color image conversion, such as RGB-to-YCbCr conversion, and image compression such as DCT, SCT, and Hadamard transforms. The advantage of implementing matrix multiplication using SIMD is that the n^3 operations can be vectorized onto a n-PE array using n operations on the n precomputed constants. Operations can be further reduced by exploiting the sparseness of the coefficient matrices involved.

APPENDIX C

Program Listings

This appendix lists typical VHDL and .cmd (comand) files necessary for performance evaluations.

C.1 VHDL Files

```
-----------------------------------------------------
-- This file was first created in September 1996
--           and last modified in September 1999.
-- File name:pe.vhd;
-- Function:architecture of the processing element
-- By:      Thinh M. Le
-- Note:increase PE local memory from 2Kb to 4Kb
-----------------------------------------------------
library IEEE;
use work.all;
use std.TEXTIO.all;
use IEEE.std_logic_textio.all;
use IEEE.std_logic_1164.all;

-- external port
entity PE is
  port(ADDR_BUS: in STD_LOGIC_VECTOR(14 downto 0);
       DATA_BUS: in STD_LOGIC_VECTOR(7 downto 0);
       QW:    inout STD_LOGIC:='1'; -- value of WE register
       QM:    in STD_LOGIC :='0';-- value of internal read
       ALU_SR: in  STD_LOGIC;-- for internal write
       ALU_OP: inout STD_LOGIC:='0';-- for internal write
       ALU_SL: in  STD_LOGIC;-- for internal write
       TIE: in STD_LOGIC; -- bus_tie
       CCK: in STD_LOGIC; -- c*ram clock
       OPS: in STD_LOGIC; -- operate pulse
       R_CB: in STD_LOGIC);
end PE;

-- internal behavioral/structural
architecture BEHAVIORAL of PE is
  signal MYX: STD_LOGIC_VECTOR(2 downto 0);
  signal QX:  STD_LOGIC :='0';
  signal QY:  STD_LOGIC :='0';
  signal QS:  STD_LOGIC :='0';
  signal CCKdddd: STD_LOGIC:='0';
  type STD_LOGIC_TABLE is array (STD_LOGIC, STD_LOGIC) of STD_LOGIC;

-- truth table for "wired-pulldown function"
constant table_WIRED_PULLDOWN: STD_LOGIC_TABLE :=
-- ------------------------------------------------
-- |  U    X    0    1    Z    W    L    H    -   |
-- ------------------------------------------------
(('U', 'U', '0', '1', 'Z', 'U', 'U', 'U', 'U'), -- | U |
( 'U', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X'), -- | X |
( '0', '0', '0', '1', '0', '0', '0', '0', '0'), -- | 0 |
( '1', '1', '1', '1', '1', '1', '1', '1', '1'), -- | 1 |
( 'Z', 'X', '0', '1', '0', 'W', 'L', 'H', '-'), -- | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'W'), -- | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'L'), -- | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'H'), -- | H |
( 'U', 'X', '0', '1', '-', 'W', 'L', 'H', '-'));-- | - |

function WIRED_PULLDOWN(A:STD_LOGIC;B:STD_LOGIC) return STD_LOGIC is
    variable result: STD_LOGIC:='0';
  begin
    result := table_WIRED_PULLDOWN(A,B);
    return result;
  end WIRED_PULLDOWN;


  -- truth table for "wired-AND function"
  constant table_WIRED_AND: STD_LOGIC_TABLE :=
-- ------------------------------------------------
-- |  U    X    0    1    Z    W    L    H    -   |
-- ------------------------------------------------
(('U', 'U', '0', '1', 'Z', 'U', 'U', 'U', 'U'), -- | U |
( 'U', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X'), -- | X |
( '0', '0', '0', '0', '0', '0', '0', '0', '0'), -- | 0 |
( '1', '1', '0', '1', '1', '1', '1', '1', '1'), -- | 1 |
( 'Z', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'), -- | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'W'), -- | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'L'), -- | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'H'), -- | H |
( 'U', 'X', '0', '1', '-', 'W', 'L', 'H', '-'));-- | - |

function WIRED_AND(A: STD_LOGIC;B:STD_LOGIC) return STD_LOGIC is
    variable result: STD_LOGIC:='0';
  begin
    result := table_WIRED_AND(A,B);
    return result;
  end WIRED_AND;

begin
     CCKdddd  <= CCK'delayed(7 ns) and ADDR_BUS(5) and OPS;
-- legen to COP's
--    W  <= ADDR_BUS(0);
--    X  <= ADDR_BUS(1);
--    Y  <= ADDR_BUS(2);
--    SL <= ADDR_BUS(3);
--    SR <= ADDR_BUS(4);
--    BT <= ADDR_BUS(5);
--    S  <= ADDR_BUS(6);
--    SE <= ADDR_BUS(7);
--    T  <= ADDR_BUS(8);

   -- three inputs to the ALU
   MYX(0) <= QX;
   MYX(1) <= QY;
   MYX(2) <= QM xor WIRED_AND(QS,ADDR_BUS(7));

   WRITE_WE: process(CCK)
     begin
    if (CCK'event and CCK='1' and OPS='0' and R_CB='0') then
         if (ADDR_BUS(0)='1') then
           QW <= ALU_OP;
         else
           QW <= QW;
         end if;
       end if;
     end process WRITE_WE;

   WRITE_X: process(CCK)
     begin
    if (CCK'event and CCK='1' and OPS='0' and R_CB='0') then
         if    ADDR_BUS(1)='1' and ADDR_BUS(3)='0' then
           QX <= ALU_OP;
         elsif ADDR_BUS(1)='0' and ADDR_BUS(3)='1' then
           QX <= ALU_SL;
         else
           QX <= QX;
         end if;
       end if;
   end process WRITE_X;

   WRITE_Y: process(CCK)
     begin
    if (CCK'event and CCK='1' and OPS='0' and R_CB='0') then
         if    ADDR_BUS(2)='1' and ADDR_BUS(4)='0' then
           QY <= ALU_OP;
         elsif ADDR_BUS(2)='0' and ADDR_BUS(4)='1' then
           QY <= ALU_SR;
         else
           QY <= QY;
         end if;
       end if;
   end process WRITE_Y;

   WRITE_S: process(CCK)
     begin
    if (CCK'event and CCK='1' and OPS='0' and R_CB='0') then
         if ADDR_BUS(6)='1' then
           QS <= ALU_OP;
         else
           QS <= QS;
         end if;
       end if;
     end process WRITE_S;

   OPERATE: process(OPS,CCKdddd)
     begin
       if (OPS'event and OPS='1') then
         case MYX is
           when "000"  => ALU_OP <= DATA_BUS(0);
           when "001"  => ALU_OP <= DATA_BUS(1);
           when "010"  => ALU_OP <= DATA_BUS(2);
           when "011"  => ALU_OP <= DATA_BUS(3);
           when "100"  => ALU_OP <= DATA_BUS(4);
           when "101"  => ALU_OP <= DATA_BUS(5);
           when "110"  => ALU_OP <= DATA_BUS(6);
           when "111"  => ALU_OP <= DATA_BUS(7);
           when others => ALU_OP <= 'Z';
         end case;
       elsif (CCKdddd'event and CCKdddd='1') then
         ALU_OP <= WIRED_PULLDOWN(TIE,ALU_OP);
       end if;
   end process OPERATE;

end BEHAVIORAL;
------------------- end of pe.vhd -------------------
```

```
-----------------------------------------------------
-- This file was first created in September 1996
--           and last modified in September 1999.
