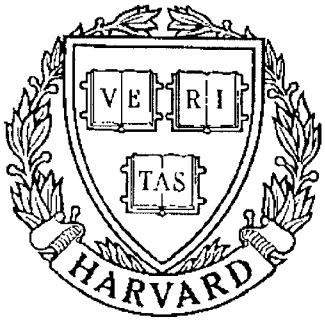
THESIS REPORT
Ph.D.

SYSTEMS
RESEARCH
CENTER



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

VLSI Architectures for Real-Time Signal Processing

by C. Chakrabarti
Advisor: J.F. JáJá

Ph.D. 90-6

VLSI ARCHITECTURES FOR REAL-TIME SIGNAL PROCESSING

by

Chaitali Chakrabarti

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1990

Advisory Committee:

Professor Joseph F. JáJá, Chairman
Professor John Baras
Professor Hung C. Lin
Professor Shihab Shamma
Professor Azriel Rosenfeld

ABSTRACT

Title of Dissertation: VLSI Architectures for Real-Time Signal Processing

Chaitali Chakrabarti, Doctor of Philosophy, 1990

Dissertation directed by: Joseph F. JáJá, Professor, Electrical Engineering

We address the problems of developing efficient special-purpose VLSI architectures for computing some important real-time signal processing tasks, namely, one-dimensional Discrete Hartley (DHT) and Discrete Cosine transforms (DCT), multidimensional transforms, template matching and block matching. An important requirement of all these architectures is that they should process huge amounts of data at very high throughput rates.

The first problem that we address involves developing systolic array architectures for computing one-dimensional DHT and DCT over N points, when N is factorizable into mutually prime factors N_1 and N_2. We map the one-dimensional transform into a two-dimensional transform over $(N_1 \times N_2)$ points such that the algorithm consists of computing one-dimensional transform over columns and rows of the two-dimensional data array. The hardware requirement is considerably reduced because of this mapping. The architecture consists of simple and regular units which are completely pipelined.

Next we look at the more general problem of computing any $(N \times N \times \ldots \times N)$ d-dimensional linear separable transform (DXT). Here we develop a family of optimal architectures with area-time trade-offs. The architecture consists of one-dimensional DXT(N) transform computation units which compute DXT(N)

over one index, and permutation units which order data so that in the next iteration DXT(N) can be computed over the next index. The architecture has an area $A = O(N^{d+2a})$ and computation time $T = O(dN^{\frac{d}{2}-a}b)$ for all a in the range $\frac{1}{2}\log_N b \leq a \leq \frac{d}{2}$, where $b = O(\log M)$ is the precision.

The third problem that we address is developing efficient architectures for computing very high input/output (I/O) bandwidth operations, like template matching and block matching. Here we develop a linear semi-systolic array architecture which balances computations in the processor array with the I/O requirements. The I/O bandwidth is reduced by storing part of the input image on-chip in shift registers in each processor, and by circulating the shift registers. The architecture achieves optimal speed-up.

Contents

List of Figures

To Ma-Baba

Acknowledgements

First and foremost, I would like to express my gratitude to my advisor and mentor, Joseph JáJá. I am indebted to Joseph for his guidance, sustained encouragement and whole-hearted support throughout the course of my graduate study. By advice and example, he taught me the essentials of being a good researcher (and I hope I learnt it!).

I thank the other members of my thesis committee, Prof. John Baras, Prof. Hung C. Lin, Prof. Azriel Rosenfeld and Prof. Shihab Shamma for their help and insightful comments. Nariman Farvardin, Kazuo Nakajima and Dave Mount provided periodic comments and suggested helpful research directions.

I would like to thank the Systems Research Center and the Electrical Engineering Department for the excellent resources and the stimulating research environment.

Maryland is not all research! My stay here would not have been this much fun without the company of my friends. Let me begin by thanking my roommates or rather my "family" at Maryland – Sukanya, Dulcy and Sharmila. I could not have asked for a more compassionate, caring and loving trio ! Thank you PJ for being such a wonderful and supportive friend; Rajasekar for the good times at 7404 Rhode Island Avenue; Sharat, Rim, Vijay, Alice and Tom for your encouragement and support; Sridhar, Jagannathan, Ravi, Shu-Sun, Yingmin, Ryu for making the VLSI Lab a fun place to work in; Sukla, Prosenjitda, Dheeraj, Malini, Raghu, Madhura, Soma, Vivek and Venu for the wonderful memories of the times spent together. Thank you all !

Members of my immediate family – my sister, my parents and my grandparents – have been a constant source of emotional support and encouragement through all phases of my studenthood, and I take this opportunity to express my gratitude for them.

Finally I would like to thank my closest friend and confidant, Rao, who entered my life during this period and stood by me through thick and thin. I owe this thesis to him.

Chapter 1

Introduction

Many computational tasks involved in image processing, video processing and speech processing schemes require the ability to carry out the operations in real-time. Real-time signal processing tasks involve processing huge amounts of data at very high throughput rates. To give an estimate of the throughput, consider the computation of template matching when the image is of size (512×512), the template is of size (8×8) and the frame rate is 30 Hz. The number of multiplication-accumulation operations is then 60 MOPs (million operations per second). It is not unusual for a real-time application to have a throughput as high as 1000 MOPs. Clearly such high computational rates cannot be achieved by sequential systems. In fact, even general-purpose parallel computers cannot match the speed and volume requirements because of severe system overheads. The only way to meet the high computational rates of real-time signal processing tasks is by developing special-purpose architectures which exploit the regularity, recursiveness and locality of the signal processing algorithms. Architectures

which consist of locally interconnected networks (and thereby circumvent the communication problem) and which incorporate both parallel processing and pipelining are well suited for such algorithms. Systolic arrays [32], wavefront arrays [33] are examples of such architectures.

The organization of the rest of this chapter is as follows. We discuss the desired characteristics of such architectures in Section 1.1 and then briefly review some architectures which can handle real-time operation in Section 1.2. In Section 1.3 we make some remarks about its evaluation. In Section 1.4 we give an overview of the architectures that we developed for handling the following signal processing tasks: one-dimensional Discrete Hartley transform (DHT) and Discrete Cosine transform (DCT), multi-dimensional linear separable transforms, template matching, and block matching. We conclude with the thesis organization in Section 1.5.

1.1 Architectural requirements

In this section we briefly describe the requirements of special-purpose architectures for real-time signal processing [32, 33]:

Simplicity and regularity of design: This is an important factor in VLSI design. If the design is simple, and if it consists of a few different types of units, then the design cost would be low enough to justify limited applicability. Moreover such a design is more likely to be modular.

Parallel and pipelined processing: The degree of concurrency in a special-purpose architecture is determined by the underlying algorithm, and whether the algorithm can be mapped into an architecture with high degrees of parallel and pipelined processing. Pipelining increases the throughput rate which is a very important system performance factor. Pipelining at all levels (bit-level, word-level and array-level) need to be exploited.

Communication: This is a very critical factor in VLSI design. In fact, inter-processor communications dominate the cost of parallel algorithms and systems. Local and regular communication schemes for both data transactions and control flow are essential to enable efficient implementation.

Balancing computation with input/output: Real-time signal processing operations are computation-bound, that is, the total number of computations is larger than the number of input/output (I/O) operations. Thus for applications with large throughput, multiple computations have to be performed per I/O access. For some applications repetitive use of data items may require it to be stored on-chip. Thus the computation rate has to be balanced with both the external and the internal memory bandwidth.

1.2 Architectures

There are two approaches to the design of special-purpose architectures for real-time signal processing. One is to design dedicated hard-wired systems and the other is to design programmable array processor systems. Systolic arrays

and their variations are used in many dedicated systems. In this section we first briefly review systolic arrays, and then describe some other parallel array processor systems.

1.2.1 Systolic arrays

A systolic array is defined by Kung and Leiserson [34] as a "network of processors which rhythmically compute and pass data through the system". The function of a processor is analogous to that of a heart. Each processor regularly pumps data in and out, each time performing some simple computation, so that a regular flow of data in maintained in the network. A systolic array has the following important properties [32, 33].

Modularity and regularity: A systolic array consists of simple and modular processors interconnected in a regular pattern. Examples of such patterns are linear (one-dimensional), orthogonal (two-dimensional), hexagonal (see Figure 1.1). The array can be extended indefinitely.

Spatial and temporal locality: The processors communicate with each other via a local interconnection structure (spatial locality). Signal transactions from one processor to the next can be completed in one unit time (temporal locality).

Pipelinability: A systolic array has a very high degree of pipelining. Once a data item is fetched from the memory, it is used to compute/update intermediate results in each processor while being 'pumped' from processor to processor along the array. Such a data flow scheme is ideal for computation-bound operations,

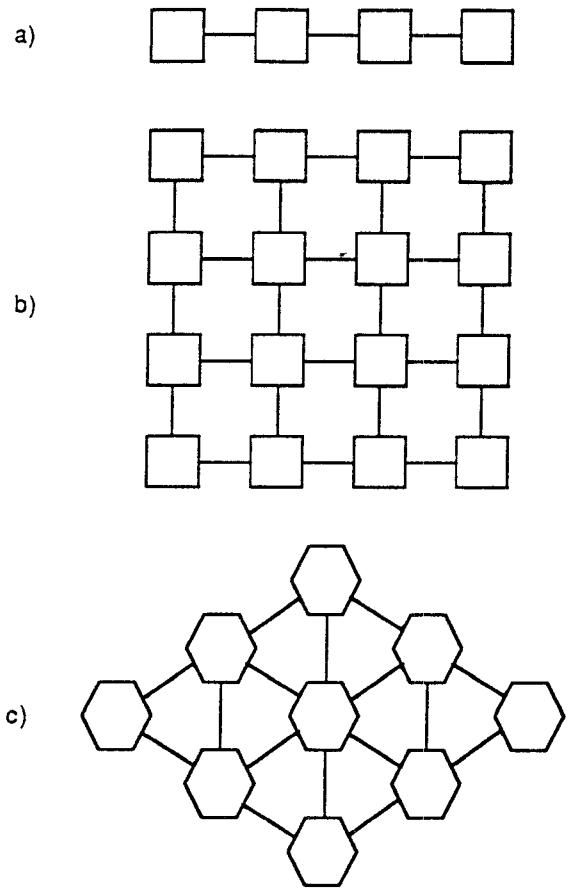


Figure 1.1: Systolic array configurations: a) Linear array b) Orthogonal array c) Hexagonal array

where multiple operations are performed on each data item.

Synchrony: In a systolic array the data passes through the array in a regular, rhythmic pattern. All data movements are synchronized by a global clock.

We illustrate the operation of a systolic array with the help of a matrix-vector multiplication example. Consider the problem of multiplying a matrix A with a vector x. Here A has a bandwidth $w = p + q - 1$ (see Figure 1.2). The architecture consists of a linear array of w inner product processors. The

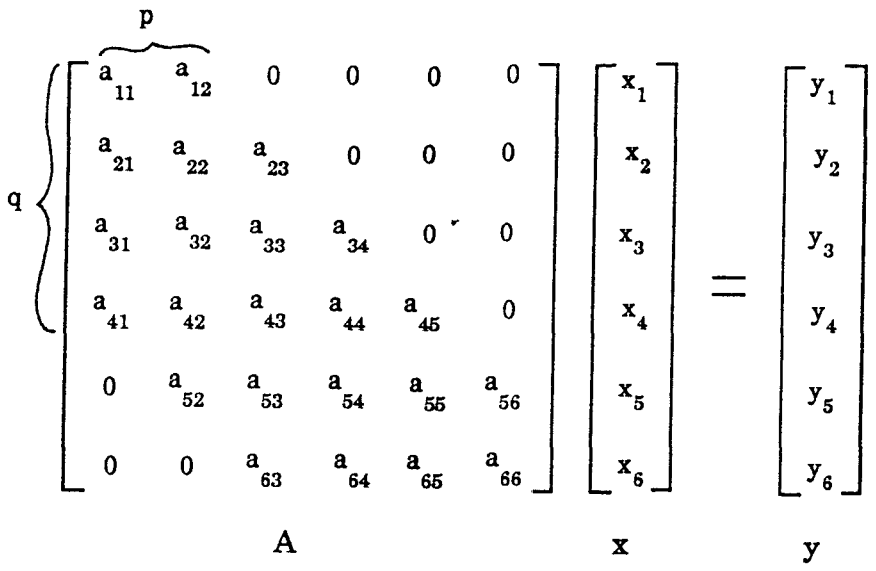


Figure 1.2: Multiplication of banded matrix A ($p = 2$, $q = 4$) with vector x of size 6

data flow through the array is as follows. The y_is move to the left, the x_is move to the right and the a_{ij}s are fed from the top. Figure 1.3 illustrates the data flow in an example where $p = 4$, $q = 2$, and $|x| = 6$. We next trace the computation of y_2 through this array. y_2 is initially set to 0 and input from the right hand side of the array in Step 2. It is updated by $PE[2]$ in Step 4 ($y_2 = a_{21} * x_1$), by $PE[1]$ in Step 5 ($y_2 = a_{21} * x_1 + a_{22} * x_2$), and by $PE[0]$ in Step 6 ($y_2 = a_{21} * x_1 + a_{22} * x_2 + a_{23} * x_3$). In this way the correct value of y_2 is output in Step 7.

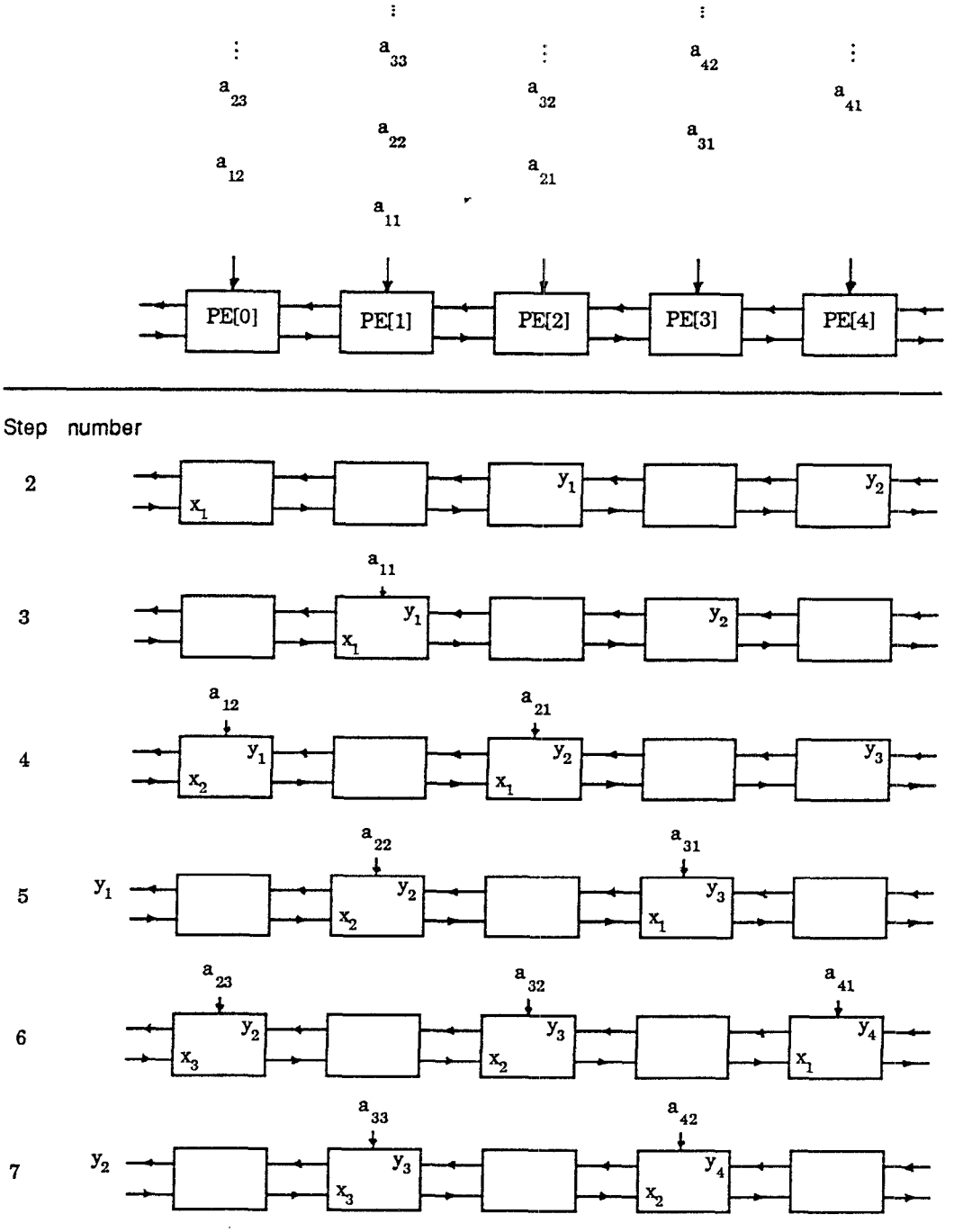


Figure 1.3: Data flow through the systolic array during the computation of Steps 2 through 7 of the matrix-vector algorithm

Systolic architectures have a very high performance for any computation-bound problem that is regular, that is, for applications which involve repetitive computations on a large set of data. Examples include convolution, filters (IIR, FIR, median), transforms, matrix computations, graph algorithms, language recognition, etc.

Semi-systolic architectures: For many signal processing applications such a rigorous framework is too rigid. In some applications it is advantageous to broadcast data or to time-share the processing elements for sequences of different operations. In some other applications it is necessary to store data in a distributed manner (in shift register memories, local pointer addressed memories, etc.) in the array. Sometimes it is necessary to design efficient I/O interfaces which are not necessarily fully regular (an example is an interface consisting of shift registers that have a conditional clock which activates only when new data is available at the inputs). Architectures which are essentially systolic but which incorporate such variations are referred to as semi-systolic architectures.

Wavefront arrays [33] are an extension of systolic arrays where the intercell synchronization is accomplished by local handshakes, instead of a global clock.

1.2.2 Array processor system

An array processor system consists of four major components: host computer, interface unit, processor array and interconnection network [33]. It is used in applications where flexible processing is important.

The functions of the various components in an array processor system are as follows. The host computer provides system monitoring, data storage, program scheduling. The interface unit is responsible for loading/unloading and buffering data to the processor array. The processor array consists of a number of processing elements each equipped with a simple computation unit and (in most cases) a local memory. The mode of operation is either SIMD (Single Instruction Multiple Data stream) [1] or MIMD (Multiple Instruction Multiple Data stream) [2]. The processors communicate with other processors via the interconnection network. Some examples of interconnection network are mesh, binary tree, pyramid, hypercube, and butterfly. Since communication is expensive in VLSI, interconnection networks play an important role in the design of array processor systems. In the rest of this section we present various interconnection networks and state the signal/image processing applications that are suited for each of the networks.

Mesh network:

In a d-dimensional mesh, the P processors are logically arranged in a d-dimensional $(\sqrt[d]{P} \times \sqrt[d]{P} \times \ldots \times \sqrt[d]{P})$ array. The processor at location $(i_{d-1}, i_{d-2}, \ldots, i_0)$ is connected to the processor at location $(i_{d-1}, \ldots, i_j \pm 1, \ldots, i_0)$, $0 \leq j \leq d-1$. In a 2-dimensional mesh, each internal processor is connected to 4 neighboring

[1] In an SIMD architecture, a single controller broadcasts instruction to all the processors. The processors then execute the instructions simultaneously.

[2] In an MIMD architecture each processor reads instructions from its private memory, decodes them, and executes these instructions.

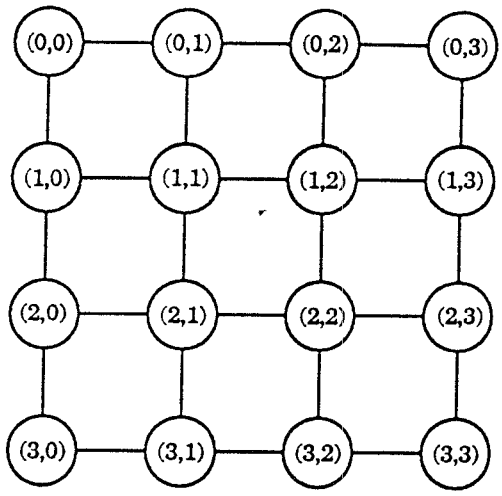


Figure 1.4: 2-dimensional mesh of size 16

processors (see Figure 1.4). The strength of such a network lies in the ease of construction and the high performance of local neighborhood operations. The weakness lies in algorithms where data have to be moved long distances, in which case it may take $O(\sqrt{P})$ time. Algorithms which require $O(\sqrt{P})$ communication operations but perform only $O(\log P)$ calculations are best solved if the mesh network is augmented with a tree of processors. Such a network is known as the mesh of trees network (see Section 3.3.1). In a $(\sqrt{P} \times \sqrt{P})$ mesh of trees network, each row of processors as well as each column of processors form the leaves of a binary tree (see Figure 3.2).

Binary tree network

In a binary tree network, the $P = 2^d - 1$ processors are connected by a complete binary tree of depth $(d-1)$. Each internal processor in the ith level is connected

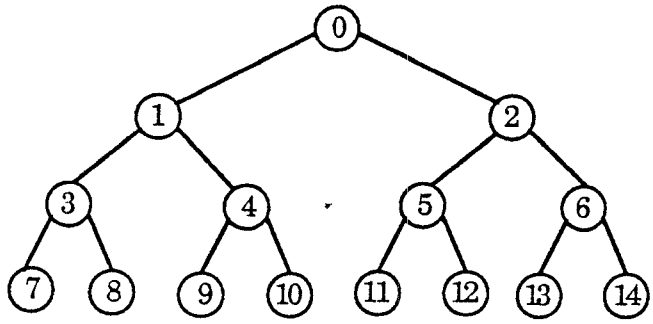


Figure 1.5: Binary tree network of size 15

to its two children in the $(i+1)$th level and to its parent in the $(i-1)$th level. Thus processor j is connected to processor $(2j+1)$, processor $(2j+2)$ and processor $\lfloor j/2 \rfloor$ (see Figure 1.5). (The leaf processors are connected only to their parents and the root processor is connected only to its children.) Such a network is suitable for problems which can be decomposed in a hierarchical way as in dictionary and database problems. Recursive algorithms may also perform better on this network.

Pyramid network

In a pyramid network, the $P = (4^{d+1}-1)/3$ processors are connected by a pyramid of depth d, such that the ith level is a mesh connected network with 4^i processors, $0 \leq i \leq d-1$. Each internal processor in the ith level is connected to its 4 children in the $(i+1)$th level, to its 4 nearest neighbors in the ith level and to its parent in the $(i-1)$th level (see Figure 1.6). All algorithms on the mesh can be run at the lowest level of the pyramid. Algorithms on mesh of trees

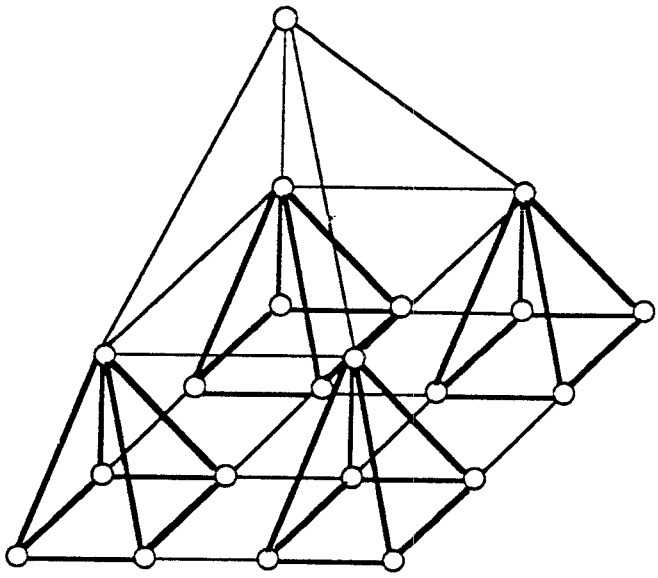


Figure 1.6: Pyramid network of size 21

can be run by using only the vertical connections and the horizontal connections in the base of the pyramid. Such a network is useful in image processing for analyzing images at multiple levels of resolution.

Hypercube network:

In a hypercube network, the $P = 2^d$ processors are connected by a d-dimensional Boolean cube. If the binary representation of i is $i_{d-1}i_{d-2}\ldots i_0$, then processor i is connected to all processors i^j, where i^j is represented by $i_{d-1}\ldots\bar{i_j}\ldots i_0$, $\bar{i_j} = 1 - i_j$ and $0 \leq j \leq d-1$. A hypercube has a recursive structure: a d-dimensional hypercube can be viewed as being two $(d-1)$-dimensional hypercubes with connections between the corresponding corners of the smaller hypercubes (see Figure 1.7). A hypercube has a high bandwidth and can solve

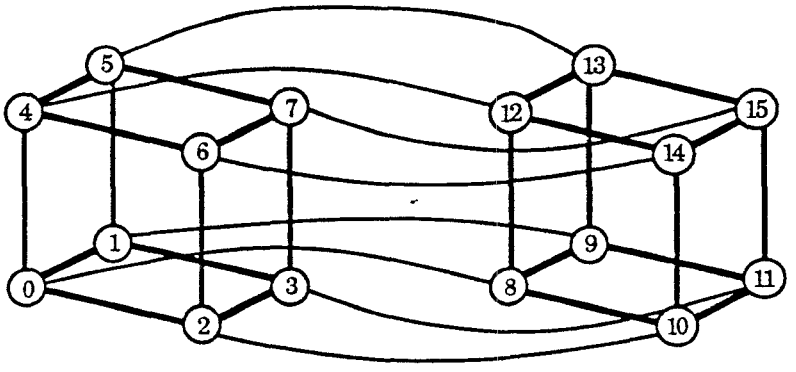


Figure 1.7: 4-dimensional hypercube network

many problems very fast. However they are difficult to build, since there are $\log P$ wires conected to each processor. Shuffle-exchange and cube-connected cycles have a performance comparable to hypercubes but have fixed number of interconnections.

Butterfly network:

In a butterfly network, the $P = (q+1)2^q$ processors are organized into $(q+1)$ ranks with 2^q processors in each rank. Processors i and i^r in rank $r+1$ are connected to processors i and i^r in rank r, where i^r is denoted by $i_{q-1} \ldots \bar{i_r} \ldots i_0$, $0 \leq r \leq q$, and $0 \leq i < 2^q$. These four connections form a 'butterfly' pattern (see Figure 1.8). This network is ideally suited for computing fast transforms like FFT (Fast Fourier Transform). It can also be used to solve problems like merging, sorting, etc. very fast. Actually this network is equivalent to the shuffle-exchange and is slightly less powerful than the hypercube.

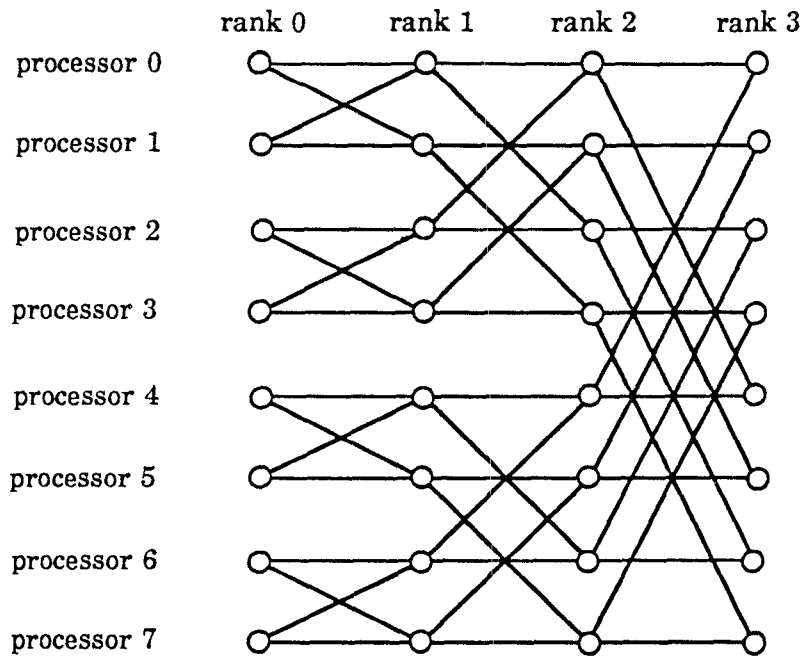


Figure 1.8: Butterfly network for $q = 3$ and $P = 4.2^3$

1.3 Architecture evaluation

Here we would like to make some remarks about the important features in the evaluation of a VLSI architecture. The basic parameters of any VLSI computation are *chip area* A and *computation time* T. Chip area is a measure of the fabrication cost and computation time is a measure of the operating cost. While VLSI designers measure the chip area in mm^2 of silicon and computation time in nanoseconds, VLSI computation theorists estimate the area and computation time asymptotically using the VLSI model of computation.

In the VLSI model of computation [6, 54], a VLSI circuit is a computation graph whose vertices are nodes which perform simple logical functions, and

whose edges are wires which carry signals to and from nodes. The total area A is the number of unit squares in the smallest rectangle that encloses the circuit consisting of nodes and wires. The computation time T is the number of time units between the appearance of the first input bit on some port and the appearance of the last output bit on some port.

The VLSI model of computation relies on asymptotic analysis to evaluate the area and time complexities. Such an analysis has the advantage of being very simple and yet pointing out the bottlenecks of the design. Moreover it provides a simple framework for the evaluation and explanation of various designs. However such an analysis does not contain the constant factors that are crucial when the number of sample points is small. Also, VLSI technology still cannot support the one-chip design for a large number of sample points. So a theoretically optimal design may not necessarily be the 'best' design in terms of actual hardware implementation. From a designers' point of view what is more important is whether the design is regular and simple, whether all the processors are active most of the time, whether the communication bandwidth between processors as well as with the external world is low, whether the data flow is regular, and so on. In this thesis we develop architectures which are on the one hand realizable using today's technology and on the other hand as close as possible to the theoretical optimality criteria.