-- File name:pe_seg.vhd;
-- Function:architecture of the processing element
-- By:    Thinh M. Le
-- Note:increase PE local memory from 2Kb to 4Kb
-----------------------------------------------------
library IEEE;
use work.all;
use std.TEXTIO.all;
use IEEE.std_logic_textio.all;
use IEEE.std_logic_1164.all;

-- external port
entity PE is
  port(ADDR_BUS: in STD_LOGIC_VECTOR(14 downto 0);
       DATA_BUS: in STD_LOGIC_VECTOR(7 downto 0);
       QW:    inout STD_LOGIC:='1'; -- value of WE register
       QM:    in STD_LOGIC :='0';-- value of internal read
       ALU_SR: in  STD_LOGIC;-- for internal write
       ALU_OP: inout STD_LOGIC:='0';-- for internal write
       ALU_SL: in  STD_LOGIC;-- for internal write
       TIE: in STD_LOGIC; -- bus_tie
       CCK: in STD_LOGIC; -- c*ram clock
       OPS: in STD_LOGIC; -- operate pulse
       R_CB: in STD_LOGIC);
end PE;

-- internal behavioral/structural
architecture BEHAVIORAL of PE is
  signal MYX: STD_LOGIC_VECTOR(2 downto 0);
  signal QX:  STD_LOGIC :='0';
  signal QY:  STD_LOGIC :='0';
  signal QS:  STD_LOGIC :='0';
  signal CCKdddd: STD_LOGIC:='0';
  type STD_LOGIC_TABLE is array (STD_LOGIC, STD_LOGIC) of STD_LOGIC;

-- truth table for "wired-pulldown function"
constant table_WIRED_PULLDOWN: STD_LOGIC_TABLE :=
-- ---------------------------------------------------
-- |  U    X    0    1    Z    W    L    H    -   |
-- ---------------------------------------------------
(('U', 'U', '0', '1', 'Z', 'U', 'U', 'U', 'U'), -- | U |
( 'U', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X'), -- | X |
( '0', '0', '0', '1', '0', '0', '0', '0', '0'), -- | 0 |
( '1', '1', '1', '1', '1', '1', '1', '1', '1'), -- | 1 |
( 'Z', 'X', '0', '1', '0', 'W', 'L', 'H', '-'), -- | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'W'), -- | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'L'), -- | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'H'), -- | H |
( 'U', 'X', '0', '1', '-', 'W', 'L', 'H', '-'));-- | - |

function WIRED_PULLDOWN(A: STD_LOGIC;B:STD_LOGIC) return STD_LOGIC is
    variable result: STD_LOGIC:='0';
  begin
    result := table_WIRED_PULLDOWN(A,B);
    return result;
  end WIRED_PULLDOWN;

-- truth table for "wired-AND function"
constant table_WIRED_AND: STD_LOGIC_TABLE :=
-- ---------------------------------------------------
-- |  U    X    0    1    Z    W    L    H    -   |
-- ---------------------------------------------------
(('U', 'U', '0', '1', 'Z', 'U', 'U', 'U', 'U'), -- | U |
( 'U', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X'), -- | X |
( '0', '0', '0', '0', '0', '0', '0', '0', '0'), -- | 0 |
( '1', '1', '0', '1', '1', '1', '1', '1', '1'), -- | 1 |
( 'Z', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'), -- | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'W'), -- | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'L'), -- | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'H'), -- | H |
( 'U', 'X', '0', '1', '-', 'W', 'L', 'H', '-'));-- | - |


function WIRED_AND(A: STD_LOGIC;B:STD_LOGIC) return STD_LOGIC is
    variable result: STD_LOGIC:='0';
  begin
    result := table_WIRED_AND(A,B);
    return result;
  end WIRED_AND;

begin
     CCKdddd  <= CCK'delayed(7 ns) and ADDR_BUS(5) and OPS;
-- legen to COP's
--    W  <= ADDR_BUS(0);
--    X  <= ADDR_BUS(1);
--    Y  <= ADDR_BUS(2);
--    SL <= ADDR_BUS(3);
--    SR <= ADDR_BUS(4);
--    BT <= ADDR_BUS(5);
--    S  <= ADDR_BUS(6);
--    SE <= ADDR_BUS(7);
--    T  <= ADDR_BUS(8);

   -- three inputs to the ALU
   MYX(0) <= QX;
   MYX(1) <= QY;
   MYX(2) <= QM.xor WIRED_AND(QS,ADDR_BUS(7));

   WRITE_WE: process(CCK)
     begin
    if (CCK'event and CCK='1' and OPS='0' and R_CB='0') then
         if (ADDR_BUS(0)='1') then
           QW <= ALU_OP;
         else
           QW <= QW;
         end if;
       end if;
    end process WRITE_WE;

   WRITE_X: process(CCK)
     begin
    if (CCK'event and CCK='1' and OPS='0' and R_CB='0') then
         if    ADDR_BUS(1)='1' and ADDR_BUS(3)='0' then
           QX <= ALU_OP;
         elsif ADDR_BUS(1)='0' and ADDR_BUS(3)='1' then
           QX <= ALU_SL;
         else
           QX <= QX;
         end if;
       end if;
    end process WRITE_X;

   WRITE_Y: process(CCK)
     begin
    if (CCK'event and CCK='1' and OPS='0' and R_CB='0') then
         if    ADDR_BUS(2)='1' and ADDR_BUS(4)='0' then
           QY <= ALU_OP;
         elsif ADDR_BUS(2)='0' and ADDR_BUS(4)='1' then
           QY <= ALU_SR;
         else
           QY <= QY;
         end if;
       end if;
    end process WRITE_Y;

   WRITE_S: process(CCK)
     begin
    if (CCK'event and CCK='1' and OPS='0' and R_CB='0') then
         if ADDR_BUS(6)='1' then
           QS <= ALU_OP;
         else
           QS <= QS;
         end if;
       end if;
     end process WRITE_S;

   WRITE_T: process(CCK)
     begin
    if (CCK'event and CCK='1' and OPS='0' and R_CB='0') then
         if ADDR_BUS(8)='1' then
           QT <= ALU_OP;
         else
           QT <= QT;
         end if;
       end if;
     end process WRITE_T;

    OPERATE: process(OPS,CCKdddd)
      begin
        if (OPS'event and OPS='1') then
          case MYX is
            when "000"  => ALU_OP <= DATA_BUS(0);
            when "001"  => ALU_OP <= DATA_BUS(1);
            when "010"  => ALU_OP <= DATA_BUS(2);
            when "011"  => ALU_OP <= DATA_BUS(3);
            when "100"  => ALU_OP <= DATA_BUS(4);
            when "101"  => ALU_OP <= DATA_BUS(5);
            when "110"  => ALU_OP <= DATA_BUS(6);
            when "111"  => ALU_OP <= DATA_BUS(7);
            when others => ALU_OP <= 'Z';
          end case;
        elsif (CCKdddd'event and CCKdddd='1') then
          ALU_OP <= WIRED_PULLDOWN(TIE,ALU_OP);
        end if;
   end process OPERATE;

end BEHAVIORAL;
------------------ end of pe_seg.vhd ----------------
```

```
------------------------------------------------------------------
-- This file was first created in September 1996
--          and last modified in September 1999.
-- File name:sram_cell.vhd;
-- Function:sram block
-- By:    Thinh M. Le
-- Note:increase PE local memory from 2Kb to 4Kb
------------------------------------------------------------------
library IEEE;
use work.all;
use std.TEXTIO.all;
use IEEE.std_logic_textio.all;
use IEEE.std_logic_1164.all;

-- external port
entity SRAM is
  port(WL: in STD_LOGIC_VECTOR(4095 downto 0); -- word line no.
       B:  in STD_LOGIC_VECTOR(7 downto 0);    -- memory bank no.
       BL: inout STD_LOGIC_VECTOR(63 downto 0);-- bit line
       ADDR_BUS: in STD_LOGIC_VECTOR(14 downto 0);  -- 15-bit bus
       DATA_BUS: inout STD_LOGIC_VECTOR(7 downto 0); -- 8-bit bus
       MCK: in STD_LOGIC;
       CCK: in STD_LOGIC;
       OPS: in STD_LOGIC;
       CS_A: in STD_LOGIC;
       CS_T: in STD_LOGIC;
       RD_WRB: in STD_LOGIC; -- read/write mode
       R_CB: in STD_LOGIC); -- RAM / C*RAM mode
end SRAM;

-- internal behavioral/structural
architecture BEHAVIORAL of SRAM is
  type CELL_2D is array (4095 downto 0,63 downto 0) of STD_LOGIC;
  type STD_LOGIC_TABLE is array (STD_LOGIC, STD_LOGIC) of STD_LOGIC;

  signal C_MEM: CELL_2D;

  signal WE: STD_LOGIC_VECTOR(63 downto 0);
  signal DM: STD_LOGIC_VECTOR(63 downto 0);
  signal TIE: STD_LOGIC_VECTOR(15 downto 0);
  signal BRK: STD_LOGIC_VECTOR(15 downto 0);

  signal CCKd:   STD_LOGIC;
  signal CCKdd:  STD_LOGIC;
  signal CCKddd: STD_LOGIC;
  signal SR_1: STD_LOGIC :='1';
  signal SR_0: STD_LOGIC :='0';
  signal SL_1: STD_LOGIC :='1';
  signal SL_0: STD_LOGIC :='0';

  -- truth table for "wired-or function"
  constant table_WIRED_OR: STD_LOGIC_TABLE :=
  -- ---------------------------------------------------
  -- |  U    X    0    1    Z    W    L    H    -   |
  -- ---------------------------------------------------
     (('U', 'U', '0', '1', 'Z', 'U', 'U', 'U', 'U'), -- | U |
      ('U', 'X', 'X', '1', 'X', 'X', 'X', 'X', 'X'), -- | X |
      ('0', 'X', '0', '1', '0', '0', '0', '0', '0'), -- | 0 |
      ('1', '1', '1', '1', '1', '1', '1', '1', '1'), -- | 1 |
      ('Z', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'), -- | Z |
      ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'W'), -- | W |
      ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'L'), -- | L |
      ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'H'), -- | H |
      ('U', 'X', '0', '1', '-', 'W', 'L', 'H', '-'));-- | - |

  -- truth table for "wired-pulldown function"
  constant table_WIRED_PULLDOWN: STD_LOGIC_TABLE :=
  -- ---------------------------------------------------
  -- |  U    X    0    1    Z    W    L    H    -   |
  -- ---------------------------------------------------
     (('U', 'U', '0', '1', 'Z', 'U', 'U', 'U', 'U'), -- | U |
      ('U', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X'), -- | X |
      ('0', '0', '0', '1', '0', '0', '0', '0', '0'), -- | 0 |
      ('1', '1', '1', '1', '1', '1', '1', '1', '1'), -- | 1 |
      ('Z', 'X', '0', '1', '0', 'W', 'L', 'H', '-'), -- | Z |
      ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'W'), -- | W |
      ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'L'), -- | L |
      ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'H'), -- | H |
      ('U', 'X', '0', '1', '-', 'W', 'L', 'H', '-'));-- | - |

  function WIRED_OR(A: STD_LOGIC_VECTOR) return STD_LOGIC is
    variable result: STD_LOGIC:='Z';
  begin
    for i in A'range loop
      result := table_WIRED_OR(result, A(i));
      exit when result = '1';
    end loop;
    return result;
  end WIRED_OR;

  function WIRED_PULLDOWN(A: STD_LOGIC_VECTOR) return STD_LOGIC is
    variable result: STD_LOGIC:='0';
  begin
    for i in A'range loop
      result := table_WIRED_PULLDOWN(result, A(i));
      exit when result = '1';
    end loop;
    return result;
  end WIRED_PULLDOWN;

  function WIRED_PULLDOWN(A: STD_LOGIC;B:STD_LOGIC) return STD_LOGIC
is
    variable result: STD_LOGIC;
  begin
    result := table_WIRED_PULLDOWN(A,B);
    return result;
  end WIRED_PULLDOWN;

  component PE1
    port(ADDR_BUS: in STD_LOGIC_VECTOR(14  downto 0);
         DATA_BUS: in STD_LOGIC_VECTOR(7 downto 0);
         QW: inout  STD_LOGIC; -- WE register
         QM: in  STD_LOGIC;-- value of internal read
                                -- from mem. to PE
         ALU_SR: in STD_LOGIC;-- can be value for internal write
         ALU_OP: inout STD_LOGIC;-- can be value for internal write
         ALU_SL: in STD_LOGIC;-- can be value for internal write
         TIE:  in STD_LOGIC;-- bus_tie
         CCK: in STD_LOGIC;
         OPS: in STD_LOGIC;
         R_CB: in STD_LOGIC);
  end component;

  component PE_SEG1
    port(ADDR_BUS: in STD_LOGIC_VECTOR(14  downto 0);
         DATA_BUS: in STD_LOGIC_VECTOR(7 downto 0);
         QW: inout  STD_LOGIC; -- WE register
         QM: in  STD_LOGIC;-- value of internal read
                                -- from mem. to PE
         ALU_SR: in STD_LOGIC;-- can be value for internal write
         ALU_OP: inout STD_LOGIC;-- can be value for internal write
         ALU_SL: in STD_LOGIC;-- can be value for internal write
         TIE:  in STD_LOGIC;-- bus_tie
         QT: inout STD_LOGIC; -- segment signal
         CCK: in STD_LOGIC;
         OPS: in STD_LOGIC;
         R_CB: in STD_LOGIC);
  end component;

  -- test the case 32 pe's
  -- order of PE: 00, 01, 02, ..,60, 61, 62, 63
  for all: PE1 use entity PE(BEHAVIORAL);
  for all: PE_SEG1 use entity PE_SEG(BEHAVIORAL);



begin
L00: PE1 port map
(ADDR_BUS,DATA_BUS,WE(0),BL(0),SR_0,DM(0),DM(1),TIE(0),CCK,OPS,R_CB);
L01: PE1 port map
(ADDR_BUS,DATA_BUS,WE(1),BL(1),DM(0),DM(1),DM(2),TIE(0),CCK,OPS,R_CB);
L02: PE1 port map
(ADDR_BUS,DATA_BUS,WE(2),BL(2),DM(1),DM(2),DM(3),TIE(0),CCK,OPS,R_CB);
L03: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(3),BL(3),DM(2),DM(3),DM(4),TIE(0),BRK(0),CCK,OPS
,R_CB);
L04: PE1 port map
(ADDR_BUS,DATA_BUS,WE(4),BL(4),DM(3),DM(4),DM(5),TIE(1),CCK,OPS,R_CB);
L05: PE1 port map
(ADDR_BUS,DATA_BUS,WE(5),BL(5),DM(4),DM(5),DM(6),TIE(1),CCK,OPS,R_CB);
L06: PE1 port map
(ADDR_BUS,DATA_BUS,WE(6),BL(6),DM(5),DM(6),DM(7),TIE(1),CCK,OPS,R_CB);
L07: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(7),BL(7),DM(6),DM(7),DM(8),TIE(1),BRK(1),CCK,OPS
,R_CB);

L08: PE1 port map
(ADDR_BUS,DATA_BUS,WE(8),BL(8),DM(7),DM(8),DM(9),TIE(2),CCK,OPS,R_CB);
L09: PE1 port map
(ADDR_BUS,DATA_BUS,WE(9),BL(9),DM(8),DM(9),DM(10),TIE(2),CCK,OPS,R_CB)
;
L10: PE1 port map
(ADDR_BUS,DATA_BUS,WE(10),BL(10),DM(9),DM(10),DM(11),TIE(2),CCK,OPS,R_
CB);
L11: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(11),BL(11),DM(10),DM(11),DM(12),TIE(2),BRK(2),CC
K,OPS,R_CB);
L12: PE1 port map
(ADDR_BUS,DATA_BUS,WE(12),BL(12),DM(11),DM(12),DM(13),TIE(3),CCK,OPS,R
_CB);
L13: PE1 port map
(ADDR_BUS,DATA_BUS,WE(13),BL(13),DM(12),DM(13),DM(14),TIE(3),CCK,OPS,R
_CB);
```

```
L14: PE1 port map
(ADDR_BUS,DATA_BUS,WE(14),BL(14),DM(13),DM(14),DM(15),TIE(3),CCK,OPS,R
_CB);
L15: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(15),BL(15),DM(14),DM(15),DM(16),TIE(3),BRK(3),CC
K,OPS,R_CB);

L16: PE1 port map
(ADDR_BUS,DATA_BUS,WE(16),BL(16),DM(15),DM(16),DM(17),TIE(4),CCK,OPS,R
_CB);
L17: PE1 port map
(ADDR_BUS,DATA_BUS,WE(17),BL(17),DM(16),DM(17),DM(18),TIE(4),CCK,OPS,R
_CB);
L18: PE1 port map
(ADDR_BUS,DATA_BUS,WE(18),BL(18),DM(17),DM(18),DM(19),TIE(4),CCK,OPS,R
_CB);
L19: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(19),BL(19),DM(18),DM(19),DM(20),TIE(4),BRK(4),CC
K,OPS,R_CB);
L20: PE1 port map
(ADDR_BUS,DATA_BUS,WE(20),BL(20),DM(19),DM(20),DM(21),TIE(5),CCK,OPS,R
_CB);
L21: PE1 port map
(ADDR_BUS,DATA_BUS,WE(21),BL(21),DM(20),DM(21),DM(22),TIE(5),CCK,OPS,R
_CB);
L22: PE1 port map
(ADDR_BUS,DATA_BUS,WE(22),BL(22),DM(21),DM(22),DM(23),TIE(5),CCK,OPS,R
_CB);
L23: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(23),BL(23),DM(22),DM(23),DM(24),TIE(5),BRK(5),CC
K,OPS,R_CB);

L24: PE1 port map
(ADDR_BUS,DATA_BUS,WE(24),BL(24),DM(23),DM(24),DM(25),TIE(6),CCK,OPS,R
_CB);
L25: PE1 port map
(ADDR_BUS,DATA_BUS,WE(25),BL(25),DM(24),DM(25),DM(26),TIE(6),CCK,OPS,R
_CB);
L26: PE1 port map
(ADDR_BUS,DATA_BUS,WE(26),BL(26),DM(25),DM(26),DM(27),TIE(6),CCK,OPS,R
_CB);
L27: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(27),BL(27),DM(26),DM(27),DM(28),TIE(6),BRK(6),CC
K,OPS,R_CB);
L28: PE1 port map
(ADDR_BUS,DATA_BUS,WE(28),BL(28),DM(27),DM(28),DM(29),TIE(7),CCK,OPS,R
_CB);
L29: PE1 port map
(ADDR_BUS,DATA_BUS,WE(29),BL(29),DM(28),DM(29),DM(30),TIE(7),CCK,OPS,R
_CB);
L30: PE1 port map
(ADDR_BUS,DATA_BUS,WE(30),BL(30),DM(29),DM(30),DM(31),TIE(7),CCK,OPS,R
_CB);
L31: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(31),BL(31),DM(30),DM(31),DM(32),TIE(7),BRK(7),CC
K,OPS,R_CB);

L32: PE1 port map
(ADDR_BUS,DATA_BUS,WE(32),BL(32),DM(31),DM(32),DM(33),TIE(8),CCK,OPS,R
_CB);
L33: PE1 port map
(ADDR_BUS,DATA_BUS,WE(33),BL(33),DM(32),DM(33),DM(34),TIE(8),CCK,OPS,R
_CB);
L34: PE1 port map
(ADDR_BUS,DATA_BUS,WE(34),BL(34),DM(33),DM(34),DM(35),TIE(8),CCK,OPS,R
_CB);
L35: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(35),BL(35),DM(34),DM(35),DM(36),TIE(8),BRK(8),CC
K,OPS,R_CB);
L36: PE1 port map
(ADDR_BUS,DATA_BUS,WE(36),BL(36),DM(35),DM(36),DM(37),TIE(9),CCK,OPS,R
_CB);
L37: PE1 port map
(ADDR_BUS,DATA_BUS,WE(37),BL(37),DM(36),DM(37),DM(38),TIE(9),CCK,OPS,R
_CB);
L38: PE1 port map
(ADDR_BUS,DATA_BUS,WE(38),BL(38),DM(37),DM(38),DM(39),TIE(9),CCK,OPS,R
_CB);
L39: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(39),BL(39),DM(38),DM(39),DM(40),TIE(9),BRK(9),CC
K,OPS,R_CB);

L40: PE1 port map
(ADDR_BUS,DATA_BUS,WE(40),BL(40),DM(39),DM(40),DM(41),TIE(10),CCK,OPS,
R_CB);
L41: PE1 port map
(ADDR_BUS,DATA_BUS,WE(41),BL(41),DM(40),DM(41),DM(42),TIE(10),CCK,OPS,
R_CB);
L42: PE1 port map
(ADDR_BUS,DATA_BUS,WE(42),BL(42),DM(41),DM(42),DM(43),TIE(10),CCK,OPS,
R_CB);
```

```
L43: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(43),BL(43),DM(42),DM(43),DM(44),TIE(10),BRK(10),