1.4 Main Contributions

In this section we give an overview of the architectures that we developed for computing some important signal processing tasks, namely, one-dimensional Discrete Hartley and Discrete Cosine transforms, d-dimensional linear separable transforms, template matching, and block matching. For each of these problems we briefly describe the importance, the definition, the existing schemes and the proposed scheme.

1.4.1 Discrete Hartley Transform and Discrete Cosine Transform

The Discrete Hartley transform (DHT) and the Discrete Cosine transform (DCT) are important transforms in signal processing. DCT is widely used to encode speech and video signals in data compression schemes because of its close performance to the statistically optimal Karhunen-Loève transform. DCT is also used for realizing filter banks in frequency-division multiplexing and time-division multiplexing systems. DHT is used in spectral analysis and other signal processing schemes. In fact, it has the same potential as the Discrete Fourier transform (DFT) and can be used in almost all schemes which employ DFT. DHT and DCT are favored in schemes which require the computation of multidimensional transforms, since both these transforms are real and do not require computations involving complex arithmetic (as in DFT).

Problem definition:

The one-dimensional DHT of N points is defined [2] by

$$H(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n) cas\,(\frac{2\pi}{N} nk), \qquad 0 \leq k \leq N-1,$$

where $cas(x) = \cos(x) + \sin(x)$. The one-dimensional DCT of N points is defined [1] by

$$C(k) = \frac{2}{N} e(k) \sum_{n=0}^{N-1} x(n) \cos[\frac{\pi}{2N}(2n+1)k], \qquad 0 \leq k \leq N-1,$$

where,

$$e(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } k = 0 \\ 1 & \text{otherwise} \end{cases}$$

Our aim here is to develop systolic array architectures for one-dimensional DHT and DCT when N is factorizable into mutually prime factors N_1 and N_2.

Existing schemes:

There exist linear systolic array architectures [14, 55] as well as two-dimensional systolic array architectures [14] for computing DCT. While the former is a direct realization of the matrix vector algorithm, the latter computes DCT by multiplying the output of DFT over $(N_1 \times N_2)$ points with appropriate factors. There also exist general-purpose architectures for DCT [57]. To the best of the author's knowledge no systolic array architectures have been proposed for DHT.

Our contribution:

Our approach for computing one-dimensional DHT and DCT consists of mapping the one-dimensional transforms over N points into two-dimensional transforms over $(N_1 \times N_2)$ points, and then realizing them by two-dimensional systolic arrays. The algorithm consists of computing one-dimensional transforms over columns and rows of the two-dimensional data array, followed by some adjustments. Our architecture is strictly systolic. All the components are completely pipelined, resulting in very high throughput.

There is a considerable reduction in hardware as a result of mapping the one-dimensional transform into two-dimensions. This is because the same hardware can be used to compute on multiple columns (rows). The reduction in hardware is at the expense of a larger computation time. The number of multipliers in our architecture is considerably less than those of the existing architectures [13, 14, 55]. This feature is important since the chip area is dominated by the area occupied by the multipliers. Let $f(N)$ be the number of multipliers required to compute an N point DFT on real data. Then the number of multipliers in our architecture for DHT is $f(N_1) + f(N_2)$ and the number of multipliers in our architecture for DCT is $3f(N_1) + 3f(N_2)$. In comparison the number of multipliers in the existing architectures for DCT is $2f(N_1) + 4f(N_2) + 4N_2$ in [14], N in [55], and N^2 in [13]. Moreover the size of the multipliers in our architecture are smaller compared to those of [14, 55], since the multipliers are fixed and can hence be embedded in the hardware.

1.4.2 Multi-dimensional Transforms

Multi-dimensional transforms are a powerful tool for multi-dimensional signal processing. Some of the important applications of such transforms are in the areas of spectrum analysis, signal reconstruction, speech processing, tomography, computer vision and image processing. Examples of such transforms are DFT, DHT, and DCT. We refer to these transforms as DXT.

Problem definition:

An $(N \times N \times \ldots \times N)$ d-dimensional linear separable transform is defined by

$$X(k_1, k_2, \ldots, k_d) = \sum_{n_d} \cdots \sum_{n_2} \sum_{n_1} x(n_1, n_2, \ldots, n_d) \alpha_1(n_1, k_1) \alpha_2(n_2, k_2) \cdots \alpha_d(n_d, k_d),$$

where the α_i's are the one-dimensional transform functions (like DFT, DHT, DCT), $0 \leq k_i, n_i \leq N-1$ for $1 \leq i \leq d$.

Our aim here is to develop optimal architectures for computing such transforms. An optimal architecture is one whose AT^2 matches the AT^2 lower bound for computing such transforms, where A is the area, T is the computation time. The AT^2 lower bound for d-dimensional DXT is $AT^2 = \Omega(n^2 \log^2 M)$, where $n = N^d$, and $M = N + 1$.

Existing schemes:

The schemes for computing $(N \times N \times \ldots \times N)$ d-dimensional transforms consist of computing either a sequence of one-dimensional transform each of size

N, or a matrix-vector product. The architecture of [24] which is based on the first scheme, has an area $A = O(n^2)$, computation time $T = O(d \log^2 n)$ and is not optimal. The architecture of [4] which is based on the second scheme, has an area $A = O(n^2 \log^2 M/T^2)$ and is optimal for all T in the range $[\Omega(\log n), O(\sqrt{n \log M})]$.

Our contribution:

Our approach for computing $(N \times N \times \ldots \times N)$ d-dimensional DXT is based on the first scheme. The architecture consists of one-dimensional DXT(N) computation units which compute DXT(N) over one index, and permutation units which order data so that in the next iteration DXT(N) can be computed over the next index. Our architecture has an area $A = O(N^{d+2a})$ and computation time $T = O(dN^{\frac{d}{2}-a}b)$ for all a in the range $\frac{1}{2}\log_N b \leq a \leq \frac{d}{2}$, where $b = O(\log M)$ is the precision. Thus an architecture with small value of a is one with small area while an architecture with large value of a is one with small computation time.

Our scheme consists of a *family* of architectures with area-time trade-offs. The architecture of [24] (which is for the case when the input data is in a single file) is a member of this family. For the same single file input data configuration, the computation time of our design is $O(d \log M)$ compared to $O(d \log^2 n)$ of [24]. The architecture of [4] is optimal for the same range of area and computation time as our architecture. For the case when d is constant, our architecture is superior to that of [4] since it is simpler, more regular, more modular, and hence easier to implement in VLSI. It is not unreasonable to make such an assumption

about the constancy of d, since for all known applications of multidimensional transforms, $d \leq 4$.

1.4.3 Template Matching

Image template matching is used to find the similarity or disagreement between a template and an equivalent size area of the input image. It is representative of many window based image processing tasks. It is used in image location, scene matching, filtering, edge detection, finding lines, spots, curves, etc.

Problem definition:

Let the input image be of size $(N \times N)$ and let the template be of size $(K \times K)$, then template matching is the inner product of the template with an equivalent size area of the input image.

$$TM[i,j] = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} I[i+m, j+n] * W[m,n], \quad 0 \leq i,j \leq N-K.$$

Our aim here is to develop a systolic architecture for template matching which efficiently balances the computations in the processor array with the I/O bandwidth. The input image is read in the line scan mode [3].

[3] In line scan mode, the pixels are run left to right and from top to bottom in a frame.

Existing schemes:

The systolic and semi-systolic architectures for template matching which address the I/O bandwidth problem can be divided into two classes: architectures which store part of the input on-chip, and architectures which store part of the input in an external memory. The I/O bandwidth for architectures with an external memory is K pixels per clock cycle [27, 51], while that of architectures with on-chip storage in only one pixel per clock cycle [17, 41]. The on-chip storage devices are FIFO line memory [17], shift-buffer pipeline [41], shift registers, RAM, etc. There are other architectures [28] which balance the large bandwidth of the first class of architectures and the large on-chip storage of the second class by including both an external memory and an on-chip storage, albeit smaller. Such architectures are suitable for large templates.

Our contribution:

Our architecture for computing template matching consists of a semi-systolic linear array of processors. A part of the input image is stored on-chip in shift registers distributed among the processors. The number of processors in our architecture is a function of the frame size, the template size and the internal clock cycle. This feature makes our architecture more versatile compared to those of [17, 28, 41], where the number of processors is fixed at K^2 irrespective of the frame specifications. An additional advantage of our scheme is that since the on-chip storage is distributed among the processors, no additional circuitry is required to ensure regular data flow from the on-chip storage devices to the

processor array.

1.4.4 Block matching

In many applications of digital image processing like video-phone and teleconferencing, there is very little motion in the entire scene. Thus successive frames are highly correlated. In such applications, algorithms based on motion compensation can be used to reduce information redundancy, and thereby achieve high data compression. Block matching is a popular motion compensation algorithm which is used to remove interframe redundancy.

In block matching the current frame is divided into reference blocks, and for each reference block, the best matched block in the previous frame is searched. It is assumed that the best matched block lies in an area surrounding the position of the reference block. The displacement between the reference block position and that of the best matched one in the previous frame is called the *displacement vector*. The receiver can construct the current frame using the available previous frame, the displacement vector and the block difference. This scheme results in tremendous bit rate reduction in interframe coding.

Problem definition

Let the current frame of size $(N \times N)$ be divided into reference blocks of size $(K \times K)$. Let $B_c(i,j)$ be one such reference block whose top-leftmost coordinate is (i,j). $B_c(i,j)$ is matched with candidate blocks in the previous frame which lie

in a search area $S(i,j)$. Let $L_{i,j}(\Delta i, \Delta j)$ be the *block-distance* between reference block $B_c(i,j)$ and candidate block $B_p(i+\Delta i, j+\Delta j)$

$$L_{i,j}(\Delta i, \Delta j) = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} |x_c(i+m, j+n) - x_p(i+m+\Delta i, j+n+\Delta j)|,$$

$-\frac{q}{2} \leq \Delta i, \Delta j \leq \frac{q}{2}$. The best match criteria is based on the minimum value of block-distance. The displacement vector $(\Delta\hat{i}, \Delta\hat{j})_{i,j}$ for reference block $B_c(i,j)$ is given by

$$(\Delta\hat{i}, \Delta\hat{j})_{i,j} = \min_{\Delta i, \Delta j}{}^{-1} L_{i,j}(\Delta i, \Delta j).$$

Our aim here is to develop a systolic architecture for block matching which efficiently balances the computations in the processor array with the I/O bandwidth. Data from the current frame and the previous frame memory are read in line scan mode.

Existing schemes

There exist many systolic and semi-systolic architectures for block matching [7, 18, 20, 30, 61] which compute the displacement vectors very efficiently but are not so efficient in handling the I/O operations. Block matching, like template matching, is an operation with a very high I/O bandwidth. A way of reducing the large I/O bandwidth is by storing part of the image which is accessed multiple times, on-chip in a storage device. Shift registers, RAMs, line buffers, register chains, etc [7, 18, 30] are popular on-chip storage devices. In almost all these architectures, the input data is fed from the frame memories

into the on-chip storage devices in block scan mode [4].

Our contribution

Our architecture for computing block matching consists of a semi-systolic linear array of processors which handles both the computations and the I/O bandwidth problem efficiently. Data from both the current frame and the previous frame-memory are read in line scan mode and stored on-chip in shift registers distributed among the processors. For this mode of data access, the linear array is a more suitable architecture (compared to the two-dimensional array). The number of processors in our architecture is a function of the frame specification (frame size, frame frequency), search area size, and internal clock rate. Our architecture can be very easily reconfigured for different problem specifications unlike the existing architectures where the number of processors is fixed [7, 18, 20, 30]. The data flow in our architecture is very regular. There is no additional circuitry to ensure that the right data are incident on the processor array as in [18]. The number of I/O pins in our architecture is only two compared to $1 + (q + K)^2/K^2$ in [7].

[4] In block scan mode, blocks of $(K \times K)$ pixels are run left to right and from top to bottom in a frame.

1.5 Thesis organization

In this thesis we present the systolic and semi-systolic architectures that we developed for computing some important real-time signal processing tasks.

In Chapter 2 we develop completely pipelined, bit-serial systolic array architectures for computing one-dimensional DHT and DCT when the number of sample points N is factorizable into mutually prime factors N_1 and N_2.

In Chapter 3 we develop a family of optimal architectures with area-time trade-offs for computing $(N \times N \times \ldots \times N)$ d-dimensional linear separable transforms. The criteria for optimality is as defined by VLSI complexity theory.

In Chapters 4 and 5 we develop semi-systolic linear array architectures for computing template matching and block matching respectively. The architecture handles the computations as well as the I/O bandwidth requirements efficiently.

In Chapter 6 we summarize the results obtained in this dissertation and suggest some directions for future research in this area.

Chapter 2

One-dimensional DHT and DCT

2.1 Introduction

In this chapter we address the problem of developing systolic array architectures for computing one-dimensional DHT and DCT over N points, when N is factorizable into mutually prime factors N_1 and N_2. The one-dimensional transform over N points is mapped into a two-dimensional transform over $(N_1 \times N_2)$ points, such that the algorithm consists of computing one-dimensional transform over columns and rows of the two-dimensional data array, followed by some adjustments. The hardware requirement is considerably reduced because of this mapping. The number of multipliers in our architecture is significantly less than those of the existing architectures [13, 14, 55]. The number of multipliers for our DHT architecture is $f(N_1) + f(N_2)$ and that of our DCT architecture is $3f(N_1) + 3f(N_2)$, where $f(N)$ is the number of multipliers required to compute

an N point DFT on real data. Our architecture consists of a few types of regular and modular units. All the units are completely pipelined, resulting in very high throughput.

The rest of the chapter is organized as follows. In Section 2.2 we define one-dimensional DHT and DCT and show how these transforms can be mapped into systolic architectures. In Section 2.3 we give the index mappings that map one-dimensional DHT and DCT to two-dimensional DHT and DCT, and show how the two-dimensional transforms can be mapped into systolic array architectures with reduced hardware. We give the details of the bit-serial systolic array implementations in Section 2.4. In Section 2.5 we make some concluding remarks.

2.2 Preliminaries

2.2.1 One-dimensional Discrete Hartley Transform (DHT)

The one-dimensional DHT of N points, DHT(N), is defined [2] by

$$H(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n) cas\,(\frac{2\pi}{N} nk), \qquad 0 \leq k \leq N-1, \tag{2.1}$$

where $cas(x) = \cos(x) + \sin(x)$. Since the factor $\frac{1}{\sqrt{N}}$ in equation 2.1 is a constant, it is sufficient to consider the DHT-like equation

$$H(k) = \sum_{n=0}^{N-1} x(n)\, cas(\frac{2\pi}{N} nk), \qquad 0 \leq k \leq N-1. \tag{2.2}$$

There are various schemes for computing one-dimensional DHT [2, 3, 25, 31, 52]. They are based on either decompositions of the form radix-2 decimation-in-time,

$$S_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix}, \hat{C}_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}, T_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}$$

Figure 2.1: Matrices S_4, $\hat{C}_4$, T_4 in the computation of DHT(4)

radix-2 decimation-in-frequency, split radix, radix-4, or Winograd transform, or recursion. We choose the Winograd-Hartley transform (WHT) algorithm [52] because of the reduced number of multiplications ($\approx N$). Moreover this algorithm can be mapped into an architecture with reduced number of multipliers. The WHT algorithm is expressed as $H = S_N \hat{C}_N T_N x$, where S_N is an $N \times J$ incidence [1] matrix, T_N is a $J \times N$ incidence matrix and $\hat{C}_N$ is a $J \times J$ diagonal matrix with real entries, $J \approx N$ for small N algorithms. This algorithm is very similar to the Winograd Fourier transform (WFT) algorithm [23], which is expressed as $F = S_N C_N T_N x$. $\hat{C}_N$ of WHT is related to C_N of WFT by $\hat{C}_N = Re[C_N] - Im[C_N]$. Figure 2.1 describes the matrices S_4, $\hat{C}_4$ and T_4 for computing DHT(4).

The WHT algorithm can be mapped into a systolic architecture in a way similar to the mapping of the WFT algorithm in [47]. The incidence matrices S_N and T_N are mapped into systolic units, called *summation* units. These units consist of arrays of 1-bit adders, 1-bit subtractors or 1-bit delays corresponding

[1] An incidence matrix is a matrix whose elements are -1, 0 or 1.

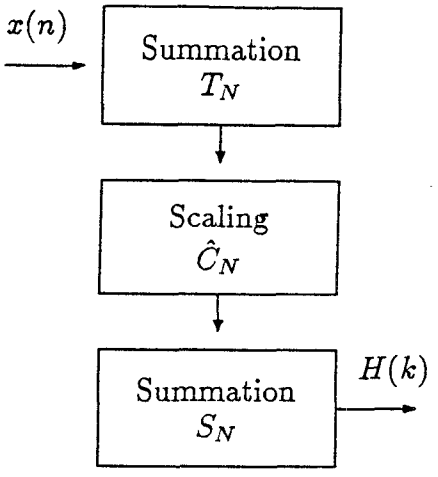


Figure 2.2: Block diagram for computing one-dimensional DHT(N)

to entries +1, -1 or 0 in the incidence matrices. The diagonal matrix $\hat{C}_N$ is mapped into a systolic unit, called *scaling* unit. This unit consists of an one-dimensional array of J fixed multipliers, corresponding to the J diagonal elements. Figure 2.2 describes the block diagram of the corresponding architecture.

The bottleneck of this architecture is the large number ($\approx N$) of multipliers in the scaling unit. The number of multipliers can be reduced to ($\approx N_1 + N_2$) by mapping the one-dimensional transform into two dimensions. In Section 2.3.1 we describe such a transformation.

2.2.2 One-dimensional Discrete Cosine Transform (DCT)

The one-dimensional DCT of N points, DCT(N), is defined [1] by

$$C(k) = \frac{2}{N} e(k) \sum_{n=0}^{N-1} x(n) \cos[\frac{\pi}{2N}(2n+1)k], \qquad 0 \leq k \leq N-1, \tag{2.3}$$

where,

$$e(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } k = 0 \\ 1 & \text{otherwise} \end{cases}$$

Since the factor $\frac{2}{N} e(k)$ results only in a slight modification of $C(k)$, it is sufficient to consider the DCT-like equation

$$C(k) = \sum_{n=0}^{N-1} x(n) \cos[\frac{\pi}{2N}(2n+1)k] \qquad 0 \leq k \leq N-1. \tag{2.4}$$

There are various schemes for computing one-dimensional DCT [11, 26, 39, 43, 46, 56]. They are based on either factorization of the DCT matrix, or rotation of the output of a Fourier or Hartley transform, or recursion. We choose a scheme based on rotation of the Hartley transform [43].

In the method proposed by Malvar [43], the input $x(n)$ is permuted so that the cosine argument in equation 2.4 is changed to $\cos[\frac{\pi}{2N}(4n+1)k]$. DHT can then be used to compute $C(k)$. Let $y(n)$ be a permutation of $x(n)$ defined by

$$y(n) = \begin{cases} x(2n) & 0 \leq n \leq \lceil N/2 \rceil - 1 \\ x(2N-2n-1) & \lceil N/2 \rceil \leq n \leq N-1. \end{cases} \tag{2.5}$$

Then

$$C(k) = \frac{1}{2}[H(k)cas(-\frac{k\pi}{2N}) + H(N-k)cas(\frac{k\pi}{2N})], \tag{2.6}$$

where $H(k)$ is the DHT of $y(n)$. Let $\sum_n y(n) \cos[\frac{\pi}{2N}(4n+1)k]$ be denoted by DCT$\acute{}$ of $y(n)$. Then $C(k)$ in equation 2.6 is the DCT$\acute{}$ of $y(n)$. We choose the

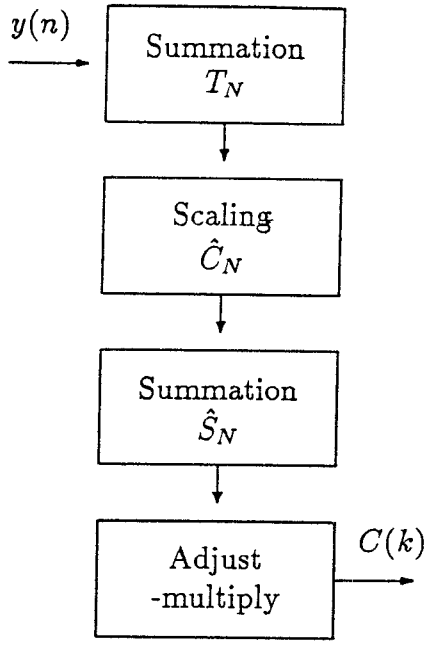


Figure 2.3: Block diagram of one-dimensional DCT(N)

Winograd-Hartley transform algorithm to compute DHT of $y(n)$. In that case the number of multiplications is only $\approx 3N$.

The above algorithm can be mapped into a systolic architecture in the following way. $H(k)$ can be computed in a systolic manner by the summation and scaling units as described in Section 2.2.1. $C(k)$ can be computed in a systolic fashion from $H(k)$ (see equation 2.6) by a systolic unit called the *adjust-multiply* unit. This unit consists of an one-dimensional array of N subunits, each of which consists of two fixed multipliers and one 1-bit adder. Figure 2.3 describes the block diagram of this architecture.

In order that the adjust-multiply unit (see Figure 2.3) be systolic, $H(k)$ and $H(N-k)$ have to be adjacent to each other. This is achieved by modifying S_N

$$P_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \qquad \hat{S}_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 2.4: Matrices P_4 and $\hat{S}_4$ in the computation of one-dimensional DCT(4)

to $\hat{S}_N$ in the following way. Let $G = S_N \hat{C}_N T_N x$ and let $\hat{G}$ be a permutation of G, such that $\hat{G} = (G(0)\, G(1)\, G(N-1) \ldots G(k)\, G(N-k) \ldots)^T$, where $G(i)$ represents the ith row of G, $1 \leq k \leq \lfloor N/2 \rfloor$. When N is even, there is only one $G(N/2)$ in $\hat{G}$. $\hat{G}$ can be expressed as $P_N G$, where P_N is a permutation matrix with the following characteristics :

$$\begin{aligned} & & P_N(0,0) &= 1 \\ \text{For even } i, \quad & i \neq 0, & P_N(i, N - i/2) &= 1 \\ & \text{For odd } i, & P_N(i, \lceil i/2 \rceil) &= 1 \end{aligned}$$

Note that since P_N is a permutation matrix, $\hat{S}_N$ is an incidence matrix, and hence can be directly embedded into the summation unit. Figure 2.4 describes the permutation matrix P_4 and the incidence matrix $\hat{S}_4 = P_4 S_4$.

As in the case of DHT, the bottleneck of this architecture is the large number of multipliers ($\approx N$ in the scaling unit and $\approx 2N$ in the adjust-multiply unit). This can be reduced to $\approx 3(N_1 + N_2)$ by mapping the one-dimensional transform into two dimensions. In Section 2.3.2 we describe such a transformation.

2.2.3 Related work

We have seen earlier in this section that the existing algorithms for computing one-dimensional DHT and DCT over N points can be divided into two classes: N is a power of 2 and N is factorizable into mutually prime factors N_1 and N_2. For the case when N is a power of 2, an FFT-like structure is obtained for both DHT and DCT. For the case when $N = N_1 N_2$ and N_1, N_2 are mutually prime, the one-dimensional transform is mapped into a two-dimensional transform over $(N_1 \times N_2)$ points with appropriate input and output index mappings.

When N is a power of 2, the VLSI architectures for DCT map the butterfly-like flow diagram of a fast algorithm directly into silicon [12, 29, 53]. The multipliers are replaced by ROM accumulators which incorporate distributed arithmetic in the architectures of [12, 53]. The architectures for DHT consist of CORDIC processors and systolic shuffle units [44], or their pipelined variations [48].

When N is factorizable into mutually prime factors, the DCT implementation of [57] is not special-purpose and contains complicated arithmetic and control units. Recently [14] proposed an algorithm for DCT with the additional constraint that N_1 and N_2 are odd. The architecture of [14] consists of two DFT computation units of [47] and an additional multiplier unit to rotate the DFT output. The number of multipliers in their architecture is $2f(N_1)+4f(N_2)+4N_2$, where $f(N)$ is the number of multipliers required to compute an N point DFT on real data.

Both DHT and DCT can be implemented by a systolic architecture based on the matrix-vector algorithm. There exist linear array architectures for DCT [14, 55] which consist of N multipliers. The data flow of the DCT coefficients for both these architectures is complicated. The architecture of [13] is a CCD implementation of the matrix-vector algorithm and consists of N^2 fixed multipliers.

2.3 Mapping into two dimensions

Any one-dimensional linear transform over N points,

$$f(k) = \sum_n a(k,n)x(n), \qquad 0 \leq n,k \leq N-1 \tag{2.7}$$

can be mapped into a two-dimensional transform over $(N_1 \times N_2)$ points, $N = N_1N_2$, by choosing the input and output index mappings appropriately. We assume that N_1 and N_2 are mutually prime. Let $\mathcal{N}$ be the set of integers 0 through $N-1$, let $\mathcal{N}_1$ be the set of integers 0 through N_1-1 and let $\mathcal{N}_2$ be the set of integers 0 through N_2-1. Let $g_1(n) = (n_1, n_2)$ be the input index mapping from $\mathcal{N}$ to $\mathcal{N}_1 \times \mathcal{N}_2$, $n \in \mathcal{N}$, $n_1 \in \mathcal{N}_1$, $n_2 \in \mathcal{N}_2$ and let $g_2(k) = (k_1, k_2)$ be the output index mapping from $\mathcal{N}$ to $\mathcal{N}_1 \times \mathcal{N}_2$, $k \in \mathcal{N}$, $k_1 \in \mathcal{N}_1$, $k_2 \in \mathcal{N}_2$. If g_1 and g_2 are one-to-one, then $f(k)$ can be mapped into a two-dimensional function $f(k_1, k_2)$, where

$$f(k_1,k_2) = \sum_{n_2}\sum_{n_1} a(k_1,k_2,n_1,n_2)x(n_1,n_2). \tag{2.8}$$

The main motive behind this transformation is to map the computation into a systolic two-dimensional array of size $(N_1 \times N_2)$. The ideal transformation

would be such that expression 2.8 can be computed by a set of column(row) operations on the two-dimensional data followed by a set of row(column) operations. The number of bit-serial systolic steps [2] will then be reduced from $O(bN)$ to $O(b(N_1+N_2))$, where b is the precision. In this section we briefly discuss such transformations for one-dimensional DHT and DCT and then sketch procedures to execute them by systolic arrays.

2.3.1 Mapping of DHT

One-dimensional DHT, $H(k)$, can be mapped into a two-dimensional DHT, $H(k_1, k_2)$,

$$H(k_1,k_2) = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x(n_1,n_2) cas\left(\frac{2\pi}{N_1} n_1 k_1 + \frac{2\pi}{N_2} n_2 k_2\right), \tag{2.9}$$

with the following input and output index mappings [2]

$$\begin{aligned} n &= N_2 n_1 + N_1 n_2 \bmod N \\ k &= (N_2^{-1} \bmod N_1) N_2 k_1 + (N_1^{-1} \bmod N_2) N_1 k_2 \bmod N. \end{aligned} \tag{2.10}$$

Since in general, $cas(\alpha+\beta) \neq cas(\alpha)cas(\beta)$, equation 2.9 is not a simple DHT over columns of $x(n_1, n_2)$ followed by a DHT over rows.

Figure 2.5 illustrates the input and output index mappings for an example where $N = 20$ ($N_1 = 4$, $N_2 = 5$). The mappings are represented in the form of tables with N_1 rows and N_2 columns, such that location (n_1, n_2) of the input

[2] A bit-serial systolic step is a 1-bit computation step (add, subtract, delay) in bit-serial systolic architectures.

n_1 \ n_2	0	1	2	3	4
0	0	4	8	12	16
1	5	9	13	17	1
2	10	14	18	2	6
3	15	19	3	7	11

n table

k_1 \ k_2	0	1	2	3	4
0	0	16	12	8	4
1	5	1	17	13	9
2	10	6	2	18	14
3	15	11	7	3	19

k table

Figure 2.5: Input and output index mappings for DHT over (4×5) points

index mapping table contains n and location (k_1, k_2) of the output index mapping table contains k (refer to equation 2.10).

Sorensen, et.al. [52] proposed a method of computing $H(k_1, k_2)$ by computing DHT(N_1) over columns of the data array, followed by combining some of the elements of the intermediate array and then computing DHT(N_2) over rows of the resulting array. The method proposed by Bracewell, et.al. [3], on the other hand, consists of computing DHT(N_1) over columns of the data array followed by DHT(N_2) over rows of the intermediate array and then combining some of the elements of the resulting array.

The scheme proposed in [52] is summarized as follows. Let

$$A(k_1, n_2) = \sum_{n_1=0}^{N_1-1} x(n_1, n_2) cas \left(\frac{2\pi}{N_1} n_1 k_1\right), \tag{2.11}$$

be the outcome after computing DHT(N_1) of the columns, and let

$$B(k_1, n_2) = \frac{1}{2}(A(k_1, n_2) + A(N_1 - k_1, n_2) + A(k_1, N_2 - n_2) - A(N_1 - k_1, N_2 - n_2)). \quad (2.12)$$

Then $H(k_1, k_2)$ is obtained by computing DHT(N_2) over rows of array B.

$$H(k_1, k_2) = \sum_{n_2=0}^{N_2-1} B(k_1, n_2) cas\left(\frac{2\pi}{N_2} n_2 k_2\right) \quad (2.13)$$

The algorithm for DHT(N) [52] can then be expressed as follows.

1. Compute DHT(N_1) over columns.
2. Compute B from A (equation 2.12).
3. Compute DHT(N_2) over rows.

Steps 1 and 3 can be mapped directly into a systolic architecture consisting of summation and scaling units (refer to Section 2.2.1). Step 2 can be computed by a systolic unit after modifying Steps 1 and 3 appropriately [8].