CCK,OPS,R_CB);
L44: PE1 port map
(ADDR_BUS,DATA_BUS,WE(44),BL(44),DM(43),DM(44),DM(45),TIE(11),CCK,OPS,
R_CB);
L45: PE1 port map
(ADDR_BUS,DATA_BUS,WE(45),BL(45),DM(44),DM(45),DM(46),TIE(11),CCK,OPS,
R_CB);
L46: PE1 port map
(ADDR_BUS,DATA_BUS,WE(46),BL(46),DM(45),DM(46),DM(47),TIE(11),CCK,OPS,
R_CB);
L47: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(47),BL(47),DM(46),DM(47),DM(48),TIE(11),BRK(11),
CCK,OPS,R_CB);

L48: PE1 port map
(ADDR_BUS,DATA_BUS,WE(48),BL(48),DM(47),DM(48),DM(49),TIE(12),CCK,OPS,
R_CB);
L49: PE1 port map
(ADDR_BUS,DATA_BUS,WE(49),BL(49),DM(48),DM(49),DM(50),TIE(12),CCK,OPS,
R_CB);
L50: PE1 port map
(ADDR_BUS,DATA_BUS,WE(50),BL(50),DM(49),DM(50),DM(51),TIE(12),CCK,OPS,
R_CB);
L51: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(51),BL(51),DM(50),DM(51),DM(52),TIE(12),BRK(12),
CCK,OPS,R_CB);
L52: PE1 port map
(ADDR_BUS,DATA_BUS,WE(52),BL(52),DM(51),DM(52),DM(53),TIE(13),CCK,OPS,
R_CB);
L53: PE1 port map
(ADDR_BUS,DATA_BUS,WE(53),BL(53),DM(52),DM(53),DM(54),TIE(13),CCK,OPS,
R_CB);
L54: PE1 port map
(ADDR_BUS,DATA_BUS,WE(54),BL(54),DM(53),DM(54),DM(55),TIE(13),CCK,OPS,
R_CB);
L55: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(55),BL(55),DM(54),DM(55),DM(56),TIE(13),BRK(13),
CCK,OPS,R_CB);

L56: PE1 port map
(ADDR_BUS,DATA_BUS,WE(56),BL(56),DM(55),DM(56),DM(57),TIE(14),CCK,OPS,
R_CB);
L57: PE1 port map
(ADDR_BUS,DATA_BUS,WE(57),BL(57),DM(56),DM(57),DM(58),TIE(14),CCK,OPS,
R_CB);
L58: PE1 port map
(ADDR_BUS,DATA_BUS,WE(58),BL(58),DM(57),DM(58),DM(59),TIE(14),CCK,OPS,
R_CB);
L59: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(59),BL(59),DM(58),DM(59),DM(60),TIE(14),BRK(14),
CCK,OPS,R_CB);
L60: PE1 port map
(ADDR_BUS,DATA_BUS,WE(60),BL(60),DM(59),DM(60),DM(61),TIE(15),CCK,OPS,
R_CB);
L61: PE1 port map
(ADDR_BUS,DATA_BUS,WE(61),BL(61),DM(60),DM(61),DM(62),TIE(15),CCK,OPS,
R_CB);
L62: PE1 port map
(ADDR_BUS,DATA_BUS,WE(62),BL(62),DM(61),DM(62),DM(63),TIE(15),CCK,OPS,
R_CB);
L63: PE_SEG1 port map
(ADDR_BUS,DATA_BUS,WE(63),BL(63),DM(62),DM(63),SL_0,TIE(15),BRK(15),CC
K,OPS,R_CB);


  ACCESS_DATA_BUS: process(CS_A,MCK,CS_T) -- reading in information
    variable c:  INTEGER range 0 to 10000 :=0;-- read input file
    variable cc: INTEGER range 0 to 10000 :=0;-- read input file
    variable bank: INTEGER range 0 to 7 :=0;
    variable row: INTEGER range 0 to 4095 :=0;
    variable V_DATA: STD_LOGIC_VECTOR(7 downto 0);
    variable V_TTOP: STD_LOGIC_VECTOR(7 downto 0);
    variable IP_LINE1: LINE;
    variable IP_LINE2: LINE;
    variable OP_LINE: LINE;
    variable dump_char: STRING(1 to 1) :=" ";

    file IP_FILE1: TEXT is in  "data_in.txt"; -- dct_vhdl.c
    file OP_FILE:  TEXT is out "data_out.txt";
    file IP_FILE2: TEXT is in  "ttop_final.txt"; -- 8-bit TTOP

    begin
      -- write data from a file to C*RAM
      if (CS_A'event and CS_A='1') then
        if (R_CB='1' and RD_WRB='0') then
          if c = 0 then
            READLINE(IP_FILE1,IP_LINE1);
          end if;
          READ(IP_LINE1,V_DATA);  -- to data bus
          DATA_BUS <= V_DATA;
```

```
        c := (c+1) mod 8;
      end if;

    -- memory external read
    elsif (MCK'event and MCK='1') then
      if (R_CB='1' and RD_WRB='1') then
        for row in 0 to 4095 loop
          if WL(row)='1' then
            for bank in 0 to 7 loop
              if B(bank)='1' then
                for bit in 0 to 7 loop
                  DATA_BUS(bit)  <= C_MEM(row,bit+8*bank);
                  WRITE(OP_LINE,C_MEM(row,bit+8*bank)); --@@
                end loop;
                WRITE(OP_LINE,dump_char);
                if bank = 7 then
                  --WRITE(OP_LINE,'R');
                  --WRITE(OP_LINE,row);
                  WRITELINE(OP_FILE,OP_LINE);
                end if;
              end if;
            end loop;
          end if;
        end loop;
      end if;

    -- C*RAM operation
    elsif (CS_T'event and CS_T='1') then
      if (R_CB='0') then
        READLINE(IP_FILE2,IP_LINE2);
        READ(IP_LINE2,V_TTOP);  -- to data bus
        DATA_BUS <= V_TTOP;
      end if;

    else
      DATA_BUS <= "ZZZZZZZZ";
    end if;
  end process ACCESS_DATA_BUS;

WR_TO_MEM: process(MCK)
  variable row: INTEGER range 0 to 4095;
  variable col: INTEGER range 0 to 63;
  begin
    -- external write to C*RAM
    if (MCK'event and MCK='1') then
      if (R_CB='1' and RD_WRB='0') then
        for row in 0 to 4095 loop
          if WL(row)='1' then
            for bank in 0 to 7 loop
              if B(bank)='1' then
                for bit in 0 to 7 loop
                  C_MEM(row,bit+8*bank) <= DATA_BUS(bit);
                end loop;
              end if;
            end loop;
          end if;
        end loop;
      end if;
    end if;

    -- internal write to C*RAM
    if (MCK'event and MCK='1') then
      if (R_CB='0' and RD_WRB='0') then
        for row in 0 to 4095 loop
          if (WL(row)='1') then
            for col in 0 to 63 loop -- @@@
              if WE(col)='1' then
                C_MEM(row,col) <= DM(col);  --- 7-bit?
              end if;
            end loop;
          end if;
        end loop;
      end if;
    end if;
  end process WR_TO_MEM;

--    legends to COP's
--    W  <= ADDR_BUS(0);
--    X  <= ADDR_BUS(1);
--    Y  <= ADDR_BUS(2);
--    SL <= ADDR_BUS(3);
--    SR <= ADDR_BUS(4);
--    BT <= ADDR_BUS(5);
--    S  <= ADDR_BUS(6);
--    SE <= ADDR_BUS(7);
--    T  <= ADDR_BUS(8);

  RD_FR_MEM: process(MCK)
    variable row: INTEGER range 0 to 4095;
    variable col: INTEGER range 0 to 63;
    begin
      -- internal read
      if (MCK'event and MCK='1') then
        if (R_CB='0' and RD_WRB='1') then
          for row in 0 to 4095 loop
            if(WL(row)='1') then
              for col in 0 to 63 loop
                BL(col) <= C_MEM(row,col);  -- 7-bit?
              end loop;
            end if;
          end loop;
        end if;
      end if;
  end process RD_FR_MEM;

  CCKd   <= CCK'delayed(4 ns) and ADDR_BUS(5) and OPS;
  CCKdd  <= CCK'delayed(5 ns) and ADDR_BUS(5) and OPS;
  CCKddd <= CCK'delayed(6 ns) and ADDR_BUS(5) and OPS;

  BUS_TIE: process(CCKd, CCKdd, CCKddd)
    variable result1: STD_LOGIC:='0';
    variable result2: STD_LOGIC:='0';
    variable result3: STD_LOGIC:='0';
    variable i,j,k,l: INTEGER;
    variable tmp: STD_LOGIC_VECTOR(15 downto 0);

    begin
      if (CCKd'event and CCKd='1') then
        for k in 0 to 7 loop
          for i in 0 to 1 loop
            result1 :='0';
            for j in 0 to 3 loop
--result1 := table_WIRED_PULLDOWN(result1,DM((k*8-1)-(4*i)-j));
              result1 := table_WIRED_PULLDOWN(result1,DM(k*8+4*i+j));
            end loop;
            tmp(2*k+i) := result1;  --@@
          end loop;
        end loop;
      end if;

      if (CCKdd'event and CCKdd='1') then
        for i in 0 to 14 loop
          if BRK(i)='1' then
            result2 :='0';
            for j in 0 to 1 loop
              result2 := table_WIRED_PULLDOWN(result2,tmp(i+j));
            end loop;
            tmp(i)   := result2;
            tmp(i+1) := result2;
          else
            tmp(i)   := tmp(i);
            tmp(i+1) := tmp(i+1);
          end if;
        end loop;
      end if;

      if (CCKddd'event and CCKddd='1') then
        for i in 14 downto 0 loop
          if BRK(i)='1' then
            result3 :='0';
            for j in 0 to 1 loop
              result3 := table_WIRED_PULLDOWN(result3,tmp(i+j));
            end loop;
            tmp(i)   := result3;
            tmp(i+1) := result3;
          else
            tmp(i)   := tmp(i);
            tmp(i+1) := tmp(i+1);
          end if;
        end loop;
      end if;

    TIE <= tmp;
  end process BUS_TIE;

end BEHAVIORAL;
-------------- end of sram_cell.vhd ---------------------------------


--------------------------------------------------------------------
-- This file was first created in September 1996
--           and last modified in September 1999.
-- File name:addr_dec.vhd;
-- Function:address decoder block
-- By:    Thinh M. Le
-- Note:increase PE local memory from 2Kb to 4Kb
--------------------------------------------------------------------
library IEEE;
use work.all;
use std.TEXTIO.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_textio.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
```

```
-- external port
entity ADDR_DEC is
  port(ADDR_BUS: buffer STD_LOGIC_VECTOR(14 downto 0);
       WL: out STD_LOGIC_VECTOR(4095 downto 0);
       B:  out STD_LOGIC_VECTOR(7 downto 0);
       MCK:    in STD_LOGIC;
       CS_A:   in STD_LOGIC;
       CS_T:   in STD_LOGIC;
       CS_D:   in STD_LOGIC;
       RD_WRB: in STD_LOGIC;  -- read/write mode
       R_CB:   in STD_LOGIC); -- RAM/C*RAM mode
end ADDR_DEC;

-- internal behavioral/structural
architecture BEHAVIORAL of ADDR_DEC is
begin
  ACCESS_ADDR_BUS: process(CS_A,CS_T)
    file IP_FILE1: TEXT is in "raddr_wr_final.txt";
    file IP_FILE2: TEXT is in "raddr_rd_final.txt";
    file IP_FILE3: TEXT is in "caddr_final.txt";
    file IP_FILE5: TEXT is in "cop_final.txt";
    variable IP_LINE1: LINE;
    variable IP_LINE2: LINE;
    variable IP_LINE3: LINE;
    variable IP_LINE5: LINE;
    variable V_ADDR1: STD_LOGIC_VECTOR(14 downto 0);
    variable V_ADDR2: STD_LOGIC_VECTOR(14 downto 0);
    variable V_ADDR3: STD_LOGIC_VECTOR(14 downto 0);
    variable   V_COP: STD_LOGIC_VECTOR(14 downto 0);

    begin
      if (CS_A'event and CS_A='1') then
        if (R_CB='1') then
          if (RD_WRB='0') then
            READLINE(IP_FILE1,IP_LINE1);
            READ(IP_LINE1,V_ADDR1);
            ADDR_BUS <= V_ADDR1;
          else
            READLINE(IP_FILE2,IP_LINE2);
            READ(IP_LINE2,V_ADDR2);
            ADDR_BUS <= V_ADDR2;
          end if;
        else
          READLINE(IP_FILE3,IP_LINE3);
          READ(IP_LINE3,V_ADDR3);
          ADDR_BUS <= V_ADDR3;
        end if;
      end if;

      if (CS_T'event and CS_T='1') then
        if (R_CB='0') then
          READLINE(IP_FILE5,IP_LINE5);
          READ(IP_LINE5,V_COP);
          ADDR_BUS <= V_COP;
        else
          ADDR_BUS <= "ZZZZZZZZZZZZZZZ";
        end if;
      end if;
    end process ACCESS_ADDR_BUS;

  ROW_DECODE: process(CS_D)
    variable s: INTEGER; -- read the addr. file for writing
    begin
      if (CS_D'event and CS_D='1') then
        s := CONV_INTEGER(ADDR_BUS(14 downto 3));
        for i in 0 to 4095 loop
          if i=s then
            WL(i) <= '1';
          else
            WL(i) <= '0';
          end if;
        end loop;
      end if;
    end process ROW_DECODE;

  BANK_DECODE: process(CS_D)
    variable t: INTEGER; -- read the addr. file for writing
    begin
      if (CS_D'event and CS_D='1') then
        t := CONV_INTEGER(ADDR_BUS(2 downto 0));
        for j in 0 to 7 loop
          if j=t then
            B(j) <= '1';
          else
            B(j) <= '0';
          end if;
        end loop;
      end if;
    end process BANK_DECODE;

end BEHAVIORAL;
----------------- end of addr_dec.vhd ------------------------------
```

```
-------------------------------------------------------------------
-- This file was first created in September 1996
--          and last modified in September 1999.
-- File name:t_bench.vhd;
-- Function:test bench for C*RAM
-- By:     Thinh M. Le
-- Note:increase PE local memory from 2Kb to 4Kb
-------------------------------------------------------------------
library IEEE;
use work.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity T_BENCH is
end T_BENCH;

architecture TEST of T_BENCH is
  constant MEM_SIZE: INTEGER :=  4095; -- actually 4096
  constant NUM_PE:   INTEGER :=    63; -- actually 64
  constant MCK_PER:  TIME    := 40 ns;
  constant CCK_PER:  TIME    := 20 ns;
  constant SETUP:    TIME    :=  5 ns;
  constant HOLD:     TIME    :=  0 ns;

  signal ADDR_BUS: STD_LOGIC_VECTOR(14 downto 0) :="ZZZZZZZZZZZZZZZ";
  signal DATA_BUS: STD_LOGIC_VECTOR(7 downto 0)  :="ZZZZZZZZ";
  signal B:        STD_LOGIC_VECTOR(7 downto 0); -- bank nunmber
  signal WL:       STD_LOGIC_VECTOR(MEM_SIZE downto 0);
  signal BL:       STD_LOGIC_VECTOR(NUM_PE downto 0);
  signal DUMMY1:   STD_LOGIC :='0';
  signal DUMMY2:   STD_LOGIC :='0';
  signal MCK:      STD_LOGIC :='0';
  signal CCK:      STD_LOGIC :='0';
  signal OPS:      STD_LOGIC :='0';
  signal CS_A:     STD_LOGIC :='0';
  signal CS_O:     STD_LOGIC :='0';
  signal CS_D:     STD_LOGIC :='0';
  signal CS_T:     STD_LOGIC :='0';
  signal RD_WRB:   STD_LOGIC :='1'; -- read/write mode
  signal R_CB:     STD_LOGIC :='0'; -- RAM/C*RAM mode

  component SRAM1
    port(WL:      in STD_LOGIC_VECTOR(MEM_SIZE downto 0);
         B:       in STD_LOGIC_VECTOR(7 downto 0);
         BL:      inout STD_LOGIC_VECTOR(NUM_PE downto 0);
         ADDR_BUS: in STD_LOGIC_VECTOR(14 downto 0);
         DATA_BUS: inout STD_LOGIC_VECTOR(7 downto 0);
         MCK:     in STD_LOGIC;
         CCK:     in STD_LOGIC;
         OPS:     in STD_LOGIC;
         CS_A:    in STD_LOGIC;
         CS_T:    in STD_LOGIC;
         RD_WRB:  in STD_LOGIC;
         R_CB:    in STD_LOGIC); -- RAM/C*RAM mode
  end component;

  component ADDR_DEC1
    port(ADDR_BUS: buffer STD_LOGIC_VECTOR(14 downto 0);
         WL:      out STD_LOGIC_VECTOR(MEM_SIZE downto 0);
         B:       out STD_LOGIC_VECTOR(7 downto 0);
         MCK:     in STD_LOGIC;
         CS_A:    in STD_LOGIC;
         CS_T:    in STD_LOGIC;
         CS_D:    in STD_LOGIC;
         RD_WRB:  in STD_LOGIC;
         R_CB:    in STD_LOGIC); -- RAM / C*RAM mode
  end component;

  for all: SRAM1     use entity SRAM(BEHAVIORAL);
  for all: ADDR_DEC1 use entity ADDR_DEC(BEHAVIORAL);

begin
  L1: SRAM1
port map(WL,B,BL,ADDR_BUS,DATA_BUS,MCK,CCK,OPS,CS_A,CS_T,RD_WRB,R_CB);
  L2: ADDR_DEC1
port map(ADDR_BUS,WL,B,MCK,CS_A,CS_T,CS_D,RD_WRB,R_CB);

  process(DUMMY1,DUMMY2)
    begin
      DUMMY1 <= not DUMMY1 after MCK_PER/2;
      CS_A   <= DUMMY1;              -- read address from a file
      CS_D   <= DUMMY1 after 2 ns;  -- decode this address
      MCK    <= DUMMY1 after SETUP; -- read/write operation
      CS_T   <= DUMMY1 after 10 ns; -- read TTOP from a file

      DUMMY2 <= not DUMMY2 after CCK_PER/2;
      CCK    <= DUMMY2 after 7 ns; -- operate on COP
  end process;
end TEST;
-------------------- end of t_bench.vhd ---------------------------
```

C.2 Test Files for VHDL models

```
--------------------------------------------------------------------
-- test.cmd
--------------------------------------------------------------------
comm init
cd t_bench
trace mck cck
trace l1/ops
trace rd_wrb r_cb
trace data_bus(7 downto 0)
trace addr_bus(12 downto 3) addr_bus(2 downto 0)
run 20
rd_op
wr_op
rd_op
wr_op
run 20
end init

comm hor_transfer8
cram
rd_op
rd_op
rd_op
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
rd_op
rd_op
end hor_transfer8


comm par_add_sub
mov8m
mov8m
add_sub8
end par_add_sub

comm seg_bus
cram
rd_op
max_min8
end seg_bus

comm sign_test
cram
rd_op
mov8
rd_op
mov8
end sign_test

comm s1
add_sub8
add_sub8
add_sub8
two8
two8
two8
acc12m
acc12m
acc12m
max_min12
end s1

comm ram_wr8
ram
wr
run 2560
end ram_wr8

comm ram_rd8
ram
rd
run 2560
end ram_rd8


comm add_sub8
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  rd_op
  wr_op
end
wr_op
end add_sub8


comm acc12m
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12
  rd_op
  rd_op
  wr_op
end
end acc12m

comm acc12mn
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  acc12m
end
end acc12mn

comm mul8c
foreach i in 1 2 3 4
  add_sub8
end
end mul8c

comm mul8
cram
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  add_sub8
end
end mul8

comm max_min8
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
  rd_op
end
wr_op
end max_min8


comm one8
cram
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
end one8

comm two8