We next describe the scheme proposed by [3] for computing $H(k_1, k_2)$. We will describe it in details because of its similarity with the DCT algorithm (see Section 2.3.2). Let $T(k_1, k_2)$ be the temporary outcome obtained after computing DHT of the columns followed by DHT of the rows.

$$T(k_1, k_2) = \sum_{n_2=0}^{N_2-1} \left\{ \sum_{n_1=0}^{N_1-1} x(n_1, n_2) cas(\frac{2\pi}{N_1} n_1 k_1) \right\} cas(\frac{2\pi}{N_2} n_2 k_2) \quad (2.14)$$

Then $H(k_1, k_2)$ can be expressed as the sum of four temporary outcomes.

$$H(k_1, k_2) = \frac{1}{2}(T(k_1, k_2) + T(N_1 - k_1, k_2) + T(k_1, N_2 - k_2) - T(N_1 - k_1, N_2 - k_2)) \quad (2.15)$$

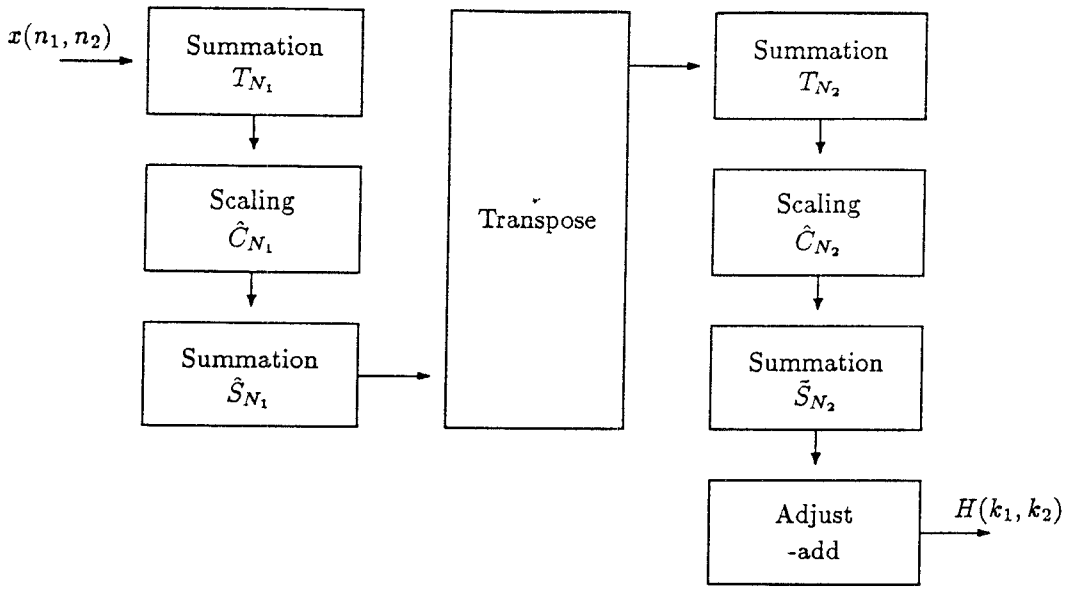


Figure 2.6: Block diagram for computing two-dimensional DHT over $(N_1 \times N_2)$ points, where N_1, N_2 are mutually prime

Note that $H(N_1 - k_1, k_2)$, $H(k_1, N_2 - k_2)$ and $H(N_1 - k_1, N_2 - k_2)$ can also be expressed as sums of $T(k_1, k_2)$, $T(N_1 - k_1, k_2)$, $T(k_1, N_2 - k_2)$ and $T(N_1 - k_1, N_2 - k_2)$. The algorithm for DHT(N) [3] is as follows.

1. Compute DHT(N_1) over columns.

2. Compute DHT(N_2) over rows.

3. Compute H from T (equation 2.15).

Steps 1 and 2 can be mapped directly into summation and scaling units (refer to Section 2.2.1). Since DHT(N_1) is computed over columns and DHT(N_2) is computed over rows, a systolic transpose unit is needed to transpose the

intermediate result obtained after computing DHT(N_1) over columns. Step 3 can be mapped into a systolic unit called the *adjust-add* unit. Figure 2.6 shows the interconnections of the various units for DHT over ($N_1 \times N_2$) points, where N_1, N_2 are mutually prime.

We next describe the modifications in the algorithm for DHT(N) that are required in order that the adjust-add unit (see Figure 2.6) be systolic. The algorithm has to be tailored so that $T(k_1, k_2)$, $T(N_1 - k_1, k_2)$, $T(k_1, N_2 - k_2)$ and $T(N_1 - k_1, N_2 - k_2)$ are adjacent to each other. This can be done by permuting the rows of $Y(k_1, n_2)$ after computing Step 1 of the algorithm and by permuting the columns of $T(k_1, k_2)$ after computing Step 2. The row permutation is such that $Y(k_1, n_2)$ and $Y(N_1 - k_1, n_2)$ are adjacent to each other for $1 \leq k_1 \leq \lceil N_1/2 \rceil - 1$, $0 \leq n_2 \leq N_2 - 1$. Such a permutation is possible by embedding $\hat{S}_{N_1}$ (instead of S_{N_1}) in the summation unit of DHT(N_1), where $\hat{S}_{N_1} = P_{N_1} S_{N_1}$ (refer to Section 2.2.2). Similarly the column permutation is such that $T(k_1, n_2)$ and $T(k_1, N_2 - n_2)$ are adjacent to each other for $0 \leq k_1 \leq N_1 - 1$ and $1 \leq n_2 \leq \lceil N_2/2 \rceil - 1$. This is possible by embedding $\hat{S}_{N_2}$ (instead of S_{N_2}) in the summation unit of DHT(N_2), where $\hat{S}_{N_2} = P_{N_2} S_{N_2}$. Note that since P_{N_1} and P_{N_2} are permutation matrices, $\hat{S}_{N_1}$ and $\hat{S}_{N_2}$ are incidence matrices and hence can be directly mapped into summation units. Figure 2.7 illustrates the row permutation in intermediate data array Y as a result of embedding $\hat{S}_4$ in the summation unit in the computation of DHT(4) over columns for the example where $N = 20$ ($N_1 = 4$, $N_2 = 5$).

The next step in the procedure is to efficiently compute $H(k_1, k_2)$, $H(N_1 -$

$$\begin{bmatrix} Y_{0,0} & Y_{0,1} & Y_{0,2} & Y_{0,3} & Y_{0,4} \\ Y_{1,0} & Y_{1,1} & Y_{1,2} & Y_{1,3} & Y_{1,4} \\ Y_{3,0} & Y_{3,1} & Y_{3,2} & Y_{3,3} & Y_{3,4} \\ Y_{2,0} & Y_{2,1} & Y_{2,2} & Y_{2,3} & Y_{2,4} \end{bmatrix}$$

Figure 2.7: Row permutation in intermediate data array Y after computing DHT(4) over columns during the computation of DHT over (4×5) points

$k_1, k_2)$, $H(k_1, N_2 - k_2)$ and $H(N_1 - k_1, N_2 - k_2)$ from $T(k_1, k_2)$, $T(N_1 - k_1, k_2)$, $T(k_1, N_2 - k_2)$ and $T(N_1 - k_1, N_2 - k_2)$ using a systolic unit. Let $A = T(k_1, k_2)$, $B = T(N_1 - k_1, k_2)$, $C = T(k_1, N_2 - k_2)$, $D = T(N_1 - k_1, N_2 - k_2)$. Then $H(k_1, k_2) = \frac{1}{2}(A + B + C - D)$. In the algorithm proposed by [3], $H(k_1, k_2)$ is obtained by computing the 'diagonal excess' defined by $E = 1/2[(A+D)-(B+C)]$ and then doing in-place replacement of the form $A \leftarrow A - E$, $B \leftarrow B + E$, $C \leftarrow C + E$ and $D \leftarrow D - E$. Though this algorithm employs only seven adds and one shift per four outputs, the systolic implementation is cumbersome. Since the adjustments for A, B, C and D are different, an additional control input line is required to distinguish between data $T(k_1, k_2)$ and $T(N_1 - k_1, k_2)$ for all k_2, that is, between A *and* B, C *and* D. This increases the complexity of the circuit.

Here we suggest a method to compute H from T with simpler systolic implementation. Let Z be the new temporary outcome defined by

$$\begin{aligned} Z(k_1, k_2) &= \frac{1}{2}[T(k_1, k_2) + T(k_1, N_2 - k_2)] \\ Z(k_1, N_2 - k_2) &= \frac{1}{2}[T(k_1, k_2) - T(k_1, N_2 - k_2)] \end{aligned} \tag{2.16}$$

for $1 \leq k_1 \leq N_1 - 1$ and $1 \leq k_2 \leq \lceil N_2/2 \rceil - 1$. Then H can be defined by

$$\begin{aligned} H(k_1, k_2) &= Z(k_1, k_2) + Z(N_1 - k_1, N_2 - k_2) \\ H(k_1, N_2 - k_2) &= Z(k_1, k_2) - Z(N_1 - k_1, N_2 - k_2). \end{aligned} \tag{2.17}$$

Note that $H(k_1, 0) = Z(k_1, 0) = T(k_1, 0)$, $H(0, k_2) = Z(0, k_2) = T(0, k_2)$ and in addition, if N_1 is even, $H(N_1/2, k_2) = Z(N_1/2, k_2) = T(N_1/2, k_2)$, and if N_2 is even, $H(k_1, N_2/2) = Z(k_1, N_2/2) = T(k_1, N_2/2)$. This scheme employs only one add per output. Moreover no additional control input line is required. There is no need for an extra unit to compute Z from T (equation 2.16). $Z(k_1, k_2)$ can be obtained at the end of Step 2 (instead of $T(k_1, k_2)$) by modifying the matrix $\hat{S}_{N_2}$ that is embedded in the summation unit of DHT(N_2) in the following way.

Let $\acute{T} = \hat{S}_{N_2} \hat{C}_{N_2} T_{N_2} Y$, where $\acute{T}$ is a permutation of the temporary matrix T such that $T(k_1, k_2)$, $T(N_1 - k_1, k_2)$, $T(k_1, N_2 - k_2)$ and $T(N_1 - k_1, N_2 - k_2)$ are adjacent to each other, Y is the output after computing DHT(N_1) over columns and $\hat{S}_{N_2}$, $\hat{C}_{N_2}$ and T_{N_2} are defined as before. Z can be expressed as $Q_{N_2} \acute{T}$, where the matrix Q_{N_2} has the following characteristics :

$$\begin{aligned} & & Q_{N_2}(0,0) = 1 & & \\ & \text{For odd } i, & Q_{N_2}(i,i) = 0.5, & \quad Q_{N_2}(i, i+1) = 0.5 \\ & \text{For even } i \text{ and } i \neq 0, & Q_{N_2}(i,i) = -0.5, & \quad Q_{N_2}(i, i-1) = 0.5, \end{aligned}$$

$1 \leq i \leq N_2 - 1$. In addition, if N_2 is even, $Q(N_1 - 1, N_2 - 1) = 1$. Thus by modifying $\hat{S}_{N_2}$ to $\tilde{S}_{N_2}$, where $\tilde{S}_{N_2} = Q_{N_2} \hat{S}_{N_2}$, $Z(k_1, k_2)$ is obtained at the

$$Q_5 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0.5 & -0.5 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0.5 & -0.5 \end{pmatrix}, \qquad \tilde{S}_5 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 1 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Figure 2.8: Matrices Q_5 and $\tilde{S}_5$ in the computation of DHT(5)

end of Step 2. The nature of Q_{N_2} and $\hat{S}_{N_2}$ [3] is such that their product is an incidence matrix. This means that $\tilde{S}_{N_2}$ can be mapped into a summation unit. Figure 2.8 describes the matrices Q_5 and $\tilde{S}_5$. Figure 2.9 describes the column permutation in data array Z as a result of embedding $\tilde{S}_5$ in the summation unit in the computation of DHT(5) over rows for the example where $N = 20$ ($N_1 = 4$, $N_2 = 5$). Note that $Z(k_1, k_2)$, $Z(N_1 - k_1, k_2)$, $Z(k_1, N_2 - k_2)$ and $Z(N_1 - k_1, N_2 - k_2)$ are adjacent to each other.

The 'adjust-add' unit in Figure 2.6 is a systolic unit which computes H from Z (see equation 2.17). Its design is simpler than that of the systolic unit which computes H from T. We will describe it in Section 2.4.

[3] For all known small N algorithms, $\hat{S}_N$ is such that either both or none of the (i, j)th and $(i+1, j)$th element is 0, for i odd and $1 \leq i \leq N-2$. This is because S_N is such that either both or none of the (i, j)th and $(N-1, j)$th element is 0, $1 \leq i \leq \lceil N/2 \rceil - 1$.

$$\begin{bmatrix} Z_{0,0} & Z_{0,1} & Z_{0,4} & Z_{0,2} & Z_{0,3} \\ Z_{1,0} & Z_{1,1} & Z_{1,4} & Z_{1,2} & Z_{1,3} \\ Z_{3,0} & Z_{3,1} & Z_{3,4} & Z_{3,2} & Z_{3,3} \\ Z_{2,0} & Z_{2,1} & Z_{2,4} & Z_{2,2} & Z_{2.3} \end{bmatrix}$$

Figure 2.9: Column permutation in intermediate data array Z after computing DHT(5) over rows during the computation of DHT over (4×5) points

2.3.2 Mapping of DCT

The existing algorithms for prime-factor decomposed DCT computation consists of computing DCT(N_1) over columns, followed by computing DCT(N_2) over rows, followed by combining the elements of the resulting data array [39, 57]. We develop a systolic architecture based on a modified version of the algorithm proposed by Lee [39]. Let $T(k_1, k_2)$ be the output after computing DCT(N_1) over columns followed by DCT(N_2) over rows [39].

$$T(k_1, k_2) = \sum_{n_2=0}^{N_2-1} \left\{ \sum_{n_1=0}^{N_1-1} x(n_1, n_2) \cos[\frac{\pi}{2N_1}(2n_1 + 1)k_1] \right\} \cos[\frac{\pi}{2N_2}(2n_2 + 1)k_2] \tag{2.18}$$

We have seen in Section 2.2.2 how one-dimensional DCT can be computed from one-dimensional DHT if the cosine argument is $\cos[\frac{\pi}{2N}(4n + 1)k]$ (instead of $\cos[\frac{\pi}{2N}(2n + 1)k]$). The cosine argument can be changed by appropriate choice of input index mapping.

n table :

n_1 \ n_2	0	1	2	3	4
0	0	7	15	16	8
1	10	2	5	13	18
2	19	12	4	3	11
3	9	17	14	6	1

Figure 2.10: Input index mapping for DCT over (4×5) points

We propose the following input index mapping $g_1(n) = (n_1, n_2)$. Let

$$\begin{aligned}
\bar{n}_1 &= n \bmod 2N_1 \\
\bar{n}_2 &= n \bmod 2N_2 \\
n_1 &= \begin{cases} \bar{n}_1/2 & \text{if } \bar{n}_1 \text{ is even} \\ N_1 - (\bar{n}_1 + 1)/2 & \text{otherwise} \end{cases} \\
n_2 &= \begin{cases} \bar{n}_2/2 & \text{if } \bar{n}_2 \text{ is even} \\ N_2 - (\bar{n}_2 + 1)/2 & \text{otherwise} \end{cases}
\end{aligned} \tag{2.19}$$

It can be easily shown that g_1 is a one-to-one mapping. Figure 2.10 describes the input index mapping for an example where $N = 20$ $(N_1 = 4, N_2 = 5)$. The index mapping is represented in the form of a table with N_1 rows and N_2 columns, where location (n_1, n_2) of the table contains n (see equation 2.19). With this index mapping let $\acute{T}(k_1, k_2)$ be the new temporary outcome defined by

$$\acute{T}(k_1, k_2) = \sum_{n_2=0}^{N_2-1} \left\{ \sum_{n_1=0}^{N_1-1} x(n_1, n_2) \cos[\frac{\pi}{2N_1}(4n_1 + 1)k_1] \right\} \cos[\frac{\pi}{2N_2}(4n_2 + 1)k_2]. \tag{2.20}$$

k_1 \ k_2	0	1	2	3	4
0	0	4	8	12	16
1	5	9	13	17	19
2	10	14	18	18	14
3	15	19	17	13	9

$\hat{k}$ table :

k_1 \ k_2	0	1	2	3	4
0	0	4	8	12	16
1	5	1	3	7	11
2	10	6	2	2	6
3	15	11	7	3	1

$\tilde{k}$ table :

Figure 2.11: Output index mapping for DCT over (4×5) points

Thus $\acute{T}(k_1, k_2)$ is the output obtained after computing DCT́ over columns followed by DCT́ over rows.

We define the output index mapping along the same lines as the input index mapping of [39]. Let $\hat{f}$ and $\tilde{f}$ be mappings from $\mathcal{N}_1 \times \mathcal{N}_2$ to $\mathcal{N}$ such that

$$\begin{aligned} \hat{k} = \hat{f}(k_1, k_2) &= \begin{cases} k_1 N_2 + k_2 N_1 & \text{if } k_1 N_2 + k_2 N_1 < N \\ 2N - (k_1 N_2 + k_2 N_1) & \text{otherwise} \end{cases} \\ \tilde{k} = \tilde{f}(k_1, k_2) &= |k_1 N_2 - k_2 N_1| \end{aligned} \tag{2.21}$$

Note that when k_1 or $k_2 = 0$, $\hat{k} = \tilde{k}$ otherwise, $\hat{k} \neq \tilde{k}$. Since $\hat{f}(N_1 - k_1, N_2 - k_2) = \hat{f}(k_1, k_2)$, the number of distinct values mapped by $\hat{f}$ when k_1 or $k_2 \neq 0$ is $(N_1 - 1)(N_2 - 1)/2$. Similarly the number of distinct values mapped by $\tilde{f}$ when k_1 or $k_2 \neq 0$ is $(N_1 - 1)(N_2 - 1)/2$. Since $\hat{k} \neq \tilde{k}$ when k_1 or $k_2 \neq 0$, the total number of values mapped by a combination of $\hat{f}$ and $\tilde{f}$ is $2(N_1 - 1)(N_2 - 1)/2 + N_1 + N_2 - 1 = N$. Figure 2.11 describes the output

index mapping for the example where $N = 20$ ($N_1 = 4$, $N_2 = 5$). As before the mapping is represented in the form of tables with N_1 rows and N_2 columns. The $\hat{k}$-table contains $\hat{f}(k_1, k_2)$ in location (k_1, k_2) of the table. Similarly the $\tilde{k}$-table contains $\tilde{f}(k_1, k_2)$ in location (k_1, k_2) of the table. For more details see [39].

When k_1 or $k_2 = 0$,

$$C(\tilde{f}(k_1, k_2)) = C(\hat{f}(k_1, k_2)) = \acute{T}(k_1, k_2)$$

otherwise,

$$\begin{aligned} C(\tilde{f}(k_1, k_2)) &= \acute{T}(k_1, k_2) + \acute{T}(N_1 - k_1, N_2 - k_2) \\ C(\hat{f}(k_1, k_2)) &= s(k_1, k_2)\{\acute{T}(k_1, k_2) - \acute{T}(N_1 - k_1, N_2 - k_2)\} \end{aligned} \tag{2.22}$$

$$\text{where} \qquad s(k_1, k_2) = \begin{cases} 1 & \text{if } k_1 N_2 + k_2 N_1 < N \\ -1 & \text{otherwise} \end{cases} \tag{2.23}$$

When N_1 is even, $C(\tilde{f}(N_1/2, k_2))$ and $C(\hat{f}(N_1/2, k_2))$ are functions of $\acute{T}(N_1/2, k_2)$ and $\acute{T}(N_1/2, N_2 - k_2)$. Similarly when N_2 is even, $C(\tilde{f}(k_1, N_2/2))$ and $C(\hat{f}(k_1, N_2/2))$ are functions of $\acute{T}(k_1, N_2/2)$ and $\acute{T}(N_1 - k_1, N_2/2)$. Notice that the value of $s(k_1, k_2)$ is essential only for the computation of $C(\hat{f}(k_1, k_2))$. For instance, for the example where $N = 20$ ($N_1 = 4$, $N_2 = 5$), $C(\tilde{f}(3, 2)) = C(7) = \acute{T}(3, 2) + \acute{T}(1, 3)$ and $C(\hat{f}(3, 2)) = C(17) = s(3, 2)\{\acute{T}(3, 2) - \acute{T}(1, 3)\} = \acute{T}(1, 3) - \acute{T}(3, 2)$.

The algorithm for DCT(N) can be summarized as follows.

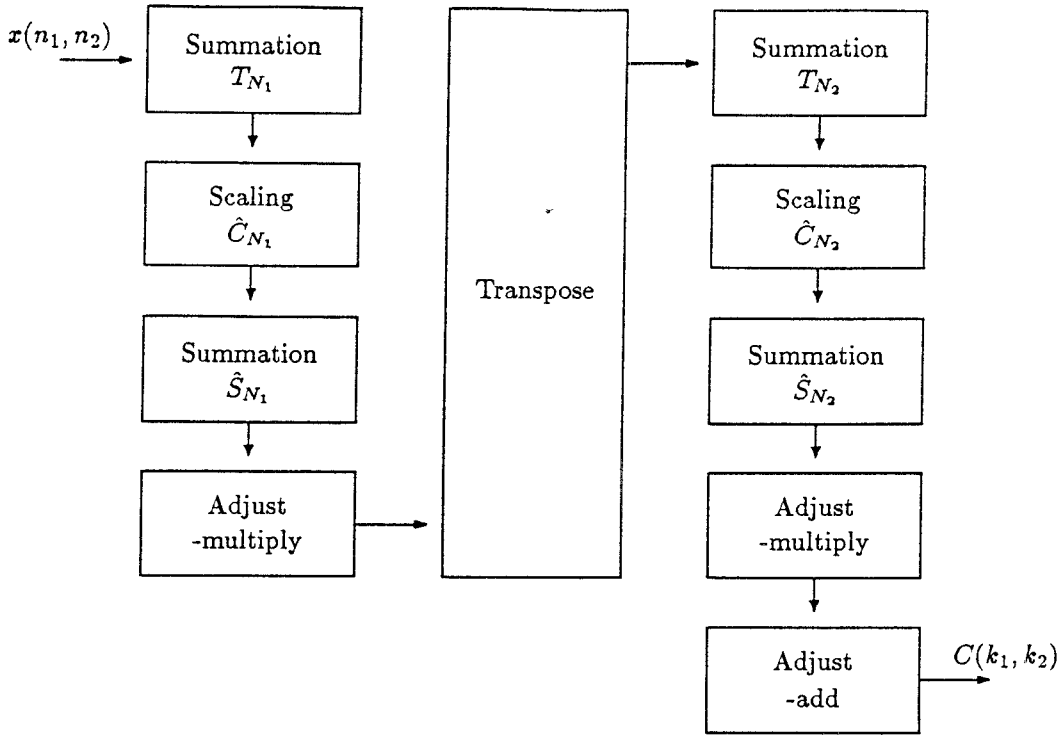


Figure 2.12: Block diagram for two-dimensional DCT over $(N_1 \times N_2)$ points, where N_1, N_2 are mutually prime

1. Compute DCT(N_1) over columns.

2. Compute DCT(N_2) over rows.

3. Compute C from $\acute{T}$ (equations 2.22,2.23).

Steps 1 and 2 can be mapped directly into a systolic architecture consisting of summation, scaling and adjust-multiply units (refer to Section 2.2.2). In order that the adjust-multiply unit of DCT(N_1) be systolic, $\hat{S}_{N_1}$ is embedded in the summation unit of DCT(N_1) (instead of S_{N_1}), where $\hat{S}_{N_1}$ is related to S_{N_1}

$$\begin{bmatrix} \acute{T}_{0,0} & \acute{T}_{0,1} & \acute{T}_{0,4} & \acute{T}_{0,2} & \acute{T}_{0,3} \\ \acute{T}_{1,0} & \acute{T}_{1,1} & \acute{T}_{1,4} & \acute{T}_{1,2} & \acute{T}_{1,3} \\ \acute{T}_{3,0} & \acute{T}_{3,1} & \acute{T}_{3,4} & \acute{T}_{3,2} & \acute{T}_{3,3} \\ \acute{T}_{2,0} & \acute{T}_{2,1} & \acute{T}_{2,4} & \acute{T}_{2,2} & \acute{T}_{2,3} \end{bmatrix}$$

Figure 2.13: Data array $\acute{T}$ after computing DCT́(4) over columns followed by DCT́(5) over rows during the computation of DCT over (4×5) points

by $\hat{S}_{N_1} = P_{N_1} S_{N_1}$ (refer to Section 2.2.2). Similarly, in order that the adjust-multiply unit of DCT́(N_2) be systolic, $\hat{S}_{N_2}$ is embedded in the summation unit of DCT́(N_2) (instead of S_{N_2}), where $\hat{S}_{N_2} = P_{N_2} S_{N_2}$. As in the case of DHT, a systolic transpose unit is needed to transpose the intermediate result obtained after computing DCT́(N_1) over columns. Figure 2.12 shows the interconnection of the various units for two-dimensional DCT over $(N_1 \times N_2)$ points, where N_1, N_2 are mutually prime.

By embedding matrices $\hat{S}_{N_1}$ and $\hat{S}_{N_2}$ in the summation units of DCT́(N_1) and DCT́(N_2) respectively, $\acute{T}(k_1, k_2)$, $\acute{T}(N_1 - k_1, k_2)$, $\acute{T}(k_1, N_2 - k_2)$ and $\acute{T}(N_1 - k_1, N_2 - k_2)$ are adjacent to each other at the end of Step 2 of the algorithm. Figure 2.13 illustrates the row and column permutations of data array $\acute{T}$ obtained after computing DCT́(4) over columns followed by DCT́(5) over rows in the example where $N = 20$ ($N_1 = 4$, $N_2 = 5$). The *adjust-add* unit in Figure 2.12 is a systolic unit which computes C from $\acute{T}$ (see equations 2.22, 2.23). Though this unit is very similar to the adjust-add unit of DHT, it needs an additional control input s corresponding to $s(k_1, k_2)$ in equation 2.23. We will describe it

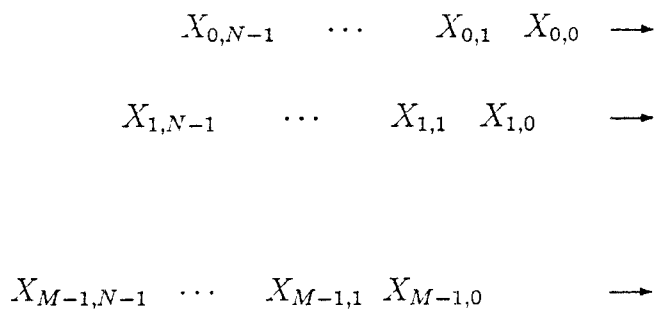


Figure 2.14: Skewed data array X of size $M \times N$

in the next section.

2.4 Bit-Serial Systolic Implementation

In this section we discuss the synchronous bit-serial systolic array implementations of the two-dimensional schemes for computing DHT and DCT. We choose to use this mode because the area of the adders, subtractors and multipliers is reduced by $\approx \frac{1}{b}$ of its word counterpart. Another advantage is that the communication within and between chips is more efficient. The disadvantage of decrease in throughput (since the bit-serial units have to be clocked b times for each time their word counterparts are clocked once) can be remedied by clocking at a faster rate [16]. This is possible because the bit-serial units have simpler logic compared to their word counterparts. In this section we give a brief description of the various units listed in Figures 2.6, 2.12.

The data flow through all the units is input-output consistent. The elements

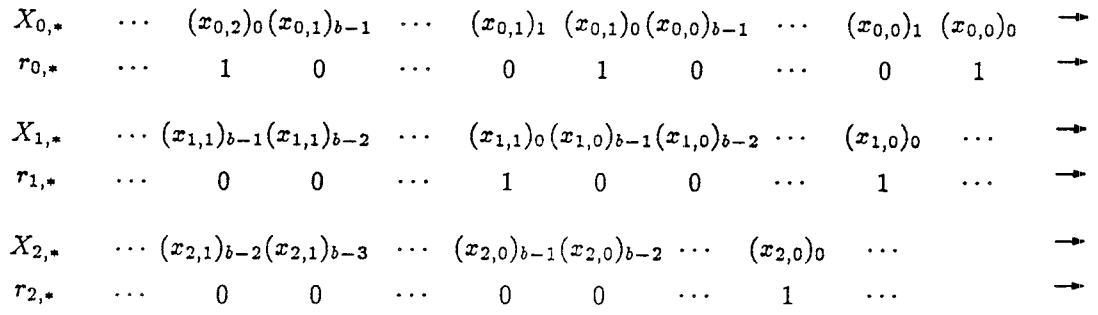


Figure 2.15: Bit level description of rows 0, 1, 2 of the data array

of the data array are fed in a skewed manner as shown in Figure 2.14. The hardware design is based on least significant bit (LSB) first binary arithmetic. Figure 2.15 describes the skewed data array at the bit level. $(x_{i,j})_k$ represents the kth LSB of the data element $x_{i,j}$. The control input r indicates that the first bit of the element is being supplied. Thus $r = 1$ for the least significant bit of the b-bit word and is 0 otherwise.