cram
rd_op
wr_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
rd_op
end two8

comm mov8
cram
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
end mov8

comm mov8m
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
rd_op
end mov8m

comm lsh_rsh
cram
rd_op
rd_op
wr_op
end lsh_rsh

comm rd_op
rd
assign '0' ops
run 15
assign '1' ops
```

```
run 20
assign '0' ops
run 5
end comm

comm wr_op
wr
assign '0' ops
run 15
assign '1' ops
run 20
assign '0' ops
run 5
end comm

comm wr
assign '0' rd_wrb
end comm

comm rd
assign '1' rd_wrb
end comm

comm ram
assign '1' r_cb
end comm

comm cram
assign '0' r_cb
end comm

-- typical pe test procedure --
comm pe_test
trace /t_bench/l1/l00/alu_op
trace /t_bench/l1/l01/alu_op
trace /t_bench/l1/l02/alu_op
trace /t_bench/l1/l03/alu_op
trace /t_bench/l1/l04/alu_op
trace /t_bench/l1/l05/alu_op
trace /t_bench/l1/l06/alu_op
trace /t_bench/l1/l07/alu_op

trace /t_bench/l1/l00/myx(2)
trace /t_bench/l1/l01/myx(2)
trace /t_bench/l1/l02/myx(2)
trace /t_bench/l1/l03/myx(2)
trace /t_bench/l1/l04/myx(2)
trace /t_bench/l1/l05/myx(2)
trace /t_bench/l1/l06/myx(2)
trace /t_bench/l1/l07/myx(2)

trace /t_bench/l1/l00/qm
trace /t_bench/l1/l01/qm
trace /t_bench/l1/l02/qm
trace /t_bench/l1/l03/qm
trace /t_bench/l1/l04/qm
trace /t_bench/l1/l05/qm
trace /t_bench/l1/l06/qm
trace /t_bench/l1/l07/qm

trace /t_bench/l1/l00/qs
trace /t_bench/l1/l01/qs
trace /t_bench/l1/l02/qs
trace /t_bench/l1/l03/qs
trace /t_bench/l1/l04/qs
trace /t_bench/l1/l05/qs
trace /t_bench/l1/l06/qs
trace /t_bench/l1/l07/qs

trace /t_bench/l1/l00/qx
trace /t_bench/l1/l01/qx
trace /t_bench/l1/l02/qx
trace /t_bench/l1/l03/qx
trace /t_bench/l1/l04/qx
trace /t_bench/l1/l05/qx
trace /t_bench/l1/l06/qx
trace /t_bench/l1/l07/qx

trace /t_bench/l1/l00/qw
trace /t_bench/l1/l01/qw
trace /t_bench/l1/l02/qw
trace /t_bench/l1/l03/qw
trace /t_bench/l1/l04/qw
trace /t_bench/l1/l05/qw
trace /t_bench/l1/l06/qw
trace /t_bench/l1/l07/qw

trace /t_bench/l1/l00/qy
trace /t_bench/l1/l01/qy
trace /t_bench/l1/l02/qy
trace /t_bench/l1/l03/qy
trace /t_bench/l1/l04/qy
trace /t_bench/l1/l05/qy
trace /t_bench/l1/l06/qy
trace /t_bench/l1/l07/qy

trace /t_bench/l1/l03/qt
trace /t_bench/l1/l07/qt
trace /t_bench/l1/l11/qt
trace /t_bench/l1/l15/qt
trace /t_bench/l1/l19/qt
trace /t_bench/l1/l23/qt
trace /t_bench/l1/l27/qt
trace /t_bench/l1/l31/qt
end pe_test

comm trace_r1023
trace /t_bench/l1/c_mem(1023,0)
trace /t_bench/l1/c_mem(1023,1)
trace /t_bench/l1/c_mem(1023,2)
trace /t_bench/l1/c_mem(1023,3)
trace /t_bench/l1/c_mem(1023,4)
trace /t_bench/l1/c_mem(1023,5)
trace /t_bench/l1/c_mem(1023,6)
trace /t_bench/l1/c_mem(1023,7)
end trace_r1023



-------------------------------------------------------------------------
-- me.cmd
-------------------------------------------------------------------------
comm init
cd t_bench
run 20
ME
end init

comm ME
  ram
  wr
  echo 'writing 1 and 0 masks'
  run    640
  echo 'writing other masks'
  run   7680
  echo 'writing distortions'
  run  81920

  rd_op
  wr_nop

  ram
  rd
  run 320

  foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
    echo 'compare column-wide for the minimum'
    add_sub16
    movl

    ram
    rd
    echo 'reading results of row-wide min search'
    run 320

    rd_op
    echo 'copy the minimum to the base row'
    mov16
    rd_op
  end

  foreach j in 1 2 3 4
    max_min16

    ram
    rd
    echo 'reading results of col-wide min search'
    run 320
  end
end  ME


comm ME_PD_proposed
ram
wr
echo 'writing 0 and 1 masks'
run   640
echo 'writing masks, boundaries, etc. '
run   7680
echo 'writing the first frame data'
run  81920
echo 'writing the second frame data'
run  81920
```

```
echo 'H=-1'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr1_8
  end
end
echo 'distortion computation'
DIS_COMP3pd

echo 'H=-2'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr2_8
  end
end
DIS_COMP3pd

echo 'H=-3'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr3_8
  end
end
DIS_COMP3pd

echo 'H=-4'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr4_8
  end
end
DIS_COMP3pd

echo 'H=-5'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr5_8
  end
end
DIS_COMP3pd

echo 'H=-6'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr6_8
  end
end
DIS_COMP3pd

echo 'H=-7'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr7_8
  end
end
DIS_COMP3pd

echo 'H=-8'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr8_8
  end
end
DIS_COMP3pd

echo 'H=0'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub8
  end
end
DIS_COMP3pd

echo 'H=+1'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr1_8
  end
end
DIS_COMP3pd

echo 'H=+2'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr2_8
  end
end
DIS_COMP3pd

echo 'H=+3'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr3_8
  end
end
DIS_COMP3pd

echo 'H=+4'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr4_8
  end
end
DIS_COMP3pd

echo 'H=+5'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr5_8
  end
end
DIS_COMP3pd

echo 'H=+6'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr6_8
  end
end
DIS_COMP3pd

echo 'H=+7'
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr7_8
  end
end
DIS_COMP3pd

MV_DET3pd

ram
rd
echo 'reading the distortion vector'
run   5440
echo 'reading the motion vectors'
run   5440
end  ME_PD_proposed

comm ME_FBMA_proposed
ram
wr
echo 'writing 0 and 1 masks'
run   640
echo 'writing masks, boundaries, etc. '
run   7680
echo 'writing the first frame data'
run  81920
echo 'writing the second frame data'
run  81920

echo 'H=-1'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr1_8
  end
end
echo 'distortion computation'
DIS_COMP3

echo 'H=-2'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr2_8
  end
end
DIS_COMP3

echo 'H=-3'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr3_8
  end
end
DIS_COMP3

echo 'H=-4'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr4_8
  end
end
DIS_COMP3
```

```
echo 'H=-5'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr5_8
  end
end
DIS_COMP3

echo 'H=-6'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr6_8
  end
end
DIS_COMP3

echo 'H=-7'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr7_8
  end
end
DIS_COMP3

echo 'H=-8'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr8_8
  end
end
DIS_COMP3

echo 'H=0'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub8
  end
end
DIS_COMP3

echo 'H=+1'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr1_8
  end
end
DIS_COMP3

echo 'H=+2'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr2_8
  end
end
DIS_COMP3

echo 'H=+3'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr3_8
  end
end
DIS_COMP3

echo 'H=+4'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr4_8
  end
end
DIS_COMP3

echo 'H=+5'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr5_8
  end
end
DIS_COMP3

echo 'H=+6'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr6_8
  end
end
DIS_COMP3

echo 'H=+7'
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    add_sub_lr7_8
  end
end
DIS_COMP3

MV_DET3

ram
rd
echo 'reading the distortion vector'
run   5440
echo 'reading the motion vectors'
run   5440
end  ME_FBMA_proposed

comm ME4
ram
wr
run    7680
run  163840
run  163840

erodil_v3
cond_erodil_v3
erodil_v3
erodil_v3

foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8
    add_sub8
    add_sub9
    rd_op
    wr_op
  end
end

ram
rd
run  20480
run  3200
end  ME4


comm ME_COMP
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr1
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr2
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr3
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr4
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr5
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr6
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr7
  end
end
DIS_COMP
```

```