Summation Unit:

This unit performs the operation $Z = S_M X$, where S_M is an incidence matrix of size $(J \times M)$, X is the input matrix of size $(M \times N)$ and Z is the output matrix of size $(J \times N)$. Since the values of S_M are predefined, they can be embedded into this unit. Thus the summation unit can be constructed with three types of subunits, 1-bit adder ($s_{ij} = 1$), 1-bit subtractor ($s_{ij} = -1$), and 1-bit delay ($s_{ij} = 0$) [47]. The on-line delay in these subunits is 1. Consequently the data array is 1-bit skewed for efficient pipelining. The time delay between when the first bit of the input is supplied to the unit and when the first bit of the output

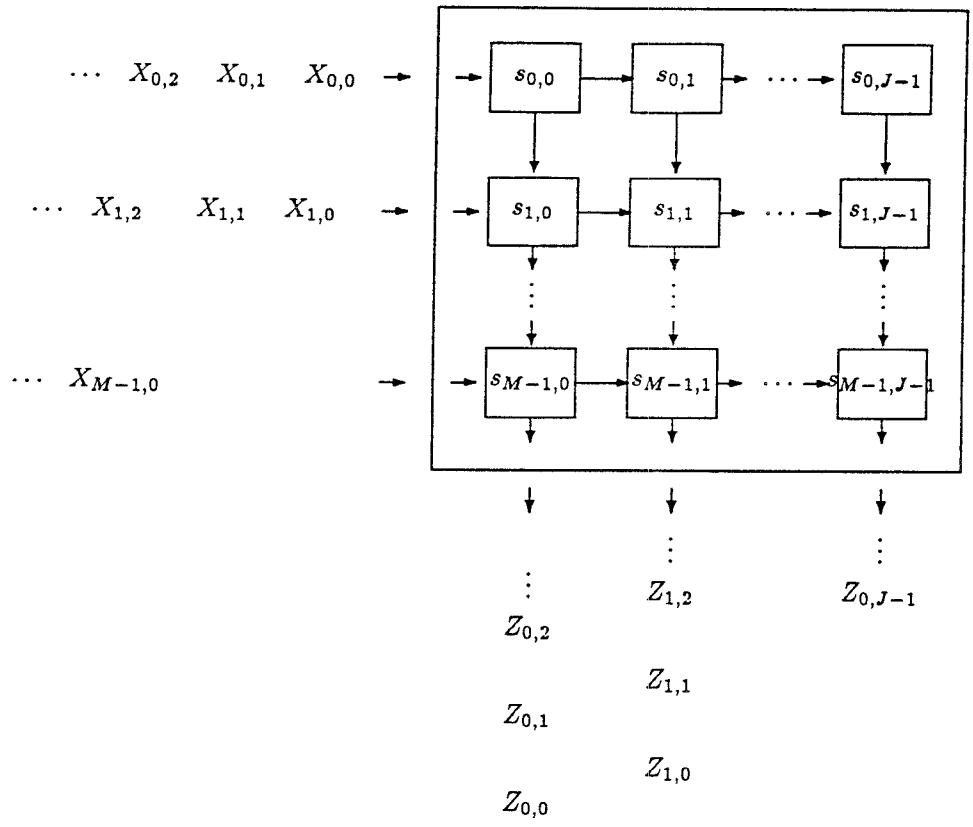


Figure 2.16: Data flow through the summation unit

is generated by the unit is M. Figure 2.16 shows the data flow through this unit.

Scaling Unit:

This unit performs the operation $Z = C_M X$, where C_M is a diagonal matrix of size $(M \times M)$, X and Z are the input and output matrices of size $(M \times N)$. Since the elements of C_M are known beforehand, they can be built into this unit [47]. Thus the jth subunit consists of a fixed multiplier corresponding to c_{jj}. The subunit is essentially a bit-serial multiplier [42]. The time delay between

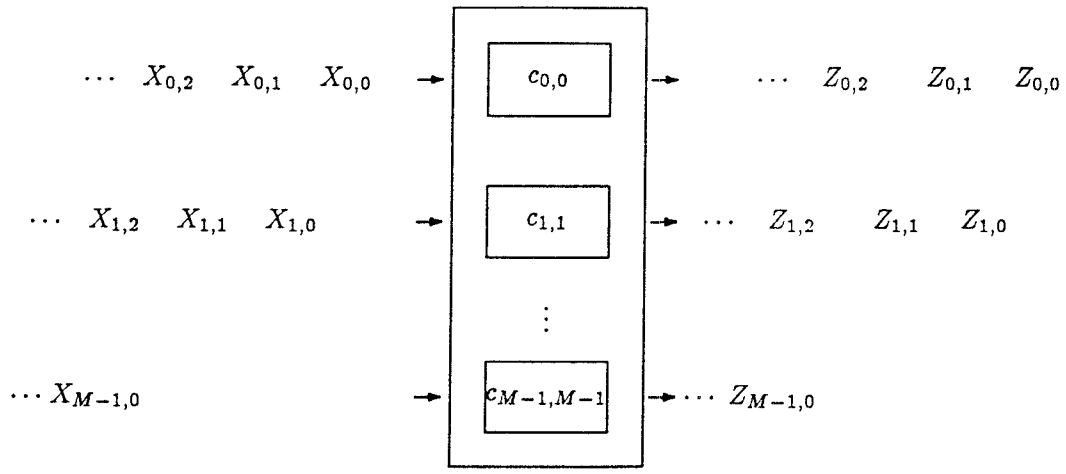


Figure 2.17: Data flow through the scaling unit

when the first bit of the input is supplied to the unit and when the first bit of the output is generated by the unit is b. Figure 2.17 shows the data flow through this unit.

Transpose Unit:

This unit performs the operation $Z = X^T$, where X and Z are the input and output matrices of sizes $(M \times N)$ and $(N \times M)$ respectively. The subunits are arranged in the form of a two-dimensional array of size $(q \times q)$, where $q = \max(M, N)$. Each subunit consists of a shift register of size $O(b)$ and control switches a and $\bar{a}$ to regulate the direction of flow of data. When a is on, the data in the ith block is read in and the data in the $(i-1)$th block is read out both along the east-west direction. When $\bar{a}$ is on, the data in the ith block is read out and the data in the $(i+1)$th block is read in both along the north-south direction. Thus alternate blocks are read in/read out along the east-west/north-

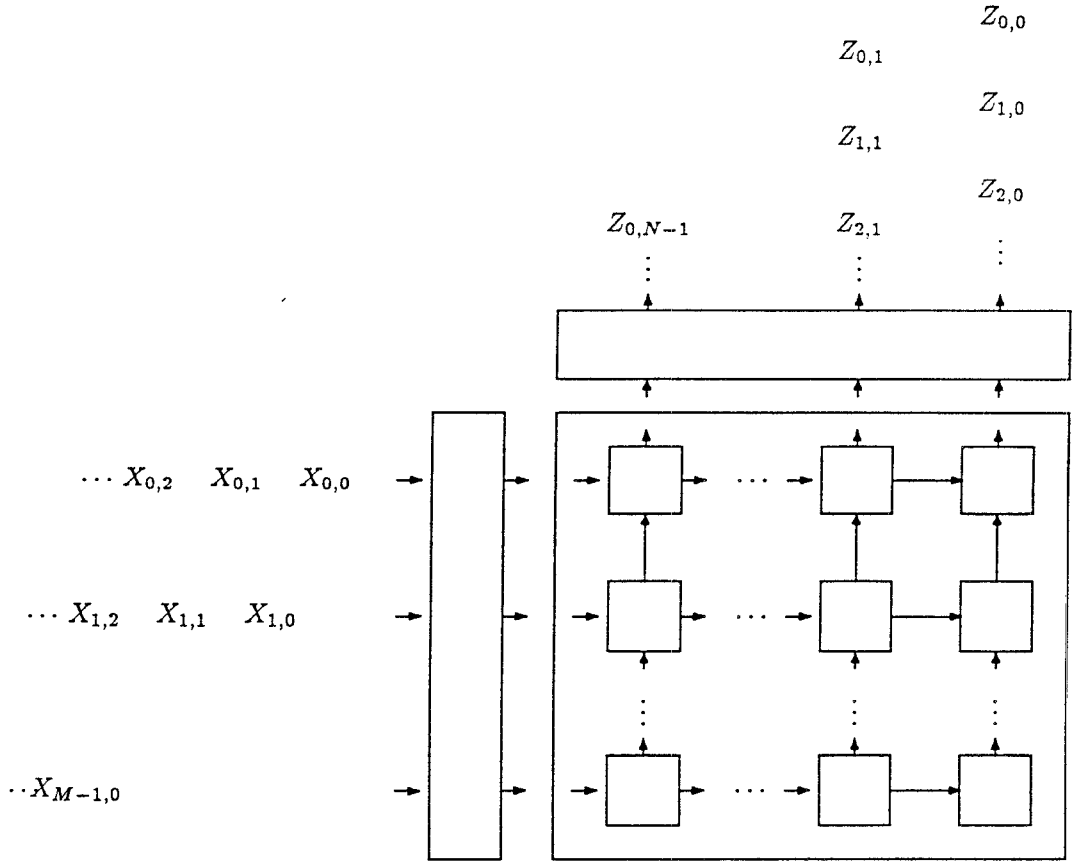


Figure 2.18: Data flow through the transpose unit

south directions. There are delays at the input and the output to adjust the bit skewness of the data. The time delay between when the first bit of the input is supplied to the unit and when the first bit of the output is generated by this unit is $b\max(M,N)+M+N-2$. Figure 2.18 shows the data flow through this unit.

Adjust-add unit of DHT:

This unit combines the elements of the data after computing DHT(N_1) over columns followed by DHT(N_2) over rows, to compute the DHT output (see

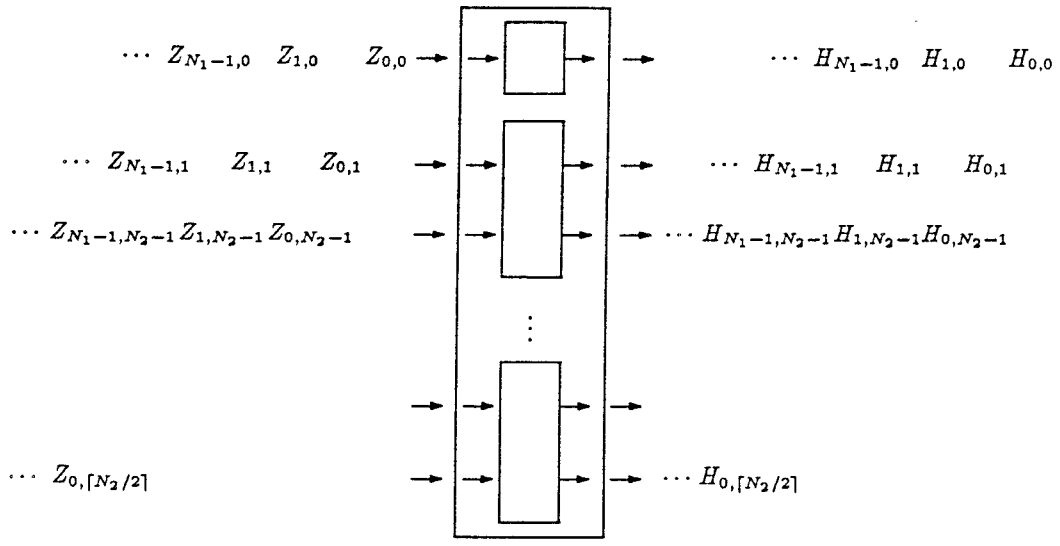


Figure 2.19: Data flow through the adjust-add unit of DHT

Figure 2.6). Equation 2.17 describes the relation between the input H and the output Z. Figure 2.19 describes the data flow through this unit.

The adjust-add unit consists of $(\lfloor N_2/2 \rfloor + 1)$ subunits. The $(\lceil k_2/2 \rceil)$th subunit operates on four adjacent inputs, $Z(k_1, k_2)$, $Z(N_1 - k_1, k_2)$, $Z(k_1, N_2 - k_2)$ and $Z(N_1 - k_1, N_2 - k_2)$ to compute four adjacent outputs, $H(k_1, k_2)$, $H(N_1 - k_1, k_2)$, $H(k_1, N_2 - k_2)$ and $H(N_1 - k_1, N_2 - k_2)$, $1 \leq k_1 \leq \lceil N_1/2 \rceil$, $1 \leq k_2 \leq \lceil N_2/2 \rceil$. The remaining subunits (corresponding to $k_2 = 0$ and $k_2 = N_2/2$) operate on one input $Z(k_1, k_2)$ to compute $H(k_1, k_2)$.

The $(\lceil k_2/2 \rceil)$th subunit, $1 \leq k_2 \leq \lceil N - 2/2 \rceil - 1$, consists of b-bit and $2b$-bit shift registers, 1-bit adder, 1-bit subtractor, two 1-bit delay units and some additional circuitry to compute $H(q, k_2)$, $q = \{0, N_1/2\}$. The time delay between when the first bit of the input is supplied to the unit and when the first bit of the output is generated by the unit is $(b + 2)$.

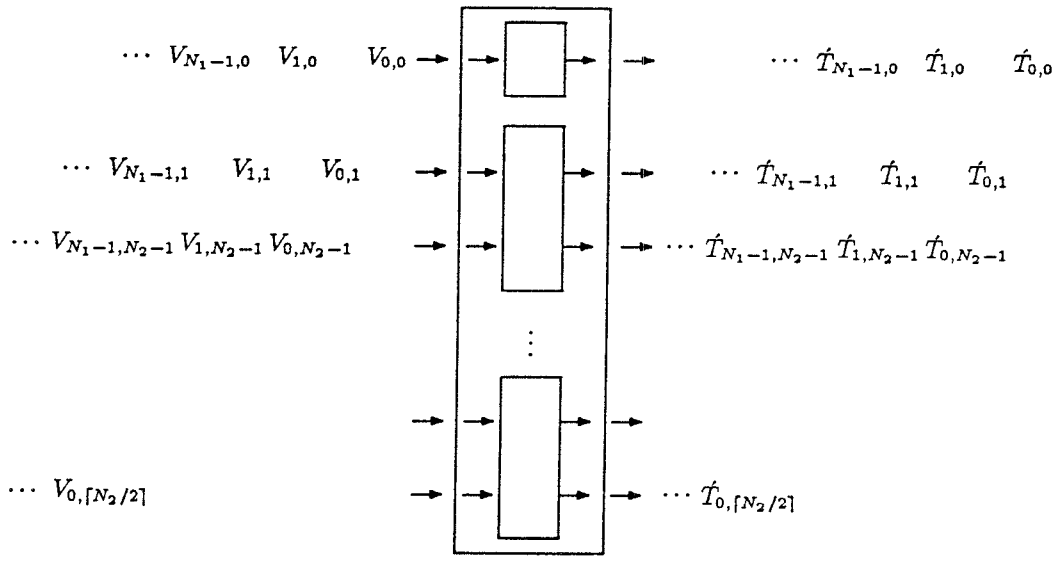


Figure 2.20: Data flow through the second adjust-multiply unit of DCT

Adjust-multiply unit of DCT:

This unit combines the elements of the data array after computing DHT over rows(columns) to compute DCT́ over rows(columns) (see equation 2.6). Here we describe the adjust-multiply unit of DCT́(N_2) (see Figure 2.12). The input is $V(k_1, k_2)$ (the output after computing DHT(N_2) over rows in Step 2) and the output is $\acute{T}(k_1, k_2)$. The adjust-multiply unit of DCT́(N_1) is very similar to this one. The data flow through this unit is illustrated in Figure 2.20.

The adjust-multiply unit consists of $(\lfloor N_2/2 \rfloor + 1)$ subunits. The $(\lceil k_2/2 \rceil)$th subunit operates on inputs $V(i, k_2)$ and $V(i. N_2 - k_2)$ to compute $\acute{T}(i, k_2)$ and $\acute{T}(i, N_2 - k_2)$, $0 \leq k_1 \leq N_1 - 1$, $1 \leq k_2 \leq \lceil N_2/2 \rceil - 1$. The remaining subunits (corresponding to $k_2 = 0$ and $k_2 = N_2/2$) operate on one input $V(i, k_2)$ to compute $T(i, k_2)$.

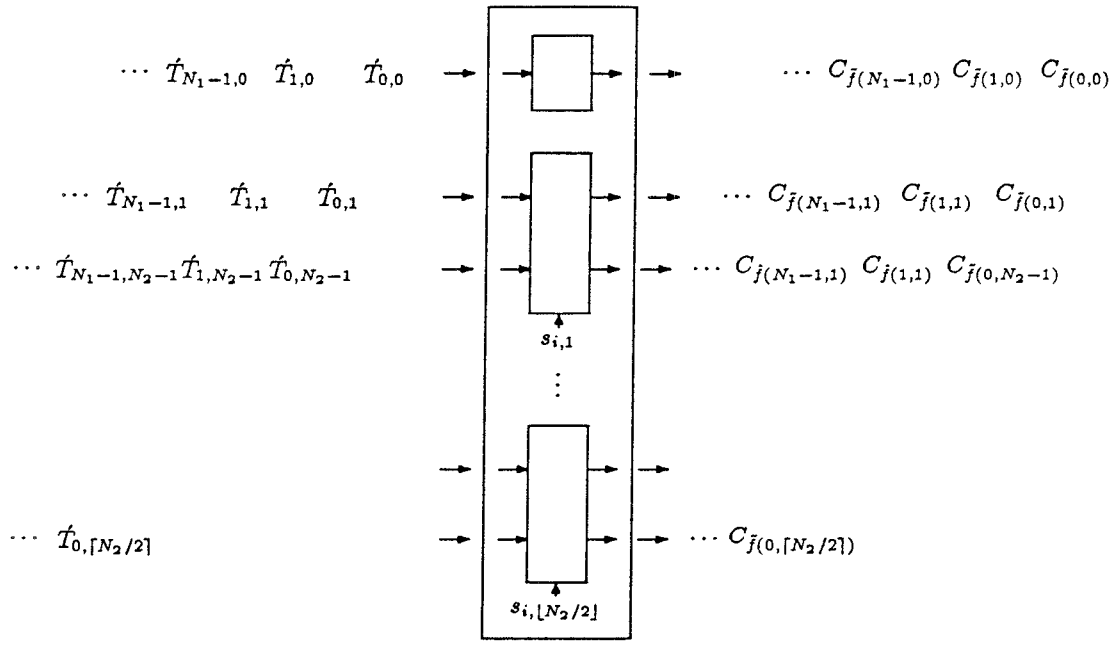


Figure 2.21: Data flow through the adjust-add unit of DCT

The $(\lceil k_2/2 \rceil)$th subunit, $1 \leq k_2 \leq \lceil N_2/2 \rceil - 1$, consists of two $cas(-\frac{k_2\pi}{2N_2})$ multipliers, two $cas(\frac{k_2\pi}{2N_2})$ multipliers, two 1-bit adders, two 1-bit delays and some additional circuitry to compute $\acute{T}(l, k_2)$, $l = \{0, N_1/2\}$. The time delay between when the first bit of the input is supplied to the unit and when the first bit of the output is generated by the unit is $(b + 2)$.

Adjust-add unit of DCT:

This unit combines the elements of the data array obtained after computing $\text{DC}\acute{\text{T}}(N_1)$ over columns followed by computing $\text{DC}\acute{\text{T}}(N_2)$ over rows, to compute the DCT output. Equation 2.22 describes the relation between the input $\acute{T}$ and the output C. Figure 2.21 illustrates the data flow through this unit.

The adjust-add unit consists of a total of $(\lfloor N_2/2 \rfloor + 1)$ subunits. The first subunit operates on $\acute{T}(k_1, 0)$ to compute $C(\check{f}(k_1, 0))$. The next $(\lceil N_2/2 \rceil - 1)$ subunits (corresponding to $k_2 = 1$ through $k_2 = \lceil N_2/2 \rceil - 1$) operate on four adjacent inputs $\acute{T}(k_1, k_2)$, $\acute{T}(N_1 - k_1, k_2)$, $\acute{T}(k_1, N_2 - k_2)$ and $\acute{T}(N_1 - k_1, N_2 - k_2)$ to compute four adjacent outputs, $C(\check{f}(k_1, k_2))$, $C(\hat{f}(k_1, k_2))$, $C(\tilde{f}(N_1 - k_1, k_2))$ and $C(\hat{f}(N_1 - k_1, k_2))$. If N_2 is even, the last subunit operates on $\acute{T}(k_1, N_2/2)$ and $\acute{T}(N_1 - k_1, N_2/2)$ to compute $C(\tilde{f}(k_1, N_2/2))$ and $C(\check{f}(N_1 - k_1, N_2/2))$.

The control input s takes up the value of $s(k_1, k_2)$ (see equation 2.23). It is essential for the computation of $C(\hat{f}(k_1, k_2))$. Since these values can be determined apriori, they are stored in N_1-bit shift registers and fed to the subtractor in the subunit. Note that the only values of $s(k_1, k_2)$ that are required are the values in the range $1 \leq k_2 \leq \lfloor N_2/2 \rfloor$.

The $(\lceil k_2/2 \rceil)$th subunit, $1 \leq k_2 \leq \lceil N_2/2 \rceil - 1$, consists of b-bit and $2b$-bit shift registers, 1-bit adder, two 1-bit subtractors, two 1-bit delays and some additional circuitry for the computation of $C(\check{f}(0, k_2))$ and $C(\tilde{f}(0, N_2 - k_2))$ and if N_1 is even, the computation of $C(\tilde{f}(N_1/2, k_2))$ and $C(\hat{f}(N_1/2, k_2))$. The time delay between when the first bit of the input is supplied to the unit and when the first bit of the output is generated by the unit is $(b + 3)$.

2.5 Conclusion

In this chapter we have presented bit-serial systolic array implementations for computing DHT and DCT when the transform size N is factorizable into mutually prime factors, N_1 and N_2. We mapped the two-dimensional formulations for DHT [3] and DCT [39] into two-dimensional systolic arrays after modifying them appropriately. The total number of systolic steps in the architectures for both DHT and DCT is $O(b(N_1 + N_2))$. This is clearly optimal since feeding the data in bit-serially will require the same order of steps.

The building blocks for the proposed architectures consist of summation, scaling, transpose, adjust-add and adjust-multiply units. The total number of adders/subtractors is approximately $(4N + N_2)$ for DHT and $(4N + 2N_2 + N_1)$ for DCT. The total number of multipliers is approximately $(N_1 + N_2)$ for DHT and $3(N_1 + N_2)$ for DCT. The total number of shift registers is approximately $(N + N_2)$ for both DHT and DCT. The total number of input pads is $(N_1 + c)$, where c is a constant (4 or 5).

The algorithm that we developed here can be extended to the case when N is factorizable into any number of relatively prime factors. Let $N = N_1 N_2 \ldots N_d$, such that the N_is are relatively prime, $1 \leq i \leq d$. Then the transform over N points can be mapped into a d-dimensional transform over $(N_1 \times N_2 \times \ldots \times N_d)$ points, by appropriate choice of input and output index mappings. The algorithm then consists of computing the transform over N_1 points along one dimension, followed by computing the transform over N_2 points along another

dimension, and so on for all d dimensions. The corresponding architecture consists of 1-D transform computation units and permutation units, which permute the data appropriately for computation over the next dimension. Though the mapping of such an algorithm into VLSI is easy for the case when $d = 2$, it is not so for arbitrary d. We address this problem in the next chapter.

Chapter 3

Multidimensional Transforms

3.1 Introduction

In this chapter we describe a family of optimal VLSI architectures for computing d-dimensional linear separable transforms (or equivalently d-dimensional DXT). The optimality criteria is based on VLSI complexity theory [54]. An architecture for computing d-dimensional DXT over $n = N^d$ points is said to be *optimal* if its $AT^2 = O(n^2 \log^2 M)$, where A is the area, T is the computation time and all computations are over the ring of integers Z_M, $M = N + 1$. Our architecture consists of one-dimensional DXT(N) computation units which compute DXT(N) over one index, and permutation units which order data so that in the next iteration DXT(N) can be computed over the next index. The architecture has an area $A = O(N^{d+2a})$ and computation time $T = O(dN^{\frac{d}{2}-a}b)$ for all a in the range $\frac{1}{2}\log_N b \leq a \leq \frac{d}{2}$, where $b = O(\log M)$ is the precision. All architectures

in this range of a achieve the AT^2 bound of $O(n^2b^2)$ for constant d. When $a = \frac{d}{2}$ our architecture has the same input data organization as that of [24]. For this case the computation time of our architecture is $O(\log M)$ compared to $O(\log^2 n)$ of [24]. The architecture of [4] is optimal for the same range of area and computation time as our architecture. However our architecture has the added advantage that it is simple, regular and hence suitable for VLSI implementation.

The rest of the chapter is organized as follows. In Section 3.2 we state the definitions and assumptions for the computation of d-dimensional DXT. We then briefly review the VLSI model of computation. In Section 3.3 we first describe an architecture when the input is in a single file and then develop a family of architectures with area-time trade-offs. We make some concluding remarks in Section 3.4.

3.2 Preliminaries

3.2.1 Definitions

Let n be the total number of data elements that are to be organized in a d-dimensional data cube. Then each element can be represented by d indexes $n_1, n_2, \ldots, n_d$. A d-dimensional linear separable transform is defined by

$$X(k_1, k_2, \ldots, k_d) = \sum_{n_d} \cdots \sum_{n_2} \sum_{n_1} x(n_1, n_2, \ldots, n_d)\alpha_1(n_1, k_1)\alpha_2(n_2, k_2) \cdots \alpha_d(n_d, k_d),$$

where the α_i's are the transform functions, $0 \leq k_i, n_i \leq N_i - 1$ for $1 \leq i \leq d$ and $n = N_1 N_2 \ldots N_d$. For instance, $\alpha_i(n_i, k_i) = exp(-j\frac{2\pi}{N_i} n_i k_i)$ for

d-dimensional DFT, $\alpha_i(n_i, k_i) = cos\,[\frac{2\pi}{N_i} n_i k_i] + sin\,[\frac{2\pi}{N_i} n_i k_i]$ for d-dimensional DHT and $\alpha_i(n_i, k_i) = cos\,[\frac{\pi}{2N_i}(2n_i + 1)k_i]$ for d-dimensional DCT. A straightforward way of computing $X(k_1, k_2, \ldots, k_d)$ consists of computing one-dimensional transform over each of the d dimensions. We use this approach to compute d-dimensional DXT.

In order to simplify our analysis we assume

- $N_1 = N_2 = \cdots = N_d = N$. Thus $n = N^d$.
- All computations involve fixed point arithmetic with b bits of precision, where $b = \Omega(\log M)$.

3.2.2 Model of computation

In this section we briefly review the VLSI model of computation [6, 54]. A VLSI circuit is a computation graph $G = (V, E)$ whose vertices V are *nodes* and whose edges E are wires. A node is a localized set of switching elements which perform a simple logical function. A wire carries signals from the output of one node to the input of another. The unit of area in this model is determined by the 'minimum feature width' of the processing technology. The unit of time is equal to the system clock for synchronous circuits. This model is characterized by a set of rules concerning layout, timing, etc. We list some of the rules here.

Wires are one unit wide. At most two wires may cross at any point in the plane. A node occupies $O(1)$ area. A node has at most $O(1)$ input and $O(1)$ output wires. Wires cannot cross over nodes. The total area A of a collection of

nodes and wires is the number of unit squares in the smallest enclosing rectangle.

Wires have unit bandwidth. Nodes have $O(1)$ delay. The propagation time through the wires is $O(1)$ irrespective of the length of the wire [6] (synchronous model). (There are other models where the propagation time through a wire of length k is $O(\log k)$ [54] (capacitive model) or even $O(k)$.) The computation time is the number of units between the appearance of the first input bit on some port and the appearance of the last output bit on some port.

The I/O assumptions are as follows. Each input is received only once and at exactly one input port. Each I/O variable is available in a prespecified sequence at a prespecified port for all instances of the problem.

AT^2 bounds:

The two main aspects of VLSI computation theory are proving lower bounds for a particular problem and constructing upper bounds. An *optimal* design is one in which the upper bound matches the lower bound. The lower bounds of area and computation time are usually based on information storage or information flow concepts. The performance metric is of the form $(area) * (time)^{2\alpha}$, where $0 \leq \alpha \leq 1$. Such lower bounds have been established for a large number of problems. The upper bounds are VLSI designs for which the area and the computation time can be determined by the VLSI model of computation. Once the lower bound is known, the designer tries to develop an algorithm and the corresponding layout such that the $area * time^{2\alpha}$ limits of the design are as close to the theoretical limit as possible.

Thompson [54] proposed a scheme for obtaining the area-time lower bounds of a problem based on the *information exchange* I, where I is defined as the minimum number of bits that two processors must exchange in order to solve a problem. It is assumed that half of the input variables are available to each processor at the beginning of the computation. The area-time complexities of a problem with information exchange I satisfies the bound $AT^2 = \Omega(I^2)$ [54]. Bilardi, Hornick and Sarrafzadeh [4] established the AT^2 lower bounds for d-dimensional DFT.

Theorem [4] : The information exchange I of an $(N \times N \times \ldots \times N)$-point d-dimensional DFT on the finite ring Z_M satisfies the relation $I \geq \frac{n-1}{4} \log M$, where M is a prime and $M = N + 1$.

Thus the AT^2 lower bound for d-dimensional DFT on Z_M is $\Omega(n^2 \log^2 M)$. Since d-dimensional DFT can be obtained from d-dimensional DHT and DCT and vice versa without any increase in the area and time complexities, one can show that the lower bound on AT^2 for d-dimensional DHT and DCT is also $\Omega(n^2 \log^2 M)$.

3.2.3 Related work

The two main schemes for computing $(N \times N \times \ldots \times N)$ d-dimensional transforms consist of either computing a sequence of one-dimensional transforms each of size N or computing a matrix-vector product. In the first scheme the number of multiplications is $M = dN^{d-1}M_1$, where M_1 denotes the number of multiplications

required for a one-dimensional transform. For instance for DFT, $M_1 = N \log N$ and $M = n \log n$, where $n = N^d$. The architectures of [9, 24] are based on this scheme. The straightforward implementation of the second scheme, (which may not be applicable to all multi-dimensional transforms), consists of $M = n^3$ multiplications. The architecture of [4] for computing d-dimensional DFT is based on this scheme.

The existing architectures for computing two-dimensional DFT are based on computing one-dimensional DFT on each of the two indexes, that is, if the input data is in a one-dimensional array, then one-dimensional DFT is computed on columns and then on rows of the data array. All these architectures [10, 62] have an AT^2 of $O(n^2 \log^3 n)$, where $n = N^2$. Gertner and Shamash [24] generalized this approach and proposed an architecture for d-dimensional DFT which consists of N^{d-1} butterfly arrays for computing one-dimensional DFT(N) and a rotation network array for permuting the data. The area of their design is $O(n^2)$ and the time complexity is $O(d \log^2 n)$, resulting in an AT^2 of $O(d^2 n^2 \log^4 n)$. Recently Bilardi, Hornick and Sarrafzadeh [4] developed a family of optimal architectures with $AT^2 = O(n^2 b^2)$ where b is the precision, $b = O(\log M)$. They showed that for the case when $N = PQ$, a $(N \times N \times \ldots \times N)$ d-dimensional DFT can be expressed as a $(P \times P \times \ldots \times P)$ d-dimensional DFT followed by a $(Q \times Q \times \ldots \times Q)$ d-dimensional DFT. They mapped this algorithm into an architecture consisting of transposers to order the data, a unit consisting of Q mesh of trees network (each of which computes a $(P \times P \times \ldots \times P)$ d-dimensional DFT), and a unit consisting of P mesh of trees network (each of which computes a $(Q \times Q \times \ldots \times Q)$ d-dimensional DFT).

3.3 Architectures for $d \geq 2$

In this section we first describe an optimal architecture for d-dimensional DXT when the input data is in a single file and then develop a family of optimal architectures with area-time trade-offs.

3.3.1 Input is in a single file

Let the input data of size $n = N^d$ be organized in a single file in the order of its indexes. Any linear separable transform DXT(n) can be computed in the following way [24].

1. Repeat steps 2 and 3 d times
2. Perform one-dimensional DXT(N) on each of the N^{d-1} groups
3. Rotate over an index

The analysis of the above algorithm is as follows. In Step 2, one-dimensional DXT(N) is computed over index n_i for each of the N^{d-1} groups of size N. In Step 3 the data is rotated over an index so that one-dimensional DXT(N) can be computed over index n_{i+1} in the next iteration. The above process is repeated d times, once for each index.

This algorithm can be mapped into an architecture consisting of an input multiplexer, MUX, a $DXT(n)$ computation unit which computes DXT(N) on each of the N^{d-1} groups, a rotator unit $R(n)$ which permutes data of size n over

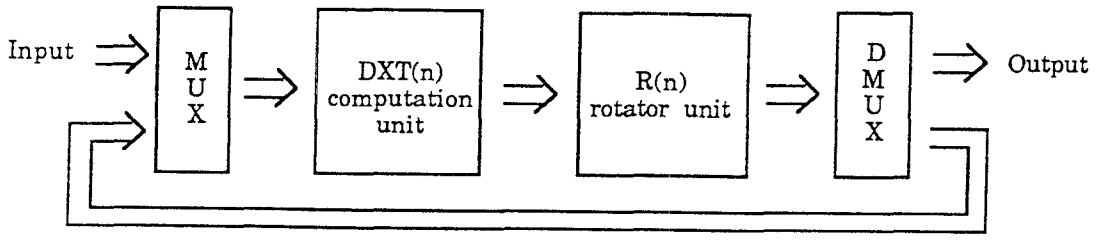


Figure 3.1: Architecture for d-dimensional DXT(n) when the input is in a single file

d indexes, and an output demultiplexer, DMUX. The data is circulated through these units d times, once for each index. Figure 3.1 illustrates the corresponding architecture. We describe the details of each of these units in the rest of this subsection.

$DXT(n)$ computation unit :

The $DXT(n)$ computation unit consists of N^{d-1} subunits, each of which computes DXT(N) on a group of size N. We use the *mesh of trees* (also referred to as orthogonal trees) network [40, 45] to compute DXT(N). This is because DXT(N) is a linear transformation and hence can be easily mapped into this architecture. Moreover this architecture achieves the optimal computation time $T = O(\log N)$. We first present the $(N \times N)$ mesh of trees network with $AT^2 = O(N^2 \log^4 N)$ and then show how to compute one-dimensional DXT(N) optimally with $AT^2 = O(N^2 \log^2 N)$.

The $(N \times N)$ mesh of trees network consists of N^2 processors arranged in the form of an $(N \times N)$ array such that each row and each column of processors

forms the leaves of a binary tree. Thus processors $P[i,j]$, $0 \leq i,j \leq N-1$, are connected by N binary trees along the columns, referred to as $CT(0)$, $CT(1)$, ..., $CT(N-1)$, and by N binary trees along the rows, referred to as $RT(0)$, $RT(1)$, ..., $RT(N-1)$. The leaves of $CT(j)$ are the processors $P[0,j]$, $P[1,j]$, ..., $P[N-1,j]$ and the leaves of $RT(i)$ are the processors $P[i,0]$, $P[i,1]$, ..., $P[i,N-1]$. Figure 3.2 describes the layout of a (4×4) mesh of trees network. The processors consist of bit-serial multipliers which can be laid out in area $O(b)$. The nodes of the row and column trees not only communicate between processors but also carry out simple operations like add, compare. Each node is of area $O(1)$. Without loss of generality the roots of the column trees are used as input ports and the roots of the row trees are used as output ports. The $(N \times N)$ mesh of trees network can be laid out in area $O(N^2 \log^2 N)$ [40, 45].

We next show how DXT(N) can be computed by an $(N \times N)$ mesh of trees network. Since DXT(N) is a linear transformation, it can be expressed as BX, where B is an $(N \times N)$ matrix and X is an $(N \times 1)$ column vector. $B(i,j)$ is embedded in the multiplier of processor $P[i,j]$. The procedure is as follows.

1. Input x_j at the root of $CT(j)$ and broadcast to all the leaves using the internal nodes of $CT(j)$.

2. Multiply x_j by $B(i,j)$ in processor $P[i,j]$.

3. Add $x_j * B(i,j)$ using the internal nodes of $RT(i)$ and output the result X_i at the root.

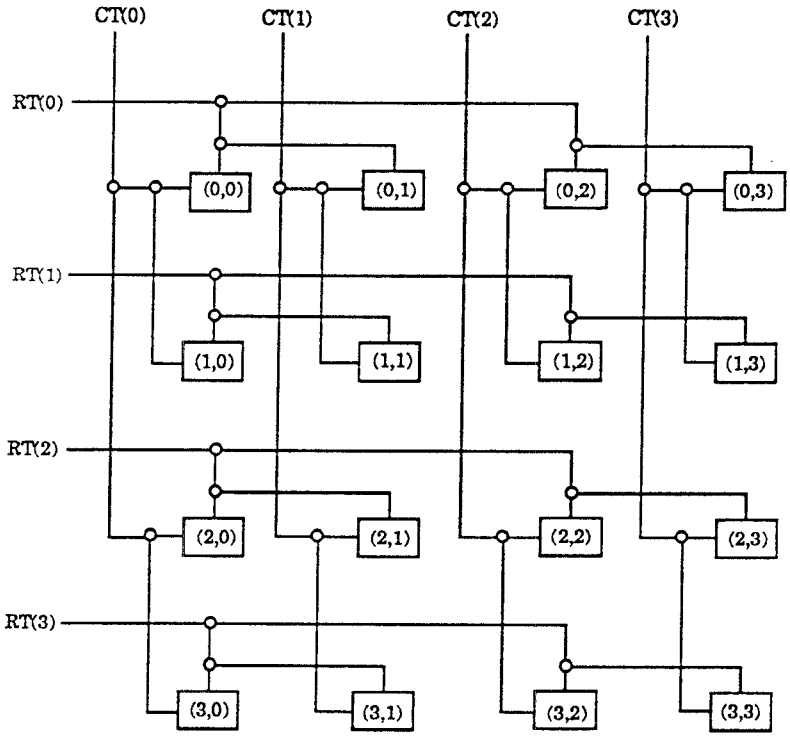


Figure 3.2: (4×4) mesh of trees network

In the pipelined bit-serial mode with $b = O(\log N)$, the time required to compute DXT(N) is $O(\log N)$. Thus AT^2 for this design is $O(N^2 \log^4 N)$, which is $O(\log^2 N)$ away from the optimal.

Since the lower bound of the computation time is $\Omega(\log N)$, in order that AT^2 be optimal, the area has to be reduced to $O(N^2)$. We now describe an optimal architecture for DFT(N) [5]. This architecture can be used to compute other linear transforms like DHT(N) and DCT(N) optimally. The one-dimensional transform over N points is first mapped into a two-dimensional transform over $(N_1 \times N_2)$ points, where $N = N_1 N_2$ and N_1, N_2 are not necessarily prime to each other. By choosing the input index mapping $g_1(n) = (n_1, n_2)$ and the output

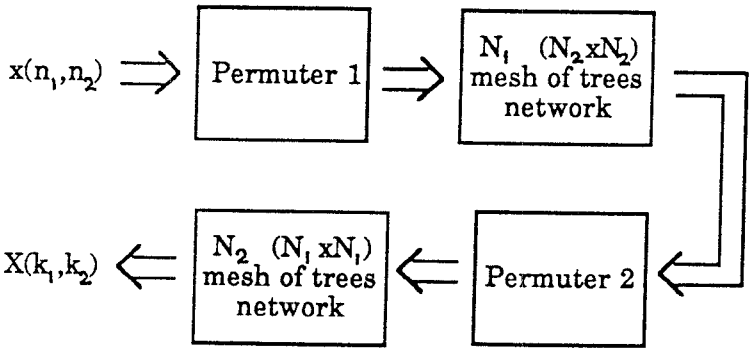


Figure 3.3: DXT(N) computation unit with $AT^2 = O(N^2 \log^2 N)$

index mapping $g_2(k) = (k_1, k_2)$ appropriately, $X(k)$ can be mapped to $\hat{X}(k_1, k_2)$, where

$$\hat{X}(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \hat{x}(n_1, n_2)\hat{a}(k_1, k_2, n_1, n_2),$$

$0 \leq k_1 \leq N_1 - 1$, $0 \leq k_2 \leq N_2 - 1$. If $\hat{a}(k_1, k_2, n_1, n_2)$ can be expressed as the product of $C_2(n_1, n_2, k_2)$ and $C_1(n_1, k_1, k_2)$, then two sets of mesh of trees network can be used to compute $\hat{X}(k_1, k_2)$. The first set of mesh of trees networks of size $(N_2 \times N_2)$ computes $Y(n_1, k_2) = \sum_{n_2} \hat{x}(n_1, n_2)C_2(n_1, n_2, k_2)$ and the second set of mesh of trees networks of size $(N_1 \times N_1)$ computes $\hat{X}(k_1, k_2)$ $= \sum_{n_1} Y(n_1, k_2)C_1(n_1, k_1, k_2)$. Such a decomposition is possible for one-dimensional DFT, DHT and DCT (see Chapter 2). Figure 3.3 illustrates the architecture for computing DXT(N). We would like to mention here that the scheme of [4] is a generalization of this decomposition scheme in d-dimensional space. In that scheme a d-dimensional DFT over $(\underbrace{N \times N \times \ldots \times N}_{d \text{ times}})$ points is mapped into a

$2d$-dimensional DFT over $(\underbrace{N_1 \times N_1 \times \ldots \times N_1}_{d \text{ times}} \times \underbrace{N_2 \times N_2 \times \ldots \times N_2}_{d \text{ times}})$.

The first stage of the DXT(N) computation unit (see Figure 3.3) consists of a permutation unit, called Permuter 1, which routes the data so that elements with the same value of n_1 are adjacent to each other. The second stage consists of N_1 $(N_2 \times N_2)$ mesh of trees network such that the n_1th network computes $Y(k_1, n_2)$, $0 \leq n_1 \leq N_1 - 1$. The third stage consists of another permutation unit, called Permuter 2, which routes the data so that the elements with the same value of n_2 are adjacent to each other. The fourth stage consists of N_2 $(N_1 \times N_1)$ mesh of trees network such that the k_2th network computes $\hat{X}(k_1, k_2)$, $0 \leq k_2 \leq N_2 - 1$.

This design has an area of $O(N^2)$, computation time of $O(\log N)$ and AT^2 of $O(N^2 \log^2 N)$ for $N_1 \in [\Omega(\log^2 N), O(N/\log^2 N)]$.

Rotator unit $R(n)$:

The rotator unit is a network which permutes data by rotation over an index in one step [24]. Thus data x represented by $(n_1, n_2, \ldots, n_d)$ is rotated to place $(n_2, n_3, \ldots, n_d, n_1)$, which is the cyclic left shift of the index representation of data x. Notice the similarity of the rotator with the perfect shuffle network, which shuffles data such that data x is shuffled to place $\hat{x}$, where $\hat{x}$ is obtained by cyclic left shift of the binary representation of x.

Let a *necklace* be defined as a collection of nodes which are formed on rotation of the indexes. For instance for $N = 3$ and $d = 4$, the necklace generated by (2120) is (2120) – (1202) – (2021) – (0212). The rotator unit permutes data with

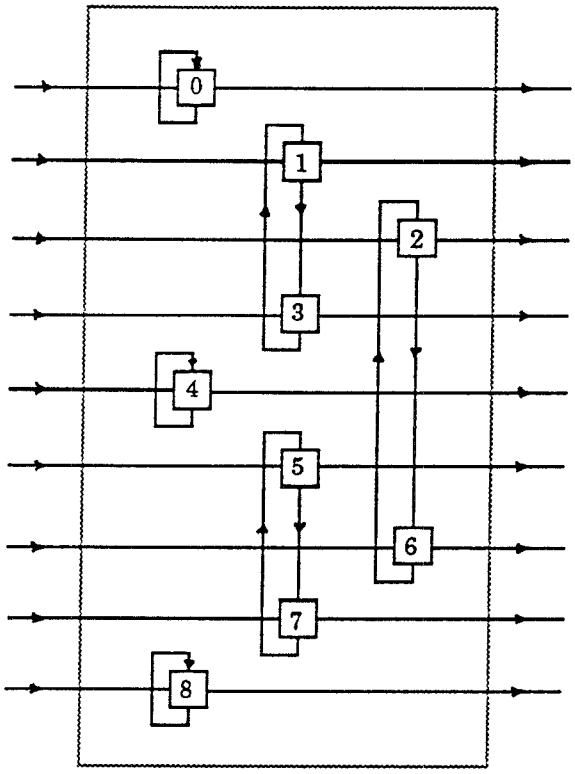


Figure 3.4: Layout of a rotator unit for $N = 3$, $d = 2$

the help of such necklaces. Of these necklaces, a few are *cyclic*, which means that the rotation of the indexes of such a necklace causes the value of the rotated necklace to cycle. The majority of the necklaces are however *non-cyclic*. While a cyclic necklace can be laid out in 2 columns, a non-cyclic necklace requires at most $O(d)$ columns.

The design of the rotator unit for any N (that is, N not necessarily a power of 2) is as follows. The area of the rotator unit is a function of the number of non-cyclic necklaces. The total number of necklaces is equivalent to the number of circular permutations of d objects of N kinds, with repetitions allowed. This is $\frac{1}{d}\sum_{q|d}\phi(q)N^{\frac{d}{q}}$ [50], which is equal to $O(\frac{N^d}{d})$, where q is a divisor of d and

$\phi(q)$ is Euler's function. This implies that the number of non-cyclic necklaces is at most $O(\frac{N^d}{d})$. Since $O(d)$ columns are required to layout each non-cyclic necklace, the total number of columns is at most $O(N^d)$ and the area of the rotator unit is at most $O(N^{2d}) = O(n^2)$. As in the design of [24] the data enter the rotator in n rows. It is then permuted with the help of load-shift cells placed at the connection of a row input and its column necklace. The time taken to permute the data over an index is $O(b)$. Figure 3.4 shows the layout of a rotator unit for $N = 3$ and $d = 2$.

Area and time complexities:

The input MUX and the output DMUX can be laid out in $O(n) \times O(1)$ area. The area of the DXT(n) computation unit is $O(N^{d+1})$ and that of the rotator unit is $O(n^2)$. Thus $A_{max} = O(n^2)$. The time taken to compute DXT(N) as well as to rotate data over an index is $O(b)$. Thus the total computation time is $O(db)$, which can be approximated to $O(b)$ for fixed d. The resulting AT^2 of the design is $O(n^2b^2)$.

3.3.2 Area-time trade-offs

We describe next how the input data can be pipelined to give a family of architectures satisfying $AT^2 = O(n^2b^2)$ with area-time trade-offs. The initial input data configuration is obtained by organizing the input data in column major order form in a two-dimensional array I of size $N^{\frac{d}{2}+i} \times N^{\frac{d}{2}-i}$, $0 \leq i \leq \frac{d}{2}$. If f is a fraction such that N^f is an integer, then this two-dimensional array can be par-

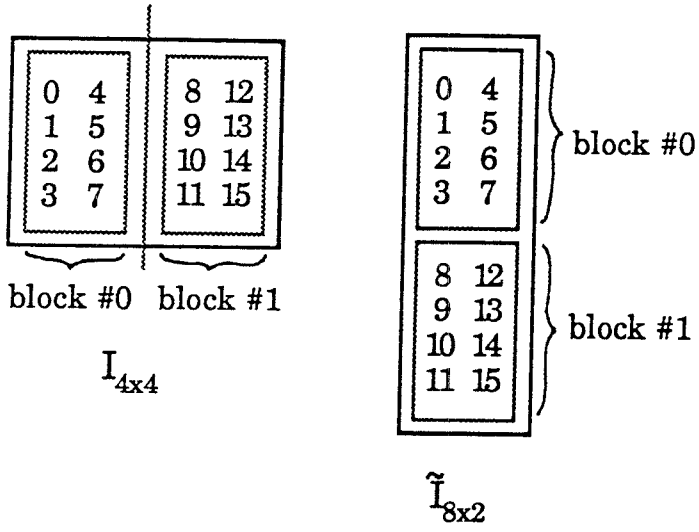


Figure 3.5: Input data configuration ($N = 4, d = 2, i = 0, f = \frac{1}{2}$)

titioned vertically into N^f *blocks*, each of size $N^{\frac{d}{2}+i} \times N^{\frac{d}{2}-i-f}$, and placed one on top of each other. The resulting input data array $\tilde{I}$ is of size $N^{\frac{d}{2}+a} \times N^{\frac{d}{2}-a}$, where $a = i + f$ and $0 < a \leq \frac{d}{2}$. Figure 3.5 illustrates the input data configuration in an example with $N = 4$, $d = 2$, $i = 0$ and $f = \frac{1}{2}$.

The scheme for computing d-dimensional DXT consists of computing $(\frac{d}{2} + i)$-dimensional DXT followed by computing $(\frac{d}{2} - i)$-dimensional DXT on the columns of $\tilde{I}$. Computation of $(\frac{d}{2}+i)$-dimensional DXT on columns of size $N^{\frac{d}{2}+a}$ is straightforward. Since $a \geq i$, a set of N^f DXT($N^{\frac{d}{2}+i}$) computation units can be used to compute on columns of size $N^{\frac{d}{2}+i}$ in the N^f vertically stacked blocks in $\hat{I}$. In order to compute $(\frac{d}{2} - i)$-dimensional DXT, the two-dimensional data array is permuted such that all the elements which were row-adjacent in I are made column-adjacent. This is done by first partitioning the two-dimensional data array $\tilde{I}$, horizontally into N^{2a} *subblocks* and then transposing

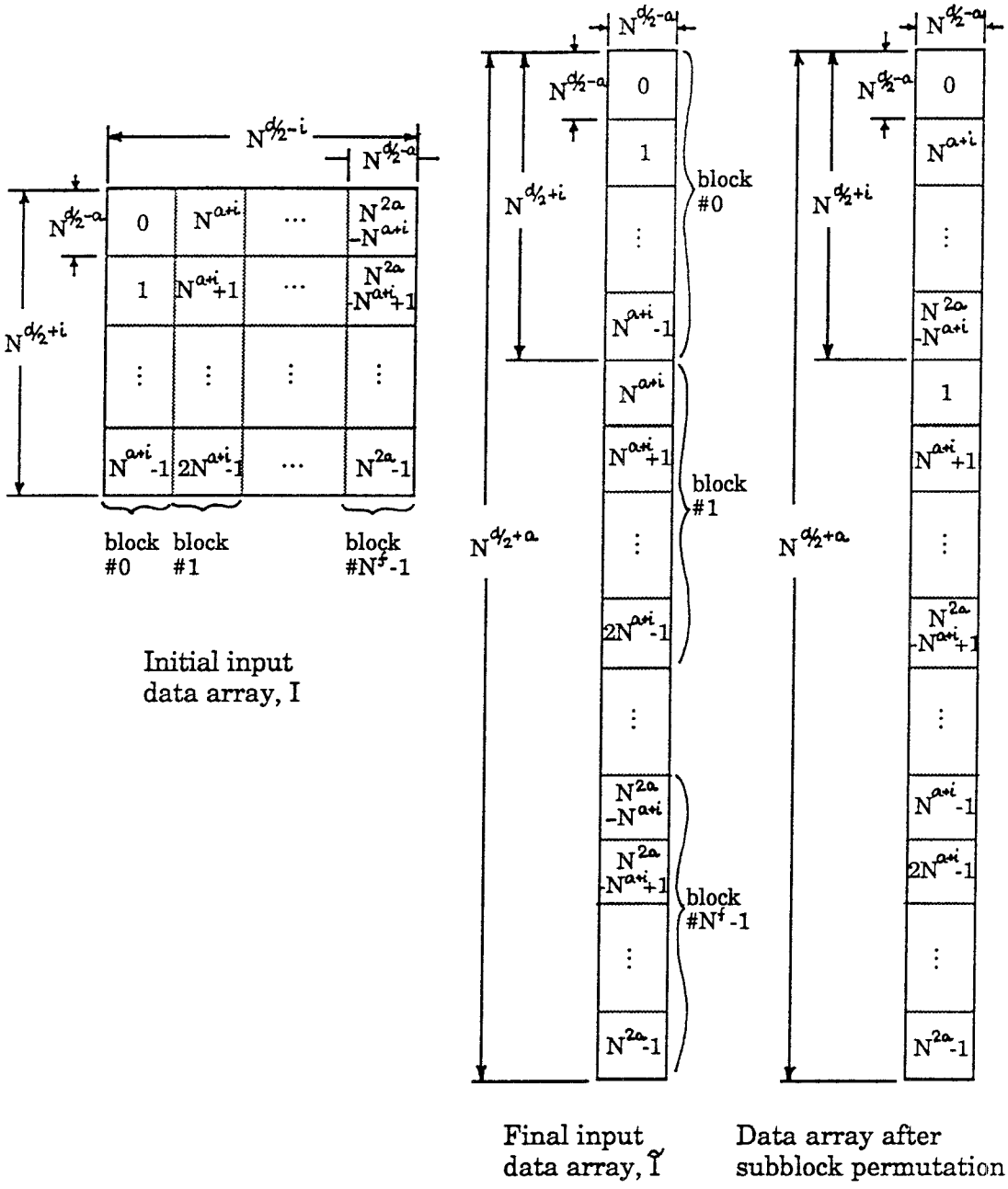


Figure 3.6: Subblock configurations in I, in $\tilde{I}$, and after subblock rotation

and permuting the subblocks. Later in this subsection we will describe the significance of these operations. Figure 3.6 shows the subblock configurations in I, $\tilde{I}$ and after subblock permutation. The subblocks are numbered in column major order in $\tilde{I}$. Notice that after subblock permutation, the ordering is in row major order.

The algorithm for computing d-dimensional DXT on $\tilde{I}$ is as follows.

1. For each column of a block perform $(\frac{d}{2}+i)$-dimensional DXT
2. Transpose subblocks
3. Permute subblocks
4. For each column of a block perform $(\frac{d}{2}-i)$-dimensional DXT

Figure 3.7 illustrates the corresponding architecture.

The analysis of this algorithm is as follows. In the input data array I, all the elements in the columns have the same values for indexes $n_{\frac{d}{2}+i+1},\ldots,n_d$ and all the elements in the rows have the same values for indexes $n_1,\ldots,n_{\frac{d}{2}+i}$. When this array is reorganized into $\check{I}$, every row of I is split into N^f parts and each part occurs in a block of $\tilde{I}$. In particular, the kth row of I occurs in the kth row of every block of $\tilde{I}$. Thus there are N^f rows with the same values for indexes $n_1,\ldots,n_{\frac{d}{2}+i}$ in $\hat{I}$. Moreover all the elements in a column of a block of $\tilde{I}$ have the same value for indexes $n_{\frac{d}{2}+i+1},\ldots,n_{d-1}$.

In Step 1 of the algorithm DXT(N) is computed over each of the indexes $n_1,\ldots,n_{\frac{d}{2}+i}$. This is performed by circulating the data on each column of a

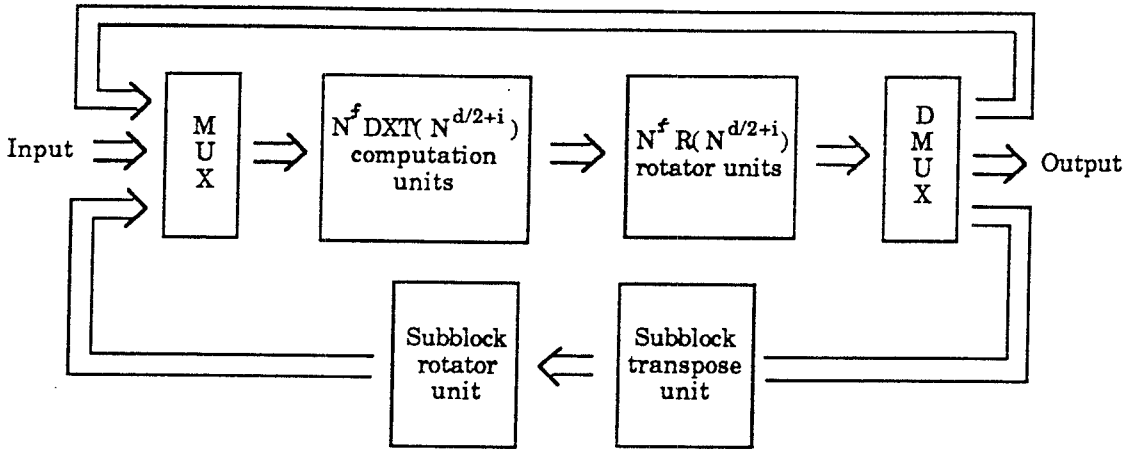


Figure 3.7: Architecture for d-dimensional DXT when the input data is a two-dimensional array of size $N^{\frac{d}{2}+a} \times N^{\frac{d}{2}-a}$

block $(\frac{d}{2}+i)$ times through a $DXT(N^{\frac{d}{2}+i})$ computation unit and a $R(N^{\frac{d}{2}+i})$ rotator unit. The $DXT(N^{\frac{d}{2}+i})$ computation unit consists of $N^{\frac{d}{2}+i-1}$ DXT(N) mesh of trees networks. The rotator unit $R(N^{\frac{d}{2}+i})$ rotates data of size $N^{\frac{d}{2}+i}$ over $(\frac{d}{2}+i)$ indexes. Note that there are N^f sets of $DXT(N^{\frac{d}{2}+i})$ computation units and $R(N^{\frac{d}{2}+i})$ rotator units. In Steps 2 and 3 of the algorithm, $N^{\frac{d}{2}-i}$ elements with the same values for indexes $n_1, n_2, \ldots, n_{\frac{d}{2}+i}$ are made adjacent to each other (note that these elements were adjacent along a row in I). These two steps are essential for the computation of $DXT(N^{\frac{d}{2}-i})$ in Step 4. We next describe the *subblock transpose* unit which transposes the subblocks in Step 2 and the *subblock rotator* unit which permutes the subblocks in Step 3.

Subblock transpose unit:

The subblock transpose unit consists of N^{2a} subunits. each of which trans-

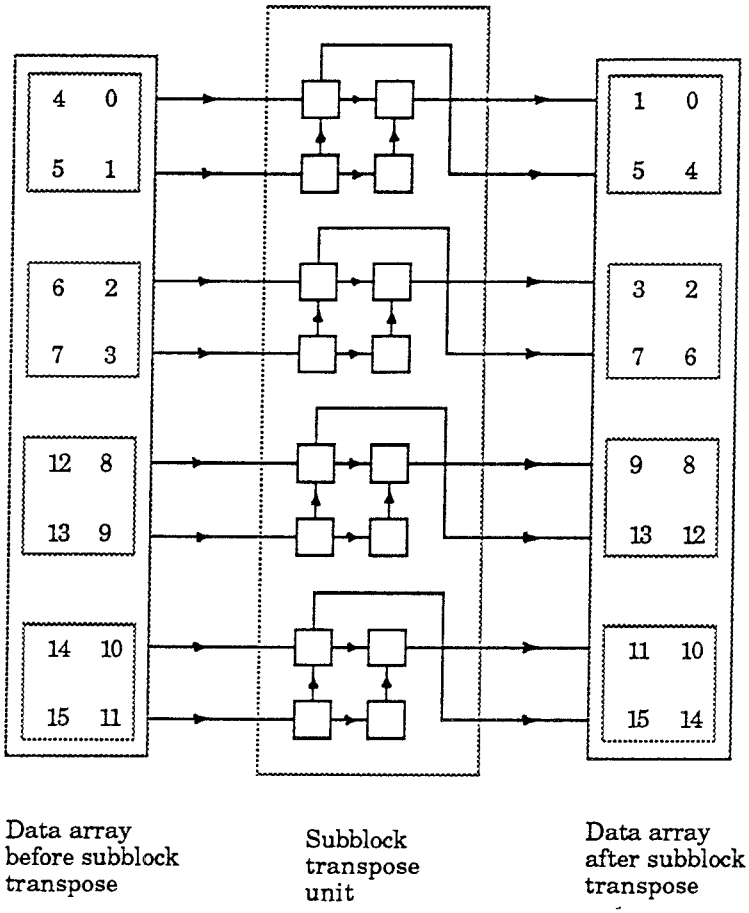


Figure 3.8: Subblock transpose unit ($N = 4, d = 2, i = 0, f = \frac{1}{2}$)

poses a subblock. Each of the subunits behave like a conventional transpose unit, that is, subblock X which is input to a subblock transpose subunit is related to subblock Z which is output by this subunit by $Z = X^T$. Figure 3.8 illustrates the function of a subblock transpose unit in an example with $N = 4$, $d = 2$, $i = 0$ and $f = \frac{1}{2}$. Since a subblock has equal dimensions, the size of the transposed data array and input data array are the same. The area of this unit is $O(nb)$ and the computation time is $O(N^{\frac{d}{2}-a}b)$.

After subblock transpose, the $N^{\frac{d}{2}-i}$ elements with the same value for indexes

$n_1, n_2, \ldots, n_{\frac{d}{2}+i}$ are in the same column of the array. These $N^{\frac{d}{2}-i}$ elements are split into N^f groups of size $N^{\frac{d}{2}-a}$ and occur in every N^{a+i}th subblock. In Step 3 the subblock rotator unit permutes the subblocks so that subblocks which are N^{a+i} apart are now adjacent.