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr8
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr1
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr2
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr3
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr4
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr5
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr6
  end
end
DIS_COMP

foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    xor_lr7
  end
end
DIS_COMP

end  ME_COMP

comm DIS_COMP3pd
cram
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4
    pas8a
  end
  foreach j in 1 2
    pas9a
  end
  foreach j in 1
    pas10a
  end

  abs11

  add_sub_12_11u
  add_sub_14_12u
  rd_op
  add_sub_18_13u
  rd_op
end
end  DIS_COMP3pd

comm DIS_COMP3
cram
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8
    pas8a
  end
  foreach j in 1 2 3 4
    pas9a
  end
  foreach j in 1 2
    pas10a
  end
  foreach j in 1
    pas11a
  end
  abs12

  add_sub_11_12u
  add_sub_12_13u
  add_sub_14_14u
  rd_op
  add_sub_18_15u
  rd_op
end
end  DIS_COMP3

comm DIS_COMP2
cram
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8
    add_sub8u
  end
  foreach j in 1 2 3 4
    add_sub9u
  end
  foreach j in 1 2
    add_sub10u
  end
  foreach j in 1
    add_sub11u
  end

  add_sub_11_12u
  add_sub_12_13u
  add_sub_14_14u
  rd_op
  add_sub_18_15u
  rd_op
end
end  DIS_COMP2

comm DIS_COMP
cram
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  foreach j in 1 2 3 4 5 6 7 8
    add_sub1u
  end
  foreach j in 1 2 3 4
    add_sub2u
  end
  foreach j in 1 2
    add_sub3u
  end
  foreach j in 1
    add_sub4u
  end

  add_sub_11_5u
  add_sub_12_6u
  add_sub_14_7u
  rd_op
  add_sub_18_8u
  rd_op
end
end  DIS_COMP

comm ME_XOR
ram
wr
run 22400
run  5120
run   640
run 163840

erodil_v3
cond_erodil_v3
erodil_v3
erodil_v3

foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8
    add_sub8
    add_sub9
    rd_op
    wr_op
  end
```

```
end

ram
rd
foreach i in 1 2 3
  run 163840
end
run 20480
end ME_XOR

comm cond_erodil_v3
mov8
mov8
add_sub8
rd_op
mov8
rd_op

for i in 1 2 3 4 5 6
  for j in 1 2 3 4 5 6 7 8 9 10
    for ka in 1
      mov8
      add_sub8
      rd_op
      mov8
      rd_op
      add_sub8
      rd_op
      mov8
      rd_op
    end
    for kb in 1 2
      add_sub8
      rd_op
      rd_op
      mov8_lr
      rd_op
    end
    for condition in 1
      add_sub8
      rd_op
      mov8
      rd_op
    end
  end
end

for i in 1 2
  for j in 1
    for ka in 1
      mov8
      add_sub8
      rd_op
      mov8
      rd_op
      add_sub8
      rd_op
      mov8
      rd_op
    end
    for kb in 1 2
      add_sub8
      rd_op
      rd_op
      mov8_lr
      rd_op
    end
    for condition in 1
      add_sub8
      rd_op
      mov8
      rd_op
    end
  end
end
end cond_erodil_v3

comm erodil_v3
mov8
mov8
add_sub8
rd_op
mov8
rd_op

for i in 1 2 3 4 5 6
  for j in 1 2 3 4 5 6 7 8 9 10
    for ka in 1
      mov8
      add_sub8
      rd_op
      mov8
      rd_op
      add_sub8
      rd_op
      mov8
      rd_op
    end
    for kb in 1 2
      add_sub8
      rd_op
      rd_op
      mov8_lr
      rd_op
    end
  end
end

for i in 1 2
  for j in 1
    for ka in 1
      mov8
      add_sub8
      rd_op
      mov8
      rd_op
      add_sub8
      rd_op
      mov8
      rd_op
    end
    for kb in 1 2
      add_sub8
      rd_op
      rd_op
      mov8_lr
      rd_op
    end
  end
end
end erodil_v3


comm erosion_v1
foreach i in 1 2 3 4 5 6 7 8 9
 foreach j in 1 2 3 4 5 6 7
  mov8
  add_sub8
  rd_op
  mov8
  rd_op
 end
end

foreach i in 1 2 3 4 5 6
 foreach j in 1 2 3 4 5 6 7 8 9 10
  add_sub8
  rd_op
  mov8
  rd_op
 end
end
 foreach j in 1 2
  add_sub8
  rd_op
  mov8
  rd_op
 end

foreach i in 1 2 3 4
 foreach j in 1 2 3
  foreach k in 1 2 3 4 5 6 7 8 9 10
   add_sub8_lr
   rd_op
   mov8_lr
   rd_op
  end
 end
 foreach l in 1
   add_sub8_lr
   rd_op
   mov8_lr
   rd_op
 end
end

mov8
mov8
end erosion

-- typical trx function ---
comm trx8
cram
rd_op
```

```
rd_op
rd_op
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_nop
end
rd_op
rd_op
end trx8


-- typical adiv functions --
comm adiv27_16_0
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  rd_op
  rd_op
end
foreach i in 1 2 3 4 5 6 7 8 9 10 11
  rd_op
  wr_op
end
end adiv27_16_0

comm adiv27_16_1
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  rd_op
  rd_op
end
foreach i in 1 2 3 4 5 6 7 8 9 10 11
  rd_op
  wr_op
end
foreach i in 1
  wr_nop
end
end adiv27_16_1

comm mov_ashl_ashr
mov8
ashl8_4_4
ashr8_4_4
end mov_ashl_ashr

comm clrm9
cram
rd_op
rd_op
foreach i in 1 2 3 4 5 6 7 8 9
  wr_nop
end
rd_op
end clrm9


comm clr2
cram
rd_op
foreach i in 1
  foreach j in 1
    wr_nop
  end
end
end  clr2

-- typical clr function --
comm clr33
cram
rd_op
foreach i in 1 2 3 4
  foreach j in 1 2 3 4 5 6 7 8
    wr_nop
  end
end
end clr33

comm hor_transfer8
cram
rd_op
rd_op
rd_op
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
rd_op
rd_op
```

```
end hor_transfer8

comm par_add_sub
mov8m
mov8m
add_sub8
end par_add_sub

comm seg_bus
cram
rd_op
max_min8
end seg_bus


-- typical pas function --
comm pas8
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  rd_op
  wr_op
end
rd_op
wr_op
rd_op
end pas8

comm pas8a
cram
rd_op
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  rd_op
  wr_op
end
rd_op
wr_op
rd_op
end  pas8a

-- typical addition/ subtraction functions --
comm add_sub8
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  rd_op
  wr_op
end
rd_op
wr_op
end add_sub8

comm add_sub8_lr
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  rd_op
  wr_op
end
rd_op
rd_op
wr_op
end add_sub8_lr

-- typical acc functions --
comm acc8
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
rd_op
wr_op
end acc8


comm acc12m
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12
  rd_op
  rd_op
  wr_op
end
end acc12m
```

```
comm acc12mn
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  acc12m
end
end acc12mn

comm acc13u
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13
  rd_op
  rd_op
  wr_op
end
end  acc13u


-- typical multiplication functions --
comm mul9_8
cram
  clr33
  rd_op
  add_sub9
  foreach i in 1 2 3 4 5 6 7
    wr_op
  end
  rd_op
  add_sub9
  foreach i in 1 2 3 4 5 6
    wr_op
  end
  rd_op
  add_sub9
  foreach i in 1 2 3 4 5
    wr_op
  end
  rd_op
  add_sub9
  foreach i in 1 2 3 4
    wr_op
  end
  rd_op
  add_sub9
  foreach i in 1 2 3
    wr_op
  end
  rd_op
  add_sub9
  foreach i in 1 2
    wr_op
  end
  rd_op
  add_sub9
  foreach i in 1
    wr_op
  end
  rd_op
  add_sub9
rd_op
rd_op
end mul9_8


-- typical max/min search function --
comm max_min8
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
  rd_op
end
wr_nop
end max_min8

comm pmax_min9
cram
rd_op
rd_op
foreach i in 1 2 3 4 5 6 7 8 9
  rd_op
  wr_op
  rd_op
end
wr_nop
end  pmax_min9


-- typical one and two complement functions --
comm one8
cram
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
end  one8

comm two8
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
rd_op
end  two8


-- typical abs function --
comm abs8
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
rd_op
end  abs8


-- typical mov functions --
comm mov1
cram
foreach i in 1
  rd_op
  wr_nop
end
end  mov1

comm mov_new1
cram
foreach i in 1
  rd_op
  wr_nop
end
end  mov_new1


-- typical ashl functions --
comm ash19_8_0
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  wr_nop
end
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_nop
end
  rd_op
  wr_op
end ash19_8_0

comm ash19_8_1
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  wr_nop
end
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_nop
end
  rd_op
  wr_op
foreach i in 1
  wr_nop
end
end ash19_8_1


-- typical ashr functions --
comm ashr27_0_1
cram
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26