Subblock rotator unit:

The subblock rotator unit permutes the subblocks in such a way that the ordering is changed from column major order to row major order (see Figure 3.6). The design of the subblock rotator unit is as follows. Let $N = c^m$. Then each of the N^{2a} subblocks can be represented by $2am$ indexes such that each index takes on a value between 0 and $c-1$. In the subblock rotator unit, the subblock in position k is rotated to position l, where l is obtained by rotating the indexes of k by mf positions to the left. If $\frac{2a}{f}$ is an integer, then the area of the subblock rotator unit can be estimated along the same lines as that of the rotator unit in Section 3.3.1. The number of non-cyclic subblock necklaces is at most $O(\frac{c^{2am}}{2a/f}) = O(\frac{N^{2a}}{2a/f})$. Each non-cyclic subblock necklace can be laid out in at most $O(2a/f)$ subblock-columns, where a subblock-column consists of $N^{\frac{d}{2}-a}$ columns, one per subblock column element. Thus the total number of subblock-columns in the subblock rotator unit is $O(N^{2a})$. Equivalently, any subblock permutation of size N^{2a} can be accomplished by $O(N^{2a})$ subblock-columns. Since the relative position between the elements in a subblock are unaffected by the permutation, the total number of columns is $O(N^{\frac{d}{2}+a})$, and the area of the subblock rotator unit is $O(N^{d+2a})$. The computation time through this unit is $O(N^{\frac{d}{2}-a}b)$. Figure 3.9 describes the function of a subblock rotator

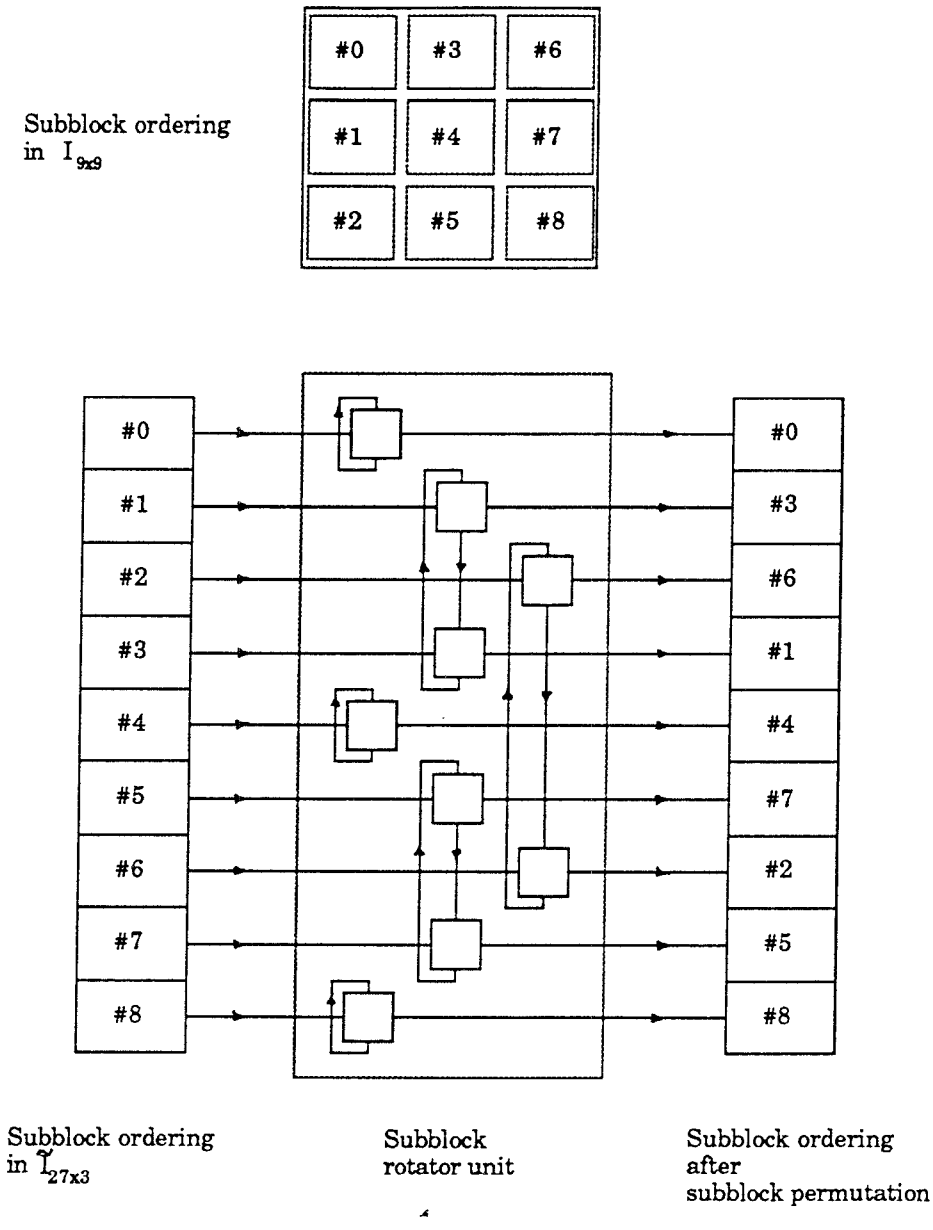


Figure 3.9: Subblock rotator unit ($N = 9$, $d = 2$, $i = 0$, $f = \frac{1}{2}$)

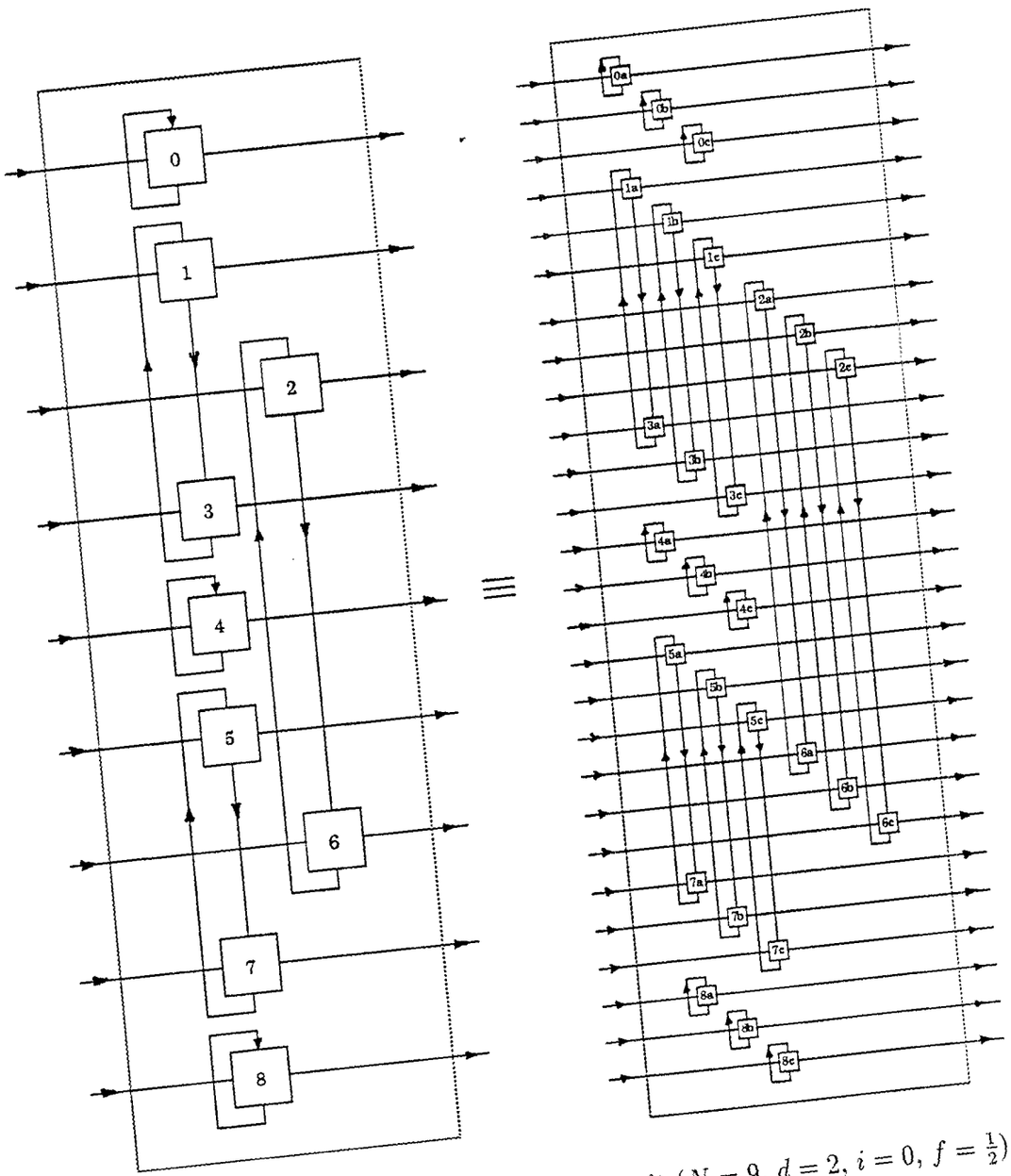


Figure 3.10: Layout of a subblock rotator unit ($N = 9$, $d = 2$, $i = 0$, $f = \frac{1}{2}$)

unit in an example with $N = 9$, $i = 0$ and $f = \frac{1}{2}$. Figure 3.10 illustrates the corresponding layout.

In Step 4 DXT(N) is computed over rest of the $(\frac{d}{2} - i)$ indexes, namely $n_{\frac{d}{2}+i+1}, \ldots, n_d$. This is performed by circulating the data on each column of a block $(\frac{d}{2} - i)$ times through the $DXT(N^{\frac{d}{2}+i})$ computation unit and the $R(N^{\frac{d}{2}+i})$ rotator unit. The rotator unit that is used to rotate over $(\frac{d}{2} + i)$ indexes can be used to rotate over the remaining $(\frac{d}{2} - i)$ indexes, since rotation over an index is obtained by cyclic left shift of the index representation.

Area and time complexities:

The area of the input MUX and the output DMUX are $O(N^{\frac{d}{2}+a})$. The area of N^f $DXT(N^{\frac{d}{2}+i})$ computation units and N^f $R(N^{\frac{d}{2}+i})$ rotator units are $O(N^{\frac{d}{2}+a+1})$ and $O(N^{d+a+i})$ respectively. The area of the subblock transpose unit is $O(nb)$ and that of the subblock rotator unit is $O(N^{d+2a})$. Thus $A_{max} = O(N^{d+2a})$ for $a \geq \frac{1}{2}\log_N b$.

The computation time for the $DXT(N^{\frac{d}{2}+i})$ unit and the $R(N^{\frac{d}{2}+i})$ rotator unit is $O(N^{\frac{d}{2}-a}b)$. The time taken to compute Steps 1 and 4 are $O((\frac{d}{2}+i)N^{\frac{d}{2}-a}b)$ and $O((\frac{d}{2}-i)N^{\frac{d}{2}-a}b)$ respectively. The computation time for the subblock transpose unit and the subblock rotator unit is $O(N^{\frac{d}{2}-a}b)$. Thus the time taken to compute Steps 2 and 3 is $O(N^{\frac{d}{2}-a}b)$. The total computation time T_{max} is thus $O(dN^{\frac{d}{2}-a}b)$, which can be approximated to $O(N^{\frac{d}{2}-a}b)$ for small d. Thus $AT^2 = O(n^2b^2)$ for all a in the range $\frac{1}{2}\log_N b \leq a \leq \frac{d}{2}$ such that N^a is an integer.

3.4 Conclusion

In this chapter we have presented a family of optimal VLSI architectures for computing $(N \times N \times \ldots \times N)$ d-dimensional linear separable transforms with area-time trade-offs. The architecture consists of one-dimensional DXT(N) computation units which compute DXT(N) over one index, and permutation units which order data so that when the data is circulated back through the DXT(N) computation units, DXT(N) can be computed over the next index.

We use the mesh of trees network to compute one-dimensional DXT(N). Though this network satisfies the optimality criteria of VLSI complexity theory, it is costly from a VLSI designers' point of view. Replacing this network by a systolic array would result in a non-optimal but practical design. The permutation unit consists of a rotator unit, a subblock transpose unit and a subblock rotator unit. The layout of all these units is very regular.

The architecture has an area $A = O(N^{d+2a})$ and computation time $T = O(dN^{\frac{d}{2}-a}b)$ for $b = \Omega(\log M)$ bit precision, $\frac{1}{2}\log_N b \leq a \leq \frac{d}{2}$. Thus there exists a family of architectures with different values of a which are optimal with $AT^2 = O(n^2b^2)$ for small d. It is reasonable to assume that d is a small constant. To the best of the author's knowledge, $d \leq 4$ for almost all known applications requiring multi-dimensional transforms.

Chapter 4

Template Matching

4.1 Introduction

In this chapter we propose a semi-systolic architecture consisting of a linear array of P processors to compute template matching between an input image of size $(N \times N)$ and a template of size $(K \times K)$. The input data is read in the line scan mode. This necessitates either storing part of the input image off-chip or on storage devices on-chip. In our architecture the on-chip storage device consists of shift registers in each processor. The shift registers are circulated so that the processor array can access the same input multiple times. For computation of real-time template matching, the number of processors in our architecture is a function of the the frame size, the template size and the internal clock cycle. This feature makes our architecture more versatile compared to the architectures of [17, 28, 41] where the number of processors is fixed (irrespective of the frame

specifications). The data flow in our architecture is very regular. Moreover no additional circuitry is required to ensure smooth data flow from the on-chip storage devices to the processor array. Our architecture achieves optimal speed-up.

The rest of the chapter is organized as follows. In Section 4.2 we give the preliminaries for real-time template matching. In Section 4.3 we give an overview of the proposed architecture and algorithm and then give a detailed description in Section 4.4. We make some concluding remarks in Section 4.5.

4.2 Preliminaries

In this section we define template matching and give an estimate of the minimum number of processors that are required for its real-time computation. We then give a brief description of the existing schemes for template matching.

4.2.1 Definition

Let the input image I be an array of size $(N_1 \times N_2)$. If W is a template of size $(K \times K)$, then the template matching of W with I is defined by

$$TM[i,j] = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} I[i+m, j+n] * W[m,n],$$

$0 \leq i \leq N_1 - K$, $0 \leq j \leq N_2 - K$. Figure 4.1 shows an $(N_1 \times N_2)$ input image, a $(K \times K)$ template and a particular match configuration. We assume that the position of the template is defined by the coordinates of the input image covered

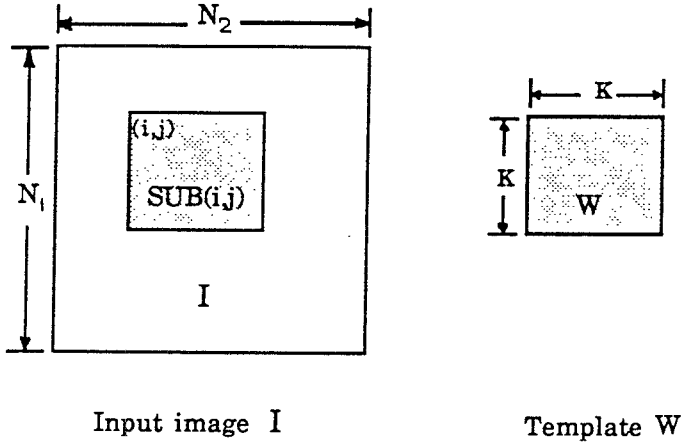


Figure 4.1: Input image I, template W and a particular match configuration

by the upper left corner of the template. Let SUB(i,j) be the input block of size $(K \times K)$ whose upper left corner is (i,j) as shown in Figure 4.1. The template is matched against every sub area SUB(i,j), and for each such configuration TM$[i,j]$ is computed.

Template matching is an extremely time consuming process since K^2 multipli cation-accumulation [1] operations have to be computed for each of the $(N_1 - K + 1)(N_2 - K + 1)$ match configurations in a frame. The total computation time per frame is thus $T = (N_1 - K + 1)(N_2 - K + 1)K^2 \alpha T_c$, where T_c is the internal clock period, α is a constant such that αT_c is the pipelined time required to compute one multiplication-accumulation. If the number of processors is P, and if the speed-up is optimal, then for real-time processing, the P-processor computation time $\frac{T}{P}$ is related to the frame frequency f_F by $\frac{T}{P} \leq \frac{1}{f_F}$. Thus the minimum value of P for real-time processing is given by $P = \lceil (N_1 - K +$

[1] The operation $Z = Z + A * W$ is known as multiplication-accumulation and the corresponding computation unit is known as multiplier-accumulator.

$1)(N_2 - K + 1)K^2\alpha T_c f_F]$. In 1.6 micron technology, if $T_c = 50ns$ and if the operations are bit parallel with 8 bits per word, then the pipelined time required to compute one multiplication-accumulation is $400ns$. For the case when the size of the template is (8×8), the minimum number of processors required for Video Telephone Standard ($N_1 = 288, N_2 = 352, f_F = 10Hz$) is 25 and for NTSC video signal ($N_1 = 512, N_2 = 480, f_F = 10Hz$) is 62.

4.2.2 Related work

There are a large number of systolic and semi-systolic architectures for computing template matching [17, 21, 22, 27, 32, 35, 36, 37, 51]. These architectures differ in the number of processors, in the algorithms and in the mappings of the algorithms into the architectures. In all these architectures the computations in the processor array are very efficient. However not all these architectures handle I/O efficiently.

In most image processing applications, the input is fed in the line scan mode. In this mode the neighboring pixels along a column are separated in time by a whole line duration. This also means that in order to compute template matching with a $(K \times K)$ template, a part of the input image (in fact $(K - 1)$ rows) and/or intermediate results have to be stored. There are two classes of architectures depending on whether the $(K - 1)$ rows of the input image are stored in an external memory or on-chip.

In the class of architectures which consist of an external memory, there are

some with a high I/O bandwidth of K pixels per clock cycle [27, 51]. In these architectures the template shifts by one column in every clock cycle. There are others which have an even larger I/O bandwidth and complicated data flow, since the same input has to be fed to different processors at different times [21, 37].

The architectures with on-chip storage have an I/O bandwidth of one pixel per clock cycle. The data is stored in a FIFO line memory [17], shift-buffer pipeline [41], etc. When the size of the template is large, on-chip storage takes up a significant amount of the chip area. Thus there exists a trade-off between high I/O bandwidth and internal storage requirements. Recently Jutand et al [28] proposed an architecture which tries to establish a balance between the two by incorporating both external memory and on-chip storage. The size of the on-chip storage in their architecture is only K^2 pixels and the I/O bandwidth is two pixels per clock cycle. The number of processors in all these architectures is however fixed at K^2. Thus these architectures are not versatile enough to compute template matching in real-time for any frame specification. Moreover additional circuitry is required in order that the data flow from the on-chip storage to the processor array is regular.

4.3 Overview

4.3.1 Model

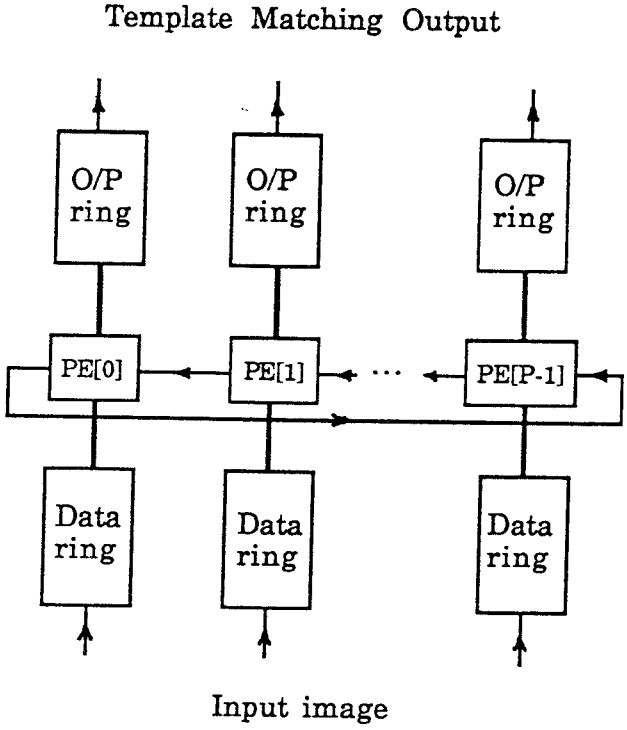


Figure 4.2: P-processor architecture

We propose an architecture consisting of a linear array of P processors. Each processor essentially consists of a multiplier-accumulator, a few storage registers and two unidirectional shift registers as shown in Figure 4.2. A single number can be stored in a storage register. Each shift register consists of a linearly connected array of storage cells. When the shift register is clocked, the output of the ith storage cell is transferred to the $(i-1)$th storage cell in 1 clock cycle, where $1 \leq i < L$ and L is the length of the shift register. When $i = 0$, the output of the shift register is transferred to a storage register. The input to the shift register is usually data from a storage register. A part of the input image is stored in the shift register, *data ring* (see Figure 4.2). The storage is essential because data from the input image is read only once and the same data

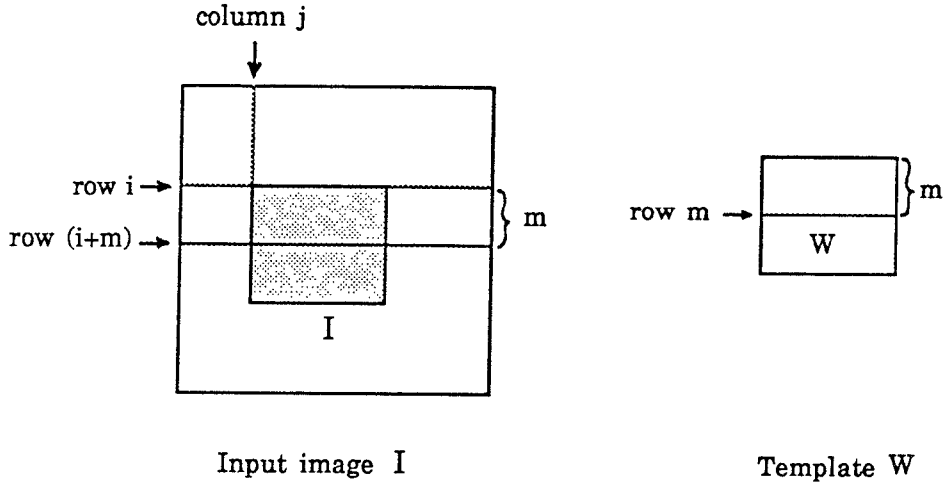


Figure 4.3: A match configuration during the computation of template matching outputs of row i

is required for the computation of multiple outputs. The data ring is circulated so that the processors can compute on the same input multiple times. The intermediate template matching outputs are stored in another shift register, *output ring* (see Figure 4.2). This unit is necessary, since the input is fed in line scan mode, the number of processors is less than the size of a row, and as a result each processor has to compute multiple intermediate template matching outputs per row. The output ring is circulated so that the intermediate template matching outputs can be updated.

4.3.2 Algorithm and Mapping for Template Matching

Let $Y[i,j,m] = \sum_{n=0}^{K-1} I[i+m, j+n] * W[m,n]$, that is, $Y[i,j,m]$ is the *row-inner-product* of the mth row of the template and the mth row of SUB(i,j).

Then $TM[i,j] = \sum_{m=0}^{K-1} Y[i,j,m]$, that is, $TM[i,j]$ is the sum of K row-inner-products (corresponding to K rows of the template). Without loss of generality we assume that $N_1 = N_2 = N$. The algorithm for template matching can be summarized as follows.

Algorithm *temp.match* :

1. *for* each row i of input image *do*
2. *for* each row m of template *do*
3. *for* each position j of row $(i+m)$ of input image *do*
4. Compute $Y[i,j,m]$ over row $(i+m)$ of input image
5. Set $TM[i,j] = TM[i,j] + Y[i,j,m]$

The order of the *for* loops in algorithm *temp.match* is important. During the computation of template matching outputs of row i, the mth row of the template is matched with the $(i+m)$th row of the input image for each m, $0 \leq m < K$ (see Figure 4.3). Thus rows i through $(i+K-1)$ of the input image are required for the computation of template matching outputs of row i.

The mapping of algorithm *temp.match* into the P processor architecture is as follows, $P \geq K$ and P divides N evenly. The input image is read in line scan mode and stored in cyclic mode in the data rings of the P processors. This means that $I[i,j]$ of the input image is read into the data ring of processor $p = j \bmod P$, $0 \leq i,j < N$. Thus each row i of the input image is divided into N/P sections and each section of P consecutive elements is stored in the data rings of P processors. The P processors compute template matching outputs for each row i of the input image in the following way. The processors compute

a set of $(N-K+1)$ row-inner-products between the $(i+m)$th row of the input image and the mth row of the template for each m, $0 \leq m < K$ (steps 2, 3 and 4 of *temp.match*). These row-inner-products are used to update the intermediate template matching outputs stored in the output rings. After K updates, the output ring of processor $p = j \bmod P$ contains the template matching output $TM[i,j]$.

4.4 Detailed description

In this section we give the details of the processor architecture and algorithm for template matching. We first discuss the operations at the processor level and then elaborate on the I/O issues.

4.4.1 Processor Architecture

In the previous section we have seen that each processor consists of a multiplier-accumulator, a few storage registers, a data ring to store data from the input image and an output ring to store the intermediate template matching outputs. Figure 4.4 illustrates the design of a single processor.

The data rings of the P processors store K rows of the input image. This is necessary since the the pixels of the input image are read only once, and since the rows of the input image that are required for the computation of template matching outputs of two consecutive rows overlap by $(K-1)$ rows. Thus a data

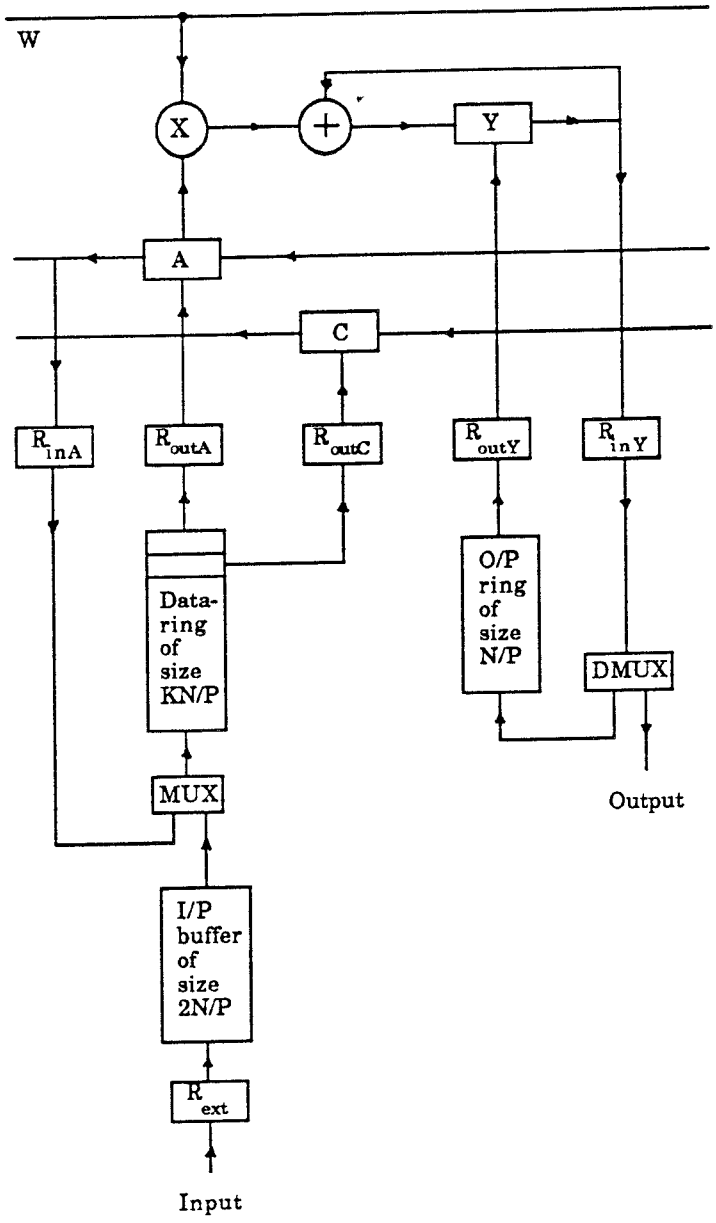


Figure 4.4: Design of a processor for template matching

ring is of size KN/P. The elements in the data ring are partitioned into K *blocks*, such that each block contains N/P elements of the input image which were in the same row. Figure 4.5 shows the elements in the data rings prior to the computation of template matching outputs of row 1, in an example where the image is of size (8×8), the template is of size (3×3) and the number of processors is 4. We refer to this example as example A.

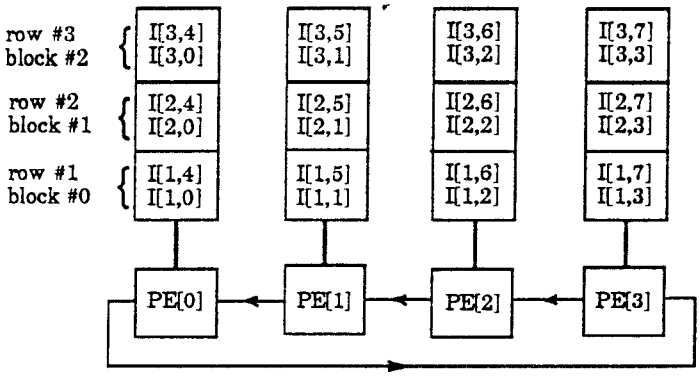
The template values are broadcast to the processors via the W line (see Figure 4.4). The same template value is broadcast to all the processors.

The size of the output ring is $\lceil \frac{N-K+1}{P} \rceil = \frac{N}{P}$, since each processor computes at most $\lceil \frac{N-K+1}{P} \rceil$ intermediate template matching outputs along a row. The intermediate template matching outputs are stored in cyclic mode in the output ring and are updated K times (once for each of the K rows of the template).

4.4.2 Processor Algorithm

We next discuss the operations of processor p in the computation of template matching outputs $TM[i,j]$, where $p = j \bmod P$ and $0 \leq i,j \leq N-K$. At the beginning of the computation of the template matching outputs of row i, the data ring in each processor contains the elements corresponding to rows i through $(i+K-1)$ of the input image in blocks 0 through $(K-1)$ (see Figure 4.5). For each of the K rows of the template $(0 \leq m < K)$, processor p computes a set of $\lceil \frac{N-K+1}{P} \rceil$ $Y[i,j,m]$s, where $j = \gamma P + p$ and $0 \leq \gamma < \lceil \frac{N-K+1}{P} \rceil$. In other words, processor p computes $\lceil \frac{N-K+1}{P} \rceil$ row-inner-products of the mth

Data ring configuration prior to the computation of Y[1,*,0] :



Register contents during the computation of Y[1,*,0] :

C	I[1,4]	I[1,5]	I[1,6]	I[1,7]
A	I[1,0]	I[1,1]	I[1,2]	I[1,3]
A*W	I[1,0]*W[0,0]	I[1,1]*W[0,0]	I[1,2]*W[0,0]	I[1,3]*W[0,0]
C	I[1,5]	I[1,6]	I[1,7]	—
A	I[1,1]	I[1,2]	I[1,3]	I[1,4]
A*W	I[1,1]*W[0,1]	I[1,2]*W[0,1]	I[1,3]*W[0,1]	I[1,4]*W[0,1]
C	I[1,6]	I[1,7]	—	—
A	I[1,2]	I[1,3]	I[1,4]	I[1,5]
A*W	I[1,2]*W[0,2]	I[1,3]*W[0,2]	I[1,4]*W[0,2]	I[1,5]*W[0,2]
	Y[1,0,0]	Y[1,1,0]	Y[1,2,0]	Y[1,3,0]

Figure 4.5: Data ring configuration prior to the computation of $Y[1,*,0]$ and contents of registers A and C during the computation of $Y[1,*,0]$ in example A

1. load A, C from data ring

   load Y from output ring

2. *for* $q := 0$ to $K-1$ *do*

3. broadcast W[m,q]

4. $Y := Y + A * W$ $\quad$ (*if* $flag = 0$)

5. Shift-left A and C by 1

6. load Y into output ring

The time taken to compute a row-inner-product is $K\alpha T_c$. The control input $flag$ is set to 1 in processors $(K-1)$ to $(P-1)$ in the N/Pth iteration so that no invalid outputs are computed. The data ring and the output ring are shifted and the above process is repeated $\lceil \frac{N-K+1}{P} \rceil = \frac{N}{P}$ times. Recall that processor p has to compute $\lceil \frac{N-K+1}{P} \rceil$ row-inner-products of the mth row of the template and the $(i+m)$th row of the input image. The data ring is clocked at $K\alpha T_c$.

4.4.3 I/O operations

We next discuss how the input image is loaded into the data ring. While the template matching outputs of row i are being computed, row $(i+K)$ is read from the input image into the input buffers (see Figure 4.4) of the P processors. Thus a row of the input image is loaded into the input buffers in the time taken to compute all the template matching outputs of a row. We propose two input loading schemes which differ in the way the data is loaded from the input buffer into the data ring.

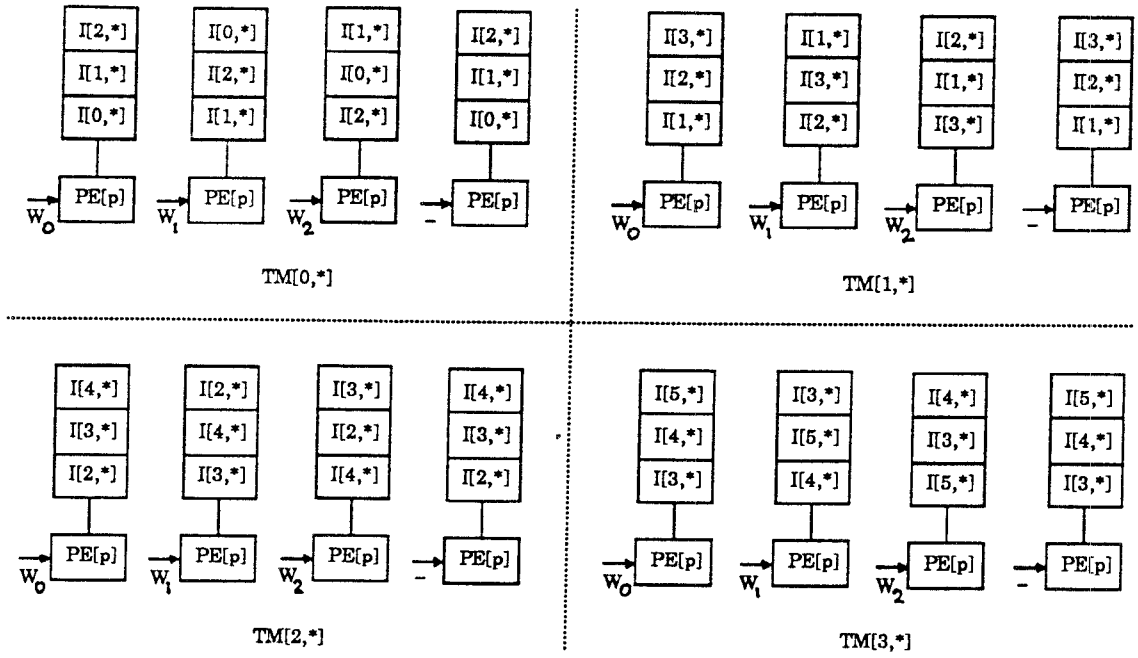


Figure 4.6: Data ring configuration of processor p during the computation of $TM[0,*]$, $TM[1,*]$, $TM[2,*]$, $TM[3,*]$ by Scheme 1 in example A

Scheme 1:

In this scheme the data ring is circulated $(K+1)N/P$ times for the computation of the template matching outputs of row i, $0 \leq i < N-K$. It is circulated KN/P times so that all the elements of the data ring corresponding to rows i through $(i+K-1)$ are incident on the processor array (and can be used for the computation of template matching outputs of row i) and then for another N/P times so that the elements corresponding to row i are once again in block 0. In the next step the elements corresponding to the $(i+K)$th row are shifted in from the input buffer (while the elements corresponding to the ith row are shifted out). At the end of this step the data ring contains elements corresponding to

rows $(i+1)$ through $(i+K)$, which are required for the computation of template matching outputs of row $(i+1)$. The above scheme is illustrated in Figure 4.6 in example A ((8×8) image, (3×3) template and 4 processors).

Let W_j be the jth row of the template. The sequence of W_js that are broadcast to the processor array are the same for all rows (see Figure 4.6). The input buffer is of size N/P. The time taken to compute all the template matching outputs of a row is $K(K+1)\alpha T_c N/P$. Note that this scheme does not achieve optimal speed-up. This is because the processor array is idle for $K\alpha T_c N/P$ time for every $K^2\alpha T_c N/P$ time that it is active.

Scheme 2:

This scheme achieves optimal speed-up by pipelining the data in the following way. Row $(i+K+j)$ is loaded into the data ring from the input buffer during the computation of $Y[i+j,*,j]$, where $i = \beta K$, β is an integer in the range $0 \leq \beta \leq \lfloor \frac{N-K+1}{K} \rfloor$, and $0 \leq j < K$. Recall that $Y[i+j,*,j]$ is the jth row-inner-product in the computation of template matching outputs of row $(i+j)$. Thus row $(i+K)$, row $(i+K+1)$, ..., row $(i+2K-1)$ are loaded into the data ring during the computations of $Y[i,*,0]$, $Y[i+1,*,1]$, ..., $Y[i+K-1,*,K-1]$ respectively. The above scheme is illustrated in Figure 4.7 in example A ((8×8) image, (3×3) template and 4 processors). Here rows 3, 4, 5, and 6 are written into the data ring during the computation of $Y[0,*,0]$, $Y[1,*,1]$, $Y[2,*,2]$, and $Y[3,*,0]$ respectively. The input buffer is of size $2N/P$ since the elements of two rows, row $(i+2K-1)$ and row $(i+2K)$, are loaded into the data ring one

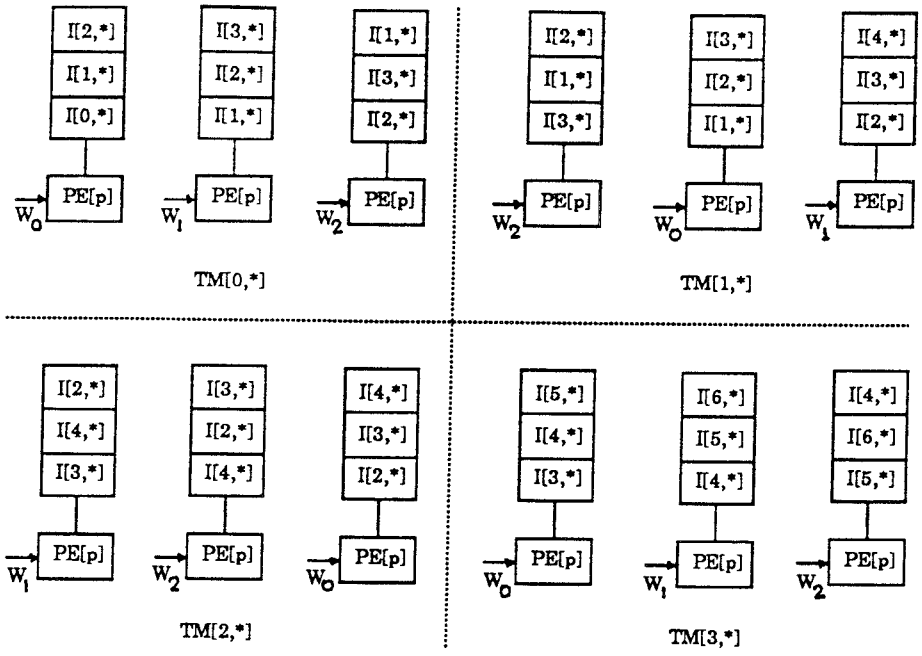


Figure 4.7: Data ring configuration of processor p during the computation of $TM[0,*]$, $TM[1,*]$, $TM[2,*]$, $TM[3,*]$ by Scheme 2 in example A

after the other [3].

The sequence of W_js that are broadcast to the processor array are not the same for the computation of template matching outputs of all rows. There are K different W_j sequences, such that the sequence for $TM[i,*]$ is the same as that for $TM[i+K,*]$. The sequence of W_js that are broadcast to the processor array for the computation of template matching outputs of rows i through $(i+K-1)$, where $i = \beta K$ and $0 \leq \beta \leq \lfloor \frac{N-K+1}{K} \rfloor$, is as follows:

[3]Row $(i+2K-1)$ is loaded during the computation of $Y[i+K-1,*,K-1]$ (the last row-inner-product in the computation of template matching outputs of row $(i+K-1)$) and row $(i+2K)$ is loaded during the computation of $Y[i+K,*,0]$ (the first row-inner-product in the computation of template matching outputs of row $(i+K)$).

$\underbrace{(W_0, W_1, \ldots, W_{K-1})}_{TM[i,*]}, \underbrace{(W_{K-1}, W_0, \ldots, W_{K-2})}_{TM[i+1,*]}, \ldots\ldots, \underbrace{(W_1, W_2, \ldots, W_0)}_{TM[i+K-1,*]},$
$\underbrace{(W_0, W_1, \ldots, W_{K-1})}_{TM[i+K,*]}, \ldots$. The template values in a row are broadcast in row major order. In the example illustrated in Figure 4.7, there are 3 sets of W_j sequences, namely, (W_0, W_1, W_2), (W_2, W_0, W_1), and (W_1, W_2, W_0). Figure 4.8 describes the details of this algorithm.

4.5 Conclusion

In this chapter we have described a semi-systolic architecture consisting of a linear array of P processors to compute template matching when the input is fed in the line scan mode. The I/O bandwidth problem is handled by storing part of the input image in shift registers in each processor, and by circulating the shift registers.

The performance of the architecture that we have developed can be affected by the delay in broadcasting the template values. An alternative scheme consists of storing the template values in shift registers in each processor and clocking them to the multiplier-accumulator. While this scheme is useful in applications of adaptive filtering, it results in an additional on-chip storage. Another possibility is to partition the processor array into α groups, such that each group stores a set of template values which are locally broadcast to the processors of that group.

```
begin    {algorithm}
    for i := 0 to N − K do begin
        Initialize output ring;
        for m := 0 to K − 1 do begin
            for r := 0 to N/P − 1 do begin
                Y = R_outY; A = R_outA; C = R_outC;
                Shift data ring and output ring;
                if ((i − m) mod K = 0)
                    then data ring reads from input buffer;
                    else data ring reads from R_inA;
                R_inA = A;
                for q := 0 to K − 1 do begin
                    Y := Y + A * W      (if flag=0)
                    Shift-left A and C by 1;
                end    {of q-loop}
                R_inY = Y;
            end    {of r-loop}
        end    {of m-loop}
    end    {of i-loop}
end    {algorithm}
```

Figure 4.8: Algorithm for computing template matching when the size of the image is $(N \times N)$, the size of the template is $(K \times K)$, and the number of processors is P

The number of processors required for real-time template matching is quite large. Though it is not unreasonable to assume that a single chip implementation is possible with submicron technology, it is interesting to investigate how the processors can be organized into multiple chips such that the inter-chip communication overhead is minimum.

Chapter 5

Block Matching

5.1 Introduction

In this chapter we propose a semi-systolic architecture consisting of a linear array of P processors to compute full-search block matching when the frames are of size $(N \times N)$, the blocks are of size $(K \times K)$, and the search area is of size $(K + q) \times (K + q)$. The computation unit in each processor consists of a comparator-accumulator and a few registers. Data from the current frame and previous frame-memory are read in line scan mode. Since block matching is a window based operation, this necessitates storing part of the input data either off-chip or on-chip in storage devices. The intermediate results have to be stored too, since the computations are split over K lines. As in the case of template matching, the on-chip storage device consists of shift registers in each processor. For computation of real-time block matching, the number of processors in our architecture is a function of the frame size, the block size, and the internal clock

rate. This makes our architecture more versatile compared to the architectures with fixed number of processors [7, 18, 20, 30]. The data flow in our architecture is very regular. Moreover no additional circuitry is required to ensure smooth data flow from the on-chip storage devices to the processor array.

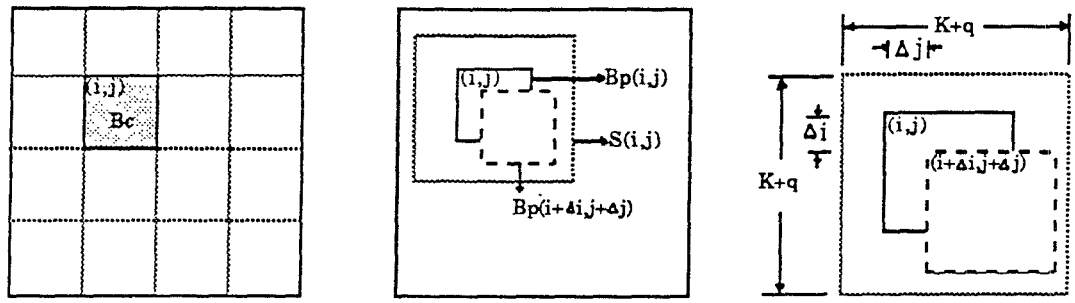
The rest of the chapter is organized as follows. In Section 5.2 we give the preliminaries for full-search block matching. In Section 5.3 we give an overview of the proposed architecture and algorithms and then give a detailed description in Section 5.4. We make some concluding remarks in Section 5.5.

5.2 Preliminaries

In this section we define full-search block matching and give an estimate of the minimum number of processors that are required for its real-time computation. We then give a brief description of the existing schemes for block matching.

5.2.1 Definition

Full-search block matching determines a *displacement vector* for every *reference* block in the current frame, by comparing it with all *candidate* blocks in a search area surrounding the position of the reference block in the previous frame. Let the current frame I_c of size $(N_1 \times N_2)$ be partitioned into reference blocks of size $(K \times K)$. Thus there are $N_1 N_2 / K^2$ disjoint reference blocks in a frame. Let $B_c(i,j)$ be a reference block of I_c whose top-leftmost coordinate is (i,j), and let the corresponding search area $S(i,j)$ in the previous frame I_p be of size $(K+q) \times (K+q)$. Let $B_p(i+\Delta i, j+\Delta j)$ be a candidate block in $S(i,j)$ whose



Reference block Bc(i,j) in current frame

Candidate blocks Bp in search area S(i,j) in previous frame

Enlarged view of search area

Figure 5.1: Reference block $B_c(i,j)$ and candidate blocks $B_p(i,j)$, $B_p(i+\Delta i, j+\Delta j)$ in search area $S(i,j)$

top-leftmost coordinate is $(i+\Delta i, j+\Delta j)$, $-q/2 \leq \Delta i, \Delta j \leq q/2$. Figure 5.1 shows the relation between $B_c(i,j)$ and its candidate blocks in $S(i,j)$. Let $L_{i,j}(\Delta i, \Delta j)$ be the *block-distance* between reference block $B_c(i,j)$ and candidate block $B_p(i+\Delta i, j+\Delta j)$, that is, $L_{i,j}(\Delta i, \Delta j)$ is the sum of absolute difference between pixels of $B_c(i,j)$ and $B_p(i+\Delta i, j+\Delta j)$.

$$L_{i,j}(\Delta i, \Delta j) = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} |x_c(i+m, j+n) - x_p(i+m+\Delta i, j+n+\Delta j)|,$$

where x_c are the pixels of current frame I_c and x_p are the pixels of previous frame I_p. Let $(\Delta i, \Delta j)_{i,j}$ be the vector corresponding to the candidate block $B_p(i+\Delta i, j+\Delta j)$, and let $(\Delta\hat{i}, \Delta\hat{j})_{i,j}$ be the vector corresponding to the candidate block with minimum block-distance, that is,

$$(\Delta\hat{i}, \Delta\hat{j})_{i,j} = \min_{\Delta i, \Delta j}{}^{-1} L_{i,j}(\Delta i, \Delta j).$$

The vector $(\Delta\hat{i}, \Delta\hat{j})_{i,j}$ is then the displacement vector for $B_c(i,j)$.

Block matching is a computationally intensive operation. Here $(q+1)^2$ block-

distances have to be computed for each of the N_1N_2/K^2 displacement vectors in a frame. This results in $N_1N_2(q+1)^2$ comparison-accumulation [1] operations per frame and a total computation time of $T = N_1N_2(q+1)^2\alpha T_c$, where T_c is the internal clock period, α is a constant such that αT_c is the pipelined time required to compute one comparison-accumulation operation. If the number of processors is P, and if the speed-up is optimal, then the minimum value of P for real-time processing is given by $P = \lceil N_1N_2(q+1)^2\alpha T_c f_F \rceil$, where f_F is the frame frequency. If $T_c = 50ns$ and if the operations are bit parallel with 8 bits per word, then the pipelined time required to compute one comparison-accumulation is $100ns$. For the case when $K = 8$ and $q = 8$, the number of processors required for Video Telephone Standard ($N_1 = 288$, $N_2 = 352$, $f_F = 10Hz$) is 9 and for NTSC video signal ($N_1 = 512$, $N_2 = 480$, $f_F = 10Hz$) is 20.

The high computational rate of full-search block matching can be reduced by applying search strategies where lesser number of candidate blocks are searched. One such strategy is hierarchical block matching, which adopts block size and maximum displacement depending on the image properties. Hierarchical block matching is useful for applications where the objects move rapidly and a fixed displacement is not sufficient to determine the displacement vector.

5.2.2 Related work

The existing systolic or semi-systolic architectures for computing block matching [7, 18, 20, 30, 61] differ in the number of processors, and consequently in the mappings of the block matching algorithm into the processor array. In all

[1] The operation $Z = Z + |A - B|$ is known as comparison-accumulation and the corresponding computation unit is known as comparator-accumulator.

these architectures the computations in the processor array are very efficient. Unfortunately not all these architectures handle the I/O bandwidth problem efficiently.

The I/O bandwidth can be reduced if the search area data and the reference block data are read as few times as possible from off-chip sources. One way of realizing this is by storing the data in local memories on-chip. On-chip storage also results in reduced number of I/O pins.

The I/O bandwidth problem is handled in [61] by reading in the data through one input port and by broadcasting data from the current (or previous) frame-memory to all the processors. In the architecture of [7], the reference block data is input through one input port and is duplicated by shift registers on-chip. However in this architecture [7], the search area data is read multiple times from the previous frame-memory and the number of input ports is $1+(K+q)^2/K^2$. In the architecture of DeVos et al [18] the local memory consists of line buffers and register chains or memory blocks. The register chains in the two-dimensional processor array are replaced by a memory block in the linear array architecture. Though the number of transistors in the memory block architecture is less, an additional pointer circuitry is required in order that appropriate data are written into and read from the memory block.

5.3 Overview

5.3.1 Model

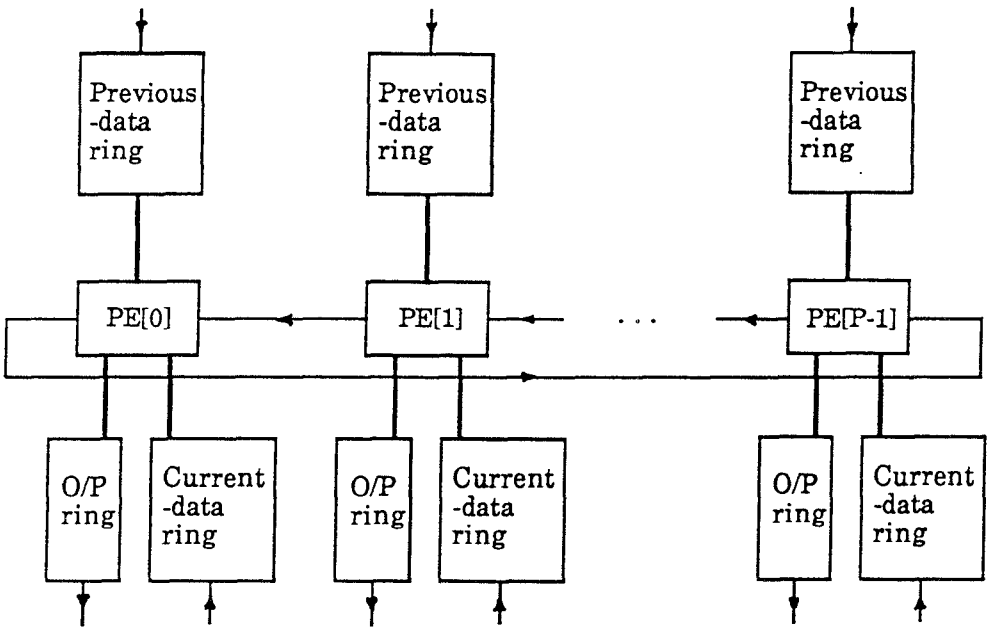


Figure 5.2: P-processor architecture

We propose an architecture consisting of a linear array of P processors. Each processor consists of a a comparator-accumulator, a few storage registers and three unidirectional shift registers, as shown in Figure 5.2. It is essential to store data from the previous and current frames, since such data are read only once, and the same data is required for the computation of multiple outputs. Data from the previous frame is stored in the shift register, *previous-data ring*, and data from the current frame is stored in the shift register, *current-data ring* (see Figure 5.2). The previous-data ring and the current-data ring are circulated so that the processors can access the same input multiple times. The intermediate block matching outputs are stored in the shift register, *output ring* (see Figure 5.2). This unit is necessary since the input is fed in line scan mode, the number of processors is less than the size of a row, and as a result each processor has to compute multiple block matching outputs per row.

5.3.2 Algorithm and Mapping for Block Matching

Let $Y_{i,j}[\Delta i, \Delta j, m] = \sum_{n=0}^{K-1} |x_c(m,n) - x_p(m+\Delta i, n+\Delta j)|$, that is, $Y_{i,j}[\Delta i, \Delta j, m]$ is the *row-distance* between the pixels of the mth row of $B_c(i,j)$ and the pixels of the mth row of $B_p(i+\Delta i, j+\Delta j)$. Then $L_{i,j}[\Delta i, \Delta j] = \sum_{m=0}^{K-1} Y_{i,j}[\Delta i, \Delta j, m]$, that is, $L_{i,j}[\Delta i, \Delta j]$ is the sum of K row-distances (corresponding to the K rows of the reference block). The displacement vector $(\Delta\hat{i}, \Delta\hat{j})_{i,j}$ is computed by first computing the vector corresponding to the best match over all Δj and then over all Δi. Without loss of generality we assume that $N_1 = N_2 = N$. The algorithm for block matching can be summarized as follows.

Algorithm *block.match* :

1. *for* $i = 0, K, 2K, \ldots$ of the current frame *do*
2. *for* each row $(i + \Delta i)$ of the previous frame *do*
3. *for* each row m of the reference block *do*
4. *for* $j = 0, K, 2K, \ldots$ of row i of the current frame *do*
5. *for* each position $(j + \Delta j)$ of row $(i + \Delta i + m)$ of the previous frame *do*
6. Compute $Y_{i,j}[\Delta i, \Delta j, m]$
7. Set $L_{i,j}[\Delta i, \Delta j] = L_{i,j}[\Delta i, \Delta j] + Y_{i,j}[\Delta i, \Delta j, m]$
8. Set $(\Delta\hat{i}, \Delta\hat{j})_{i,j} = \min^{-1}\{\min\{L_{i,j}(\Delta\hat{i}, \Delta\hat{j}), L_{i,j}(\Delta i, \Delta j)\}\}$

We chose this order of the *for* loops in algorithm *block.match* since the input from the current frame as well as the previous frame are read in line scan mode. Figure 5.3 describes a particular match configuration during the computation of displacement vector $(\Delta\hat{i}, \Delta\hat{j})_{i,j}$ corresponding to reference block $B_c(i,j)$.

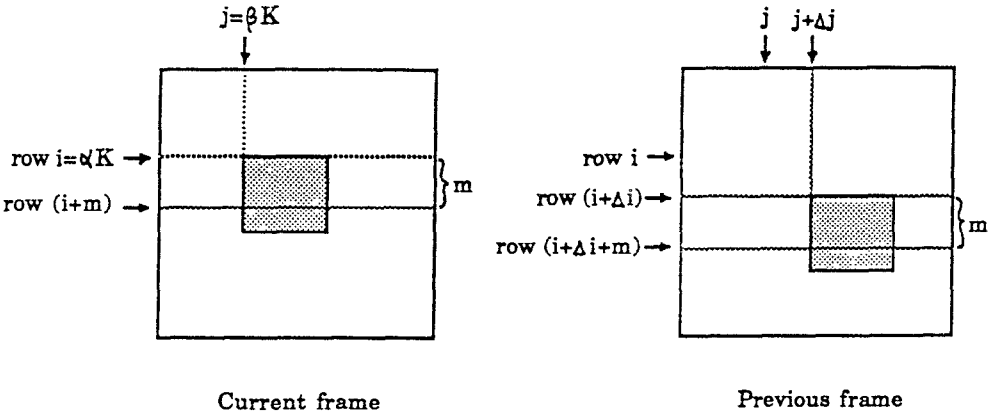


Figure 5.3: A match configuration during the computation of $(\Delta\hat{i}, \Delta\hat{j})_{i,j}$

The mapping of algorithm *block.match* into the P processor architecture is as follows. The input from the current frame is read in line scan mode and stored in cyclic mode in the current-data rings of the P processors. Thus $I_c[i,j]$ of the current frame is read into the current-data ring of processor $p = j \bmod P$, $0 \leq i,j < N$. The previous frame is appended with a ring of zeroes of width $\frac{q}{2}$ so that the same procedure can be used to compute all the displacement vectors. Thus the size of the previous frame is increased to $\hat{N} \times \hat{N}$, where $\hat{N} = N + q$. The input from the previous frame is also read in line scan mode and stored in cyclic mode in the previous-data rings of the P processors. Thus $I_p[i,j]$ of the previous frame is read into the previous-data ring of processor $p = j \bmod P$, where $0 \leq i,j < \hat{N}$.

The P processors compute N/K displacement vectors (corresponding to N/K reference blocks of a row) in every Kth row i of the current frame ($i = 0, K, 2K, \ldots$). For each Δi, the N/K reference blocks with top-leftmost coordinate in row i of the current frame are matched with their candidate blocks with top-leftmost coordinate in row $(i + \Delta i)$ of the previous frame in the following

way. A set of $(q+1)N/K$ row-distances (or sum of absolute differences over a row) is computed between the $(i+m)$th row of the current frame and the $(i+m+\Delta i)$th row of the previous frame for each m, $0 \leq m < K$ (Steps 3 through 7 of *block.match*). These row-distances are used to update the intermediate block-distances (or sum of absolute differences over a block) which are stored in the output rings of the processors. After K updates (corresponding to the K rows of the reference block), the output rings contain the correct value of $(q+1)N/K$ block-distances between N/K reference blocks whose top-leftmost coordinates are in row i of the current frame and their candidate blocks whose top-leftmost coordinates are in row $(i+\Delta i)$ of the previous frame. We refer to this set of block-distances as $blk.dist(i, i+\Delta i)$. The processors then update the temporary displacement vector for each of the N/K reference blocks of a row (Step 7 of *block.match*). Steps 2 through 7 are repeated for each Δi, $-\frac{q}{2} \leq \Delta i \leq \frac{q}{2}$. After $(q+1)$ updates (corresponding to $(q+1)$ values of Δi), the displacement vectors for N/K reference blocks of a row are computed.

5.4 Detailed description

In this section we give the details of the processor architecture and algorithm for block matching. We first discuss the operations at the processor level and then elaborate on the I/O issues.

5.4.1 Processor Architecture

The P processors in a processor array can be partitioned into δ groups, with K processors per group. Thus $P = \delta K$. Each group of processors is responsible for computing the displacement vector of N/P reference blocks of a row. There are two processor array configurations, depending on whether $K > q$ or $K \leq q$.