  rd_op
  wr_nop
end
  rd_op
  wr_op
foreach i in 1
  wr_nop
end
```

```
end ashr27_0_1

comm ashr28_16_0
cram
foreach i in 1 2 3 4 5 6 7 8 9 10 11
  rd_op
  wr_nop
end
  rd_op
  wr_op
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  wr_nop
end
end ashr28_16_0


comm mov8m
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  rd_op
  wr_op
end
rd_op
end mov8m

comm lsh_rsh
cram
rd_op
rd_op
wr_op
end lsh_rsh


-- typical unsigned add/sub functions --
comm add_sub1u
cram
rd_op
foreach i in 1
  rd_op
  rd_op
  wr_op
end
wr_nop
end  add_sub1u

-- typical left/right add/sub functions --
comm add_sub_lr1_8
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1
    rd_op
  end
  rd_op
  wr_op
end
rd_op
wr_nop
end  add_sub_lr1_8

comm add_sub_lr2_8
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2
    rd_op
  end
  rd_op
  wr_op
end
rd_op
wr_nop
end  add_sub_lr2_8

comm add_sub_ll_5u
cram
rd_op
foreach i in 1 2 3 4 5
  foreach j in 1
    rd_op
  end
  rd_op
  wr_op
end
wr_nop
end  add_sub_ll_5u

comm add_sub_ll_12u
cram
rd_op
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12
  foreach j in 1
    rd_op
  end
  rd_op
  wr_op
end
wr_nop
end  add_sub_ll_12u



-- typical xor precedure --
comm xor
cram
rd_op
rd_op
wr_nop
end  xor

comm xor_lr1
cram
rd_op
rd_op
wr_nop
end  xor_lr1

comm xor_lr2
cram
rd_op
foreach i in 1
  rd_op
end
rd_op
wr_nop
end  xor_lr2


-- typical read/write (no) operation cycles --
comm rd_op
cram
rd
assign '0' ops
run 15
assign '1' ops
run 20
assign '0' ops
run 5
end rd_op

comm rd_nop
cram
rd
assign '0' ops
run 40
end rd_nop

comm wr_op
cram
wr
assign '0' ops
run 15
assign '1' ops
run 20
assign '0' ops
run 5
end wr_op

comm wr_nop
cram
wr
assign '0' ops
run 40
end wr_nop

comm wr
assign '0' rd_wrb
end wr

comm rd
assign '1' rd_wrb
end rd

comm ram
assign '1' r_cb
end ram

comm cram
assign '0' r_cb
end cram
```

```
-- typical wave form tracing procedure --
comm tracing
cd t_bench
trace mck cck
trace /t_bench/l1/ops
trace rd_wrb r_cb
trace data_bus(7 downto 0)
trace addr_bus(14 downto 3) addr_bus(2 downto 0)
trace /t_bench/l1/l01/alu_op
trace /t_bench/l1/l01/qx
trace /t_bench/l1/we(1)
trace /t_bench/l1/l01/qy
trace /t_bench/l1/l01/qm
trace /t_bench/l1/l01/qs
trace /t_bench/l1/l01/myx(2 downto 0)
trace /t_bench/l1/l01/q_tmp
run 20
SEQUENCE_USED
end  tracing
-------------------------------------------------------------------------



-------------------------------------------------------------------------
-- dct.cmd
-------------------------------------------------------------------------

comm init
cd t_bench
run 20
LEE
end init


comm CHEN_H
ram
wr
run 22400
run  5120
run 163840

foreach outer_loop in 1 2 3 4 5 6 7 8
foreach inner_loop in 1 2 3 4 5 6 7 8
foreach i in 1 2 3 4 5 6 7 8
  trx8
end
pas8

foreach i in 1 2
  trx9
end
clrm9
pas9
mul10_9

foreach i in 1 2 3 4 5 6 7 8
  trx19
end
pas19

foreach i in 1 2 3 4 5 6 7 8
  trx20
end
mul20_9
adiv28_8_0
pas20
mul21_9
adiv29_16_0

foreach i in 1 2 3 4 5 6 7 8
  trx10
end

end
end

ram
rd
run 245760
end CHEN_H


comm chen_h_unoptimized
ram
wr
run 16000
run  5120
run 40960

foreach i in 1 2 3 4 5 6 7 8
  trx8
end
```

```
pas8

foreach i in 1 2 3 4 5 6
  trx9
end
clrm9
pas9
mul10_9

foreach i in 1 2 3 4 5 6 7 8
  trx19
end
mul19_9
adiv27_8_0

pas19
mul20_9
adiv29_16_0
foreach i in 1 2 3 4
  trx13
end

foreach i in 1 2 3 4
  trx13
end
mul13_9

adiv21_8_0
pas13
mul14_9
adiv23_8_0
foreach i in 1 2 3 4
  trx13
end

ram
rd
run 40960
run 40960
run  6400
end chen_h_unoptimized


comm trace_data
trace mck cck
trace /t_bench/l1/ops
trace rd_wrb r_cb
trace data_bus(7 downto 0)
trace addr_bus(13 downto 3) addr_bus(2 downto 0)
end trace_data

comm CHO
ram
wr
run 163840
foreach i in 1 2 3 4 5 6 7 8
  foreach j in 1 2 3 4 5 6 7 8
    add_sub8
  end
end
foreach i in 1 2 3 4 5 6 7 8
  run_dct1a1
  run_dct2a1
  run_dct3a1
  run_dct4a1
end

foreach i in 1 2 3 4
  foreach j in 1 2 3 4 5 6 7 8
    add_sub13
  end
end

foreach i in 1 2
  foreach j in 1 2 3 4 5 6 7 8
    add_sub14
  end
end

foreach i in 1 2
  ashr14_0_1
  foreach i in 1 2 3 4 5 6 7
    add_sub14
  end
end

foreach i in 1 2 3 4 5 6 7 8
  mov15
end
```

```
ashr15_0_1
foreach i in 1 2 3
  add_sub15
end
ashr15_0_1
foreach i in 1 2 3
  add_sub15
end
foreach i in 1 2 3 4 5 6 7 8
  add_sub15
end
foreach i in 1 2
  add_sub15
end
ashr15_0_1
foreach i in 1 2 3
  add_sub15
end
two15
ashr15_0_1
add_sub15
two16

foreach i in 1 2 3 4 5 6 7 8
  ashr15_0_1
end

foreach i in 1 2 3 4
  adiv16_1_1
end
mov16
foreach i in 1 2 3 4 5 6 7 8 9 10 11 12 13
  adiv16_1_1
end
mov16
foreach i in 1 2 3
  adiv16_1_1
end
mov16
adiv16_1_1

ram
rd
run 245760
end CHO

comm CHEN
ram
wr
run 163840

foreach i in 1 2 3 4 5 6 7 8
  run_dct1a
  run_dct2a
  run_dct3a
  run_dct4a
end
foreach j in 1 2 3 4 5 6 7 8
  run_dct1b
  run_dct2b
  run_dct3b
  run_dct4b
end

ram
rd
run 245760
end CHEN


comm run_dct4old
foreach i in 1 2 3 4
  ashr19_8_0
end
foreach i in 1 2 3 4
  mul19_8
  mul19_8
  add_sub27
  ashr28_16_0
end
end run_dct4old

comm run_dct1a
foreach i in 1 2 3 4 5 6 7 8
  add_sub8
end
end run_dct1a

comm run_dct2a
add_sub9
add_sub9
add_sub9
```

```
add_sub9
ashl9_8_1
add_sub9
mul10_8
add_sub9
mul10_8
ashl9_8_1
end run_dct2a

comm run_dct3a
add_sub10
mul11_8
add_sub10
mul11_8
mul10_8
mul10_8
add_sub18
mul10_8
mul10_8
add_sub18
add_sub18
add_sub18
add_sub18
add_sub18
end run_dct3a

comm run_dct4a
foreach i in 1 2 3 4
  adiv18_8_0
end
foreach i in 1 2 3 4
  mul19_8
  mul19_8
  add_sub27
  adiv26_16_0
end
end run_dct4a

comm run_dct1a1
add_sub9
add_sub9
add_sub9
add_sub9
add_sub9
add_sub9
add_sub9
add_sub9
end run_dct1a1

comm run_dct2a1
add_sub10
add_sub10
add_sub10
add_sub10
ashl10_8_1
add_sub10
mul11_8
add_sub10
mul11_8
ashl10_8_1
end run_dct2a1

comm run_dct3a1
add_sub11
mul12_8
add_sub11
mul12_8
mul11_8
mul11_8
add_sub19
mul11_8
mul11_8
add_sub19
add_sub19
add_sub19
add_sub19
add_sub19
end run_dct3a1

comm run_dct4a1
foreach i in 1 2 3 4
  adiv20_8_1
end
foreach i in 1 2 3 4
  mul20_8
  mul20_8
  add_sub28
  adiv29_16_0
end
end run_dct4a1

comm run_dct1b
```

```
foreach i in 1 2 3 4 5 6 7 8
  add_sub10
end
end run_dct1b

comm run_dct2b
add_sub11
add_sub11
add_sub11
add_sub11
ashl11_8_1
add_sub11
mul12_8
add_sub11
mul12_8
ashl11_8_1
end run_dct2b

comm run_dct3b
add_sub12
mul13_8
add_sub12
mul13_8
mul12_8
mul12_8
add_sub20
mul12_8
mul12_8
add_sub20
add_sub20
add_sub20
add_sub20
add_sub20
end run_dct3b

comm run_dct4b
foreach i in 1 2 3 4
  adiv20_8_0
end
foreach i in 1 2 3 4
  mul21_8
  mul21_8
  add_sub29
  adiv28_16_0
end
end run_dct4b

comm LEE
echo 'Loading image data ...'
ram
wr
run 163840

echo 'Start DCT ...'
foreach i in 1
  echo 'dct stage1 ...'
  run_dct_lee1a
  echo 'dct stage2 ...'
  run_dct_lee2a
  echo 'dct stage3 ...'
  run_dct_lee3a
  echo 'dct stage4 ...'
  run_dct_lee4a
end

echo 'Printing results ...'
ram
rd
run 245760
end LEE

comm run_dct_lee1a
foreach i in 1 2 3 4 5 6 7 8
  add_sub8
end
mul9_8
ashl17_0_2
mul9_8
ashl17_0_2
mul9_8
ashl17_2_0
mul9_8
ashl17_0_2
```

```
end run_dct_lee1a

comm run_dct_lee2a
foreach i in 1 2 3 4
  add_sub9
end
foreach i in 1 2 3 4
  add_sub19
end

  mul10_8
  ashl18_0_1
  mul10_8
  ashl18_1_0

  mul20_8
  ashl28_0_1
  mul20_8
  ashl28_1_0
end run_dct_lee2a

comm run_dct_lee3a
foreach i in 1 2
  add_sub10
end
foreach i in 1 2
  add_sub19
end
foreach i in 1 2
  add_sub20
end
foreach i in 1 2
  add_sub29
end

  mul11_8
  adiv19_8_0
  mov10

  mul11_8
  adiv19_8_0
  mov10

  mul20_8
  adiv28_16_0
  mov10

  ashl20_8_0
  add_sub28
  adiv29_16_0
  mov10
end run_dct_lee3a

comm run_dct_lee4a
foreach i in 1 2
  adiv30_8_0
end
  mul21_8
  adiv29_8_0
  mul22_8
  adiv30_8_0

  ashl21_0_2
  ashl21_0_2
  add_sub22
  ashl22_0_1

  add_sub23
  adiv24_8_0
  mov10

  add_sub23
  adiv24_8_0
  mov10

  add_sub23
  adiv24_8_0
  mov10

  adiv23_8_0
  mov10
end run_dct_lee4a
```