When $K > q$, there are two types of processors, processors of type A and of type B. While all the processors contain previous-data rings, current-data rings, and storage registers, only processors of type A contain comparator-accumulators and output rings. Out of the K processors in a group, $(q + 1)$ processors are of type A and $(K - q - 1)$ processors are of type B. The $(q + 1)$ processors of type A in a group compute the block-difference of $(q + 1)$ candidate blocks corresponding to each of the reference blocks that are assigned to that group. The processors of type B are involved only with the flow of data from/to the previous-data rings and current-data rings. Figure 5.4 illustrates the processor array configuration in an example with a (4×4) reference block, a (6×6) search area and 4 processors. We refer to this example as example B. Note that in Figure 5.4 $PE[3]$ which is of type B does not contain an output ring. Figure 5.5 illustrates the design of a processor of type A.

When $K \leq q$, all the processors are of type A. The K processors in a group now compute the block-difference of $(q + 1)$ candidate blocks corresponding to the reference block assigned to that group. Thus each processor computes the block-differences of at most $\lceil (q+1)/K \rceil$ candidate blocks for each of the reference blocks assigned to that group.

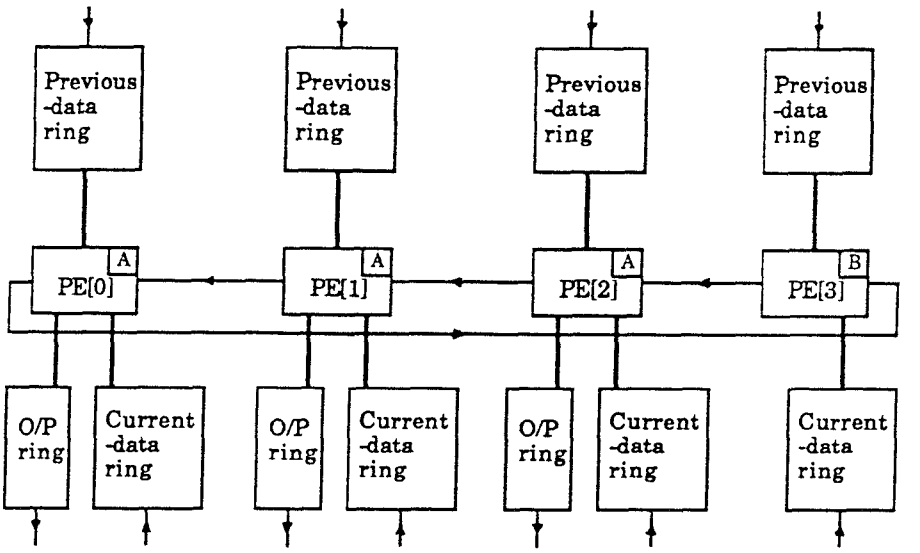


Figure 5.4: Processors of types A and B in example B

The algorithms for the case when $K > q$ and when $K \leq q$ are quite similar. In the rest of this chapter we assume that $K > q$. We will however point out the modifications in the algorithm for the case when $K \leq q$.

The previous-data rings of the P processors store K rows of the previous frame. This storage is required since the pixels of the previous frame are read only once from the external memory, and since the rows of the previous frame that are required for the computation of $blk.dist(i, i+\Delta i)$ and $blk.dist(i, i+\Delta i+1)$ overlap by $(K-1)$ rows. Thus a previous-data ring is of size $K\lceil \hat{N}/P \rceil$. The elements of the previous-data ring are partitioned into K blocks, such that a block contains the $\lceil \hat{N}/P \rceil$ elements which were in the same row in the previous frame (see Figure 5.6).

The current-data rings of the P processors store K rows of the current frame. As in the case of the previous-data ring, the storage is required since the pixels

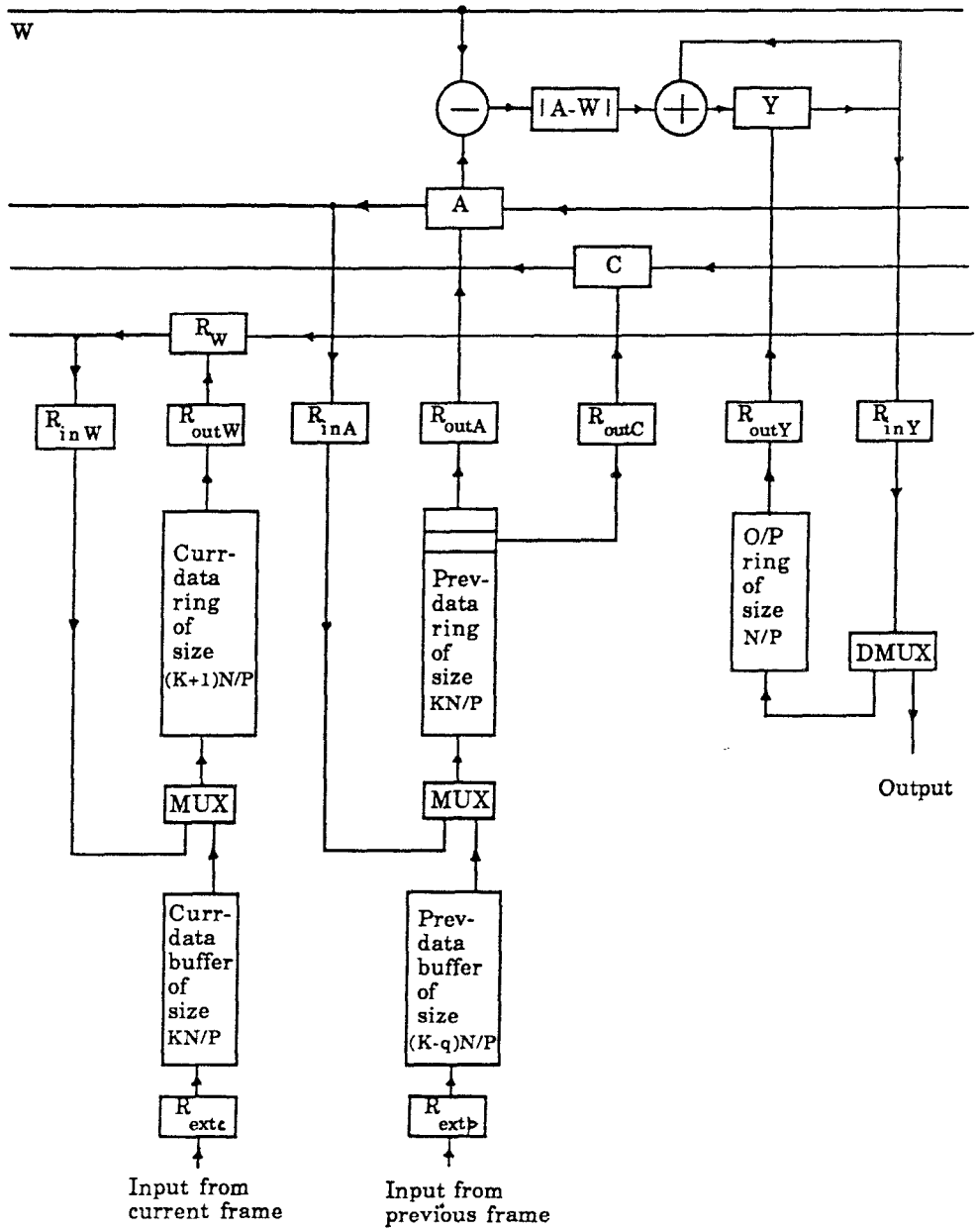


Figure 5.5: Design of a processor of type A for block matching

of the current frame are read only once, and since the rows of the current frame that are required for the computation of $blk.dist(i, i + \Delta i)$ and $blk.dist(i, i + \Delta i + 1)$ overlap by $(K - 1)$ rows. The minimum size of the current-data ring is KN/P. Since the previous-data ring is circulated $(K + 1)\lceil \hat{N}/P \rceil$ times for the computation of $blk.dist(i, i + \Delta i)$, and since it is desirable to clock the current-data ring and the previous-data ring at the same rate, the size of the current-data ring is increased to $(K + 1)\lceil \hat{N}/P \rceil$. The elements of the current-data ring are partitioned into $(K + 1)$ blocks, such that a block contains the $\lceil \hat{N}/P \rceil$ elements which were in the same row in the current frame. The current-data ring has N/P valid data (and 1 invalid data) in each of its K blocks and $\lceil \hat{N}/P \rceil$ invalid data in its $(K + 1)$th block. Figure 5.6 illustrates the data organization in the previous-data ring and current-data ring prior to the computation of $blk.dist(0, -1)$ in example B ((8×8) image, (4×4) reference block, (6×6) search area and 4 processors).

The contents of the current-data ring of the leftmost processor in a group is broadcast locally to all the processors of type A of that group via the W line (see Figure 5.5).

We have seen earlier that each group of processors is responsible for computing displacement vectors of N/P reference blocks of a row. Moreover, each group of processors computes $(q + 1)$ block-distances of $(q + 1)$ candidate blocks for each reference block. Thus for the case when $K > q$, each processor of type A computes N/P block-distances. This means that the size of the output ring should be at least N/P. In the proposed architecture, the size of the output ring is $\lceil \hat{N}/P \rceil = 1 + N/P$ so that the output ring can be clocked at the same rate as the data ring. For the case when $K \leq q$, each processor computes at

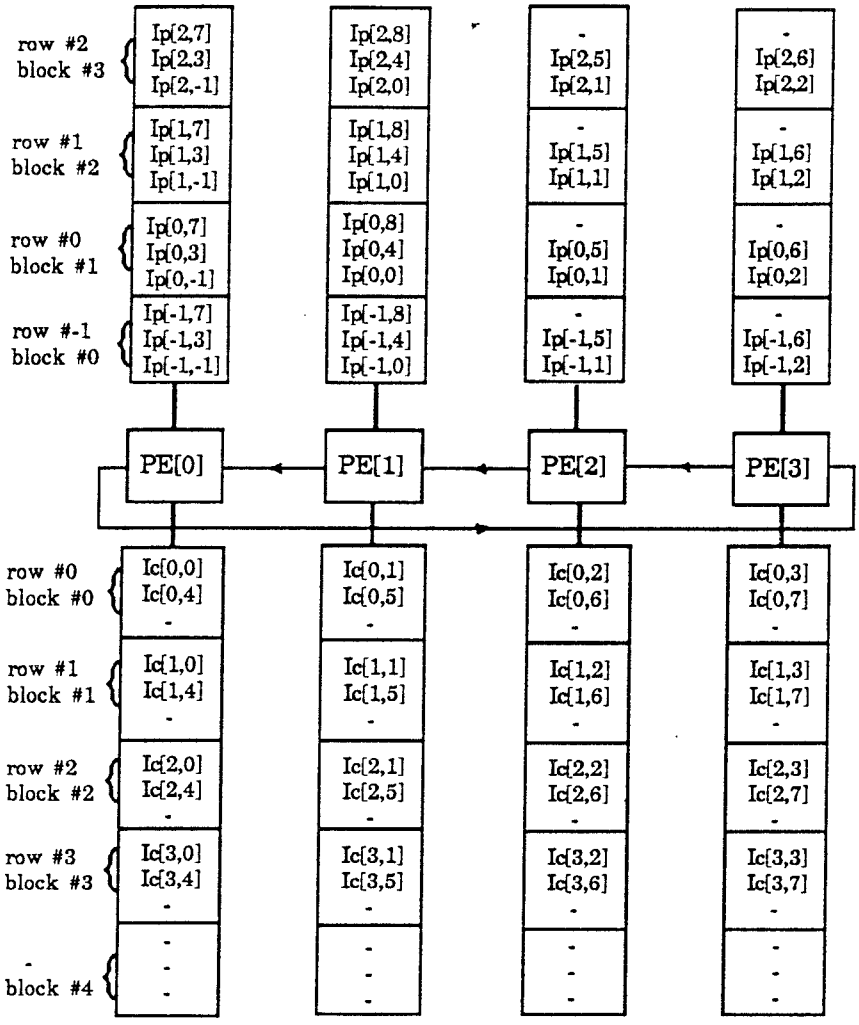


Figure 5.6: Previous-data ring and current-data ring configurations prior to the computation of $blk.dist(0, -1)$ in example B

most $\lceil (q+1)/K \rceil N/P$ block-distances for the N/P reference blocks assigned to it. The size of the output ring in this case is $\lceil (q+1)/K \rceil N/P$. Also, it is clocked at a rate different from that of the data rings.

5.4.2 Processor Algorithm

We next discuss the operations of a processor of type A in the computation of a displacement vector for the case when $K > q$. Recall that for each Δi, the processors compute $blk.dist(i, i + \Delta i)$ (which is a set of $(q + 1)N/K$ block-distances between the N/K reference blocks whose top-leftmost coordinates are in row i of the current frame and their candidate blocks whose top-leftmost coordinates are in row $(i + \Delta i)$ of the previous frame) and then each group of processors updates the temporary displacement vectors corresponding to the N/P reference blocks that are assigned to it.

At the beginning of the computation of $blk.dist(i, i + \Delta i)$, the previous-data ring contains elements of rows $(i + \Delta i)$ through $(i + \Delta i + K - 1)$ of the previous frame in blocks 0 through $(K - 1)$, and the current-data ring contains rows i through $(i + K - 1)$ of the current frame in blocks 0 through $(K - 1)$. In Figure 5.6 the previous-data ring contains elements of rows -1, 0, 1, and 2 of the previous frame and the current-data ring contains elements of rows 0, 1, 2, and 3 prior to the computation of $blk.dist(0, -1)$. Each processor computes the block-difference of one candidate block for each reference block. The procedure consists of summing K row-differences (corresponding to the K rows of a reference block). In fact, processor p computes N/P row-differences between the $(i + m)$th row of the current frame and the $(i + m + \Delta i)$th row of the previous frame. In

particular, it computes $Y_{i,j}[\Delta i, \Delta j, m]$, for each row m of the current block, where $0 \leq m < K$, $p = j \bmod P + \Delta j + \frac{q}{2}$, $0 \leq j < N$ and j is a multiple of K.

The procedure to compute one such row-difference is as follows. The previous-data ring loads two elements into registers A and C such that $A := prevdata.ring[0]$ and $C := prevdata.ring[1]$. The current-data ring loads an element into register R_W such that $R_W := currdata.ring[0]$. The contents of R_W of the leftmost processor in a group is broadcast via the W line. The output ring loads an intermediate block-distance value into register Y. The absolute difference of the contents of register A and W gives one of the terms needed to compute a row-difference. The contents of registers A, C and W are shifted and a new term is computed. The above process is repeated for each of the K entries in a row of the reference block. Note that by shifting the contents of R_W K times, the K elements in a row of the reference block are incident to all the processors of type A of that group. The updated intermediate block-difference value is then loaded back into the output ring. Figure 5.7 illustrates the procedure during the computation of $Y_{0,0}[-1,-1,0]$, $Y_{0,0}[-1,0,0]$, $Y_{0,0}[-1,1,0]$ in example B ((8×8) image, (4×4) reference block, (6×6) search area and 4 processors). See Figure 5.6 for the data ring configurations prior to the computation of $Y_{0,0}[-1,*,0]$. Notice that in this example $PE(3)$ is a processor of type B and so does not participate in the computation of any Y. The series of operations to compute a row-difference are summarized as follows.

	PE(0)	PE(1)	PE(2)	PE(3)
C	Ip[-1,3]	Ip[-1,4]	Ip[-1,5]	Ip[-1,6]
A	Ip[-1,-1]	Ip[-1,0]	Ip[-1,1]	Ip[-1,2]
R	Ic[0,0]	Ic[0,1]	Ic[0,2]	Ic[0,3]
W	Ic[0,0]	Ic[0,0]	Ic[0,0]	_
C	Ip[-1,4]	Ip[-1,5]	Ip[-1,6]	_
A	Ip[-1,0]	Ip[-1,1]	Ip[-1,1]	Ip[-1,3]
R	Ic[0,1]	Ic[0,2]	Ic[0,3]	_
W	Ic[0,1]	Ic[0,1]	Ic[0,1]	_
C	Ip[-1,5]	Ip[-1,6]	_	_
A	Ip[-1,1]	Ip[-1,2]	Ip[-1,3]	Ip[-1,4]
R	Ic[0,2]	Ic[0,3]	_	_
W	Ic[0,2]	Ic[0,2]	Ic[0,2]	_
C	Ip[-1,6]	_	_	_
A	Ip[-1,2]	Ip[-1,3]	Ip[-1,4]	Ip[-1,5]
R	Ic[0,3]	_	_	_
W	Ic[0,3]	Ic[0,3]	Ic[0,3]	_
	$Y_{0,0}$[-1,-1,0]	$Y_{0,0}$[-1,0,0]	$Y_{0,0}$[-1,1,0]	_

Figure 5.7: Contents of registers A, C, R_W during the computation of $Y_{0,0}[-1,*,0]$ in example B

1. load A and C from previous-data ring
   load R_W from current-data ring
   load Y from output ring
2. *for* $t := 0$ to $K - 1$ *do*
3. $\quad Y := Y + |A - W|$ (if $flag = 0$)
4. $\quad$ shift-left A, C, R_W by 1
5. load Y into output ring

The time taken to compute a row-difference is $K\alpha T_c$. Note that $flag$ is set to 1 when invalid data is loaded from the current-data ring. The previous-data ring, current-data ring and the output ring are shifted and the above process is repeated $\lceil \hat{N}/P \rceil$ times. Recall that processor p has to compute N/P block-distances corresponding to N/P reference blocks of a row. The previous-data ring and the current-data ring are clocked at $K\alpha T_c$.

After the computation of $blk.dist(i, i + \Delta i)$, each group of processors updates the temporary displacement vector of the reference blocks assigned to it in the following way. Processors of type A compute the minimum value of block-difference of the $(q+1)$ candidate blocks for each reference block. This is used to update the contents of the *comparator* which stores the minimal value of block-difference along with the corresponding displacement vector. The minimal value of block-difference is computed in parallel with the first set of row-differences of $blk.dist(i, i + \Delta i + 1)$. The value of the comparator is updated in successive iterations of Δi, so that at the end of $(q + 1)$ iterations of Δi, this register contains the minimum value of block-difference and the corresponding displacement vector. Note that each group contains N/P comparators since each group computes the displacement vectors of N/P reference blocks in a row.

5.4.3 I/O operations

The input from the current frame and previous frame external memory are loaded into the current-data ring and previous-data ring in the following way. During the computation of displacement vectors of row i, K rows of both the current frame as well as the previous frame are read in. In fact, rows $(i + K)$ through $(i + 2K - 1)$ of the current frame and rows $(i + K - \frac{q}{2})$ through $(i + 2K - \frac{q}{2} - 1)$ of the previous frame are read in. All K rows of the current frame are stored in the *current-data buffers* (see Figure 5.5) of the P processors. These rows need to be stored for the computation of displacement vectors of row $(i + K)$. While the same number of rows of the previous frame are read in during the same time, only $(K - q)$ rows are stored in the *previous-data buffers* (see Figure 5.5) of the P processors. Out of the K rows of the previous frame that are read in, q rows are required for the computation of displacement vectors of row i, and the remaining $(K - q)$ rows need to be stored for the computation of displacement vectors of row $(i + K)$.

The previous-data ring is circulated $(K+1)\lceil \hat{N}/P \rceil$ times for the computation of $blk.dist(i, i + \Delta i)$, for each Δi. It is circulated $K\lceil \hat{N}/P \rceil$ times so that every element of rows $(i + \Delta i)$ through $(i + \Delta i + K - 1)$ of the previous frame are incident on the processor array and then for another $\lceil \hat{N}/P \rceil$ times so that the elements of row $(i + \Delta i)$ of the previous frame are once again in block 0. In the next step the elements of the $(i+\Delta i+K)$th row are shifted in from the previous-data buffer while the elements of the $(i + \Delta i)$th row are shifted out. At the end of this step the previous-data ring contains elements of rows $(i+\Delta i+1)$ through $(i + \Delta i + K)$ which are required for the computation of $blk.dist(i, i + \Delta i + 1)$. This procedure is repeated q times, once for each Δi, $-\frac{q}{2} \leq \Delta i \leq \frac{q}{2} - 1$. In this

way the elements of q rows (rows $(i+K-\frac{q}{2})$ through $(i+K+\frac{q}{2}-1)$) of the previous frame are loaded into the previous-data ring. In order that the delay between the computation of $blk.dist(i, i+\frac{q}{2})$ (the last set of block-distances of row i) and $blk.dist(i+K, i+K-\frac{q}{2})$ (the first set of block-distances of row $(i+K)$) is minimum, the elements of $(K-q)$ rows are loaded into the previous-data ring from the previous-data buffer. The elements of row $(i+K+\frac{q}{2})$ are loaded at the end of the computation of $blk.dist(i, i+\frac{q}{2})$ and the elements of rows $(i+K+\frac{q}{2}+1)$ through $(i+2K-\frac{q}{2}-1)$ are loaded during the *adjustment time*. At the end of the loading operation, the previous-data ring contains elements of rows $(i+K-\frac{q}{2})$ through $(i+2K-\frac{q}{2}-1)$ which are required for the computation of $blk.dist(i+K, i+K-\frac{q}{2})$. The above process is illustrated in Figure 5.8 in example B ((8×8) image, (4×4) reference block, (6×6) search area and 4 processors). Here rows 3, 4, 5 of the previous frame are loaded during the computation of $blk.dist(0,-1)$, $blk.dist(0,0)$, $blk.dist(0,1)$ respectively, and row 6 is loaded during the adjustment time. The processors do not compute anything during the adjustment time. The adjustment time is for a duration of $(K-q-1)\lceil \hat{N}/P \rceil$.

The elements of the K rows of the current frame are loaded from the current-data buffer into the current-data ring in the following way. The elements of rows $(i+K)$ through $(i+K+q+1)$ are loaded during the computation of $blk.dist(i, i+\frac{q}{2})$ and the elements of rows $(i+K+q+2)$ through $(i+2K-1)$ are loaded during the adjustment time. At the end of the loading operation, the current-data ring contains elements of rows $(i+K)$ through $(i+2K-1)$ which are required for the computation of displacement vectors of row $(i+K)$. In Figure 5.8 rows 4, 5, 6, and 7 of the current frame are loaded during the computation of $blk.dist(0,1)$. The current-data ring is shifted once during the

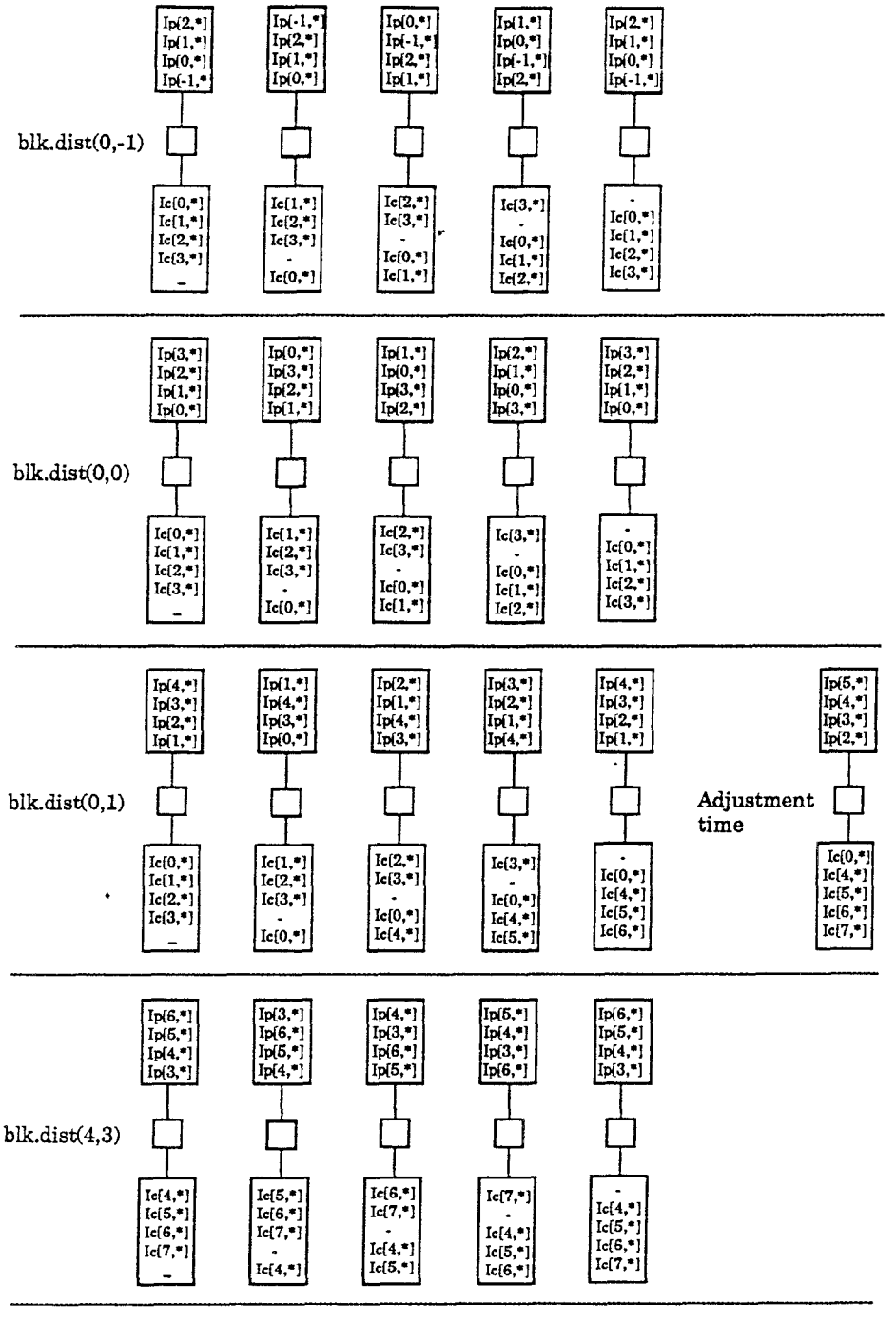


Figure 5.8: Rows of the previous frame and current frame in the data rings of a processor during the computation of $blk.dist(0,-1)$, $blk.dist(0,0)$, $blk.dist(0,1)$, and $blk.dist(4,3)$ in example B

```
begin    {algorithm}
  for i := 0 to N − 1 in multiples of K do begin
    Initialize output ring
    for Δi := −q/2 to q/2 do begin
      for m := 0 to K do begin
        for r := 0 to ⌊N̂/P⌋ do begin
          Y = R_outY; A = R_outA; C = R_outC; R_W = R_outW;
          if (m = 0) then R_inY = Y; Y = 0; update displacement vector;
          Shift previous-data ring, current-data ring, output ring;
          if ((Δi = q/2)and(m > K − q − 2))
            then current-data ring reads from current-data buffer;
            else current-data ring reads from R_inW;
          if (m = K)
            then previous-data ring reads from previous-data buffer;
            else previous-data ring reads from R_inA;
          for t := 0 to K − 1 do begin
            Z := Z + |A − W|    (if flag = 0);
            Shift-left A, C, R_W by 1;
          end    {of t-loop}
        end    {of r-loop}
      end    {of m-loop}
    end    {of Δi-loop}
    for s := 0 to K − q − 2 do begin
      for r := 0 to ⌊N̂/P⌋ do begin
        Previous-data ring reads from previous-data buffer;
        Current-data ring reads from current-data buffer;
      end    {of r-loop}
    end    {of s-loop}
  end    {of i-loop}
end    {algorithm}
```

Figure 5.9: Algorithm for computing block matching when the size of the image is $(N \times N)$, the size of the block is $(K \times K)$, the size of the search area in the previous frame is $(K + q) \times (K + q)$ and the number of processors is P

adjustment time so that block $(K+1)$ contains invalid data.

The time required to compute N/K displacement vectors of a row is $T_{row} = ((q+1)(K+1)+K-q-1)K\alpha T_c\lceil \hat{N}/P \rceil$ and the total computation time $T = (q+2)K\alpha T_c N\lceil \hat{N}/P \rceil$. Let the optimal computation time be T_{opt}. Then $T \approx \frac{K(q+2)}{(q+1)^2}T_{opt}$. For $K > q$, $T > T_{opt}$. Figure 5.9 describes the algorithm for the case when $K > q$.

5.5 Conclusion

In this chapter we have described a semi-systolic architecture consisting of a linear array of P processors to compute block matching when data from both the current frame and previous frame are fed in line scan mode. This input mode necessitates storing part of the data from current and previous frames as well as intermediate results on-chip. The intermediate results need not have been stored, had the data been fed in block scan mode. The on-chip data storage consists of shift registers distributed among the P processors. While shift registers are suitable for regular data flow algorithms like full-search block matching, they are not suitable for algorithms with conditional data flow like hierarchical block matching. RAMS are better suited for such applications.

Chapter 6

Conclusion

In this dissertation, we have studied the problem of developing efficient VLSI architectures for some important real-time signal processing tasks, namely, one-dimensional Discrete Hartley and Discrete Cosine Transforms, multi-dimensional linear separable transforms, template matching and block matching. We chose systolic architectures to compute these tasks since they satisfy the real-time signal processing architectural requirements of design simplicity and regularity, high degree of concurrency, locality and regularity of communication, and balancing of computation with I/O. In the rest of the chapter we give a summary of our contribution and discuss some of the related open problems and possible improvements for each of these tasks.

Discrete Hartley and Discrete Cosine Transforms:

The Discrete Hartley transform (DHT) and the Discrete Cosine transform (DCT) are important transforms in speech encoding, data compression, spectral analysis

and other signal processing schemes. In Chapter 2 we described a systolic architecture for computing these transforms over N points, where N is factorizable into mutually prime factors N_1 and N_2. We mapped the one-dimensional transform over N points into a two-dimensional transform over $(N_1 \times N_2)$ points, and then realized them by two-dimensional systolic arrays. The resulting algorithm consisted of computing the one-dimensional transform over columns and rows of the two-dimensional data array, followed by some adjustments. All the units in our architecture are completely pipelined, resulting in very high throughput.

The number of multipliers in our architecture for computing DCT is $3(f(N_1)+f(N_2))$, where $f(N)$ is the number of multipliers required to compute an N point DFT on real data ($f(N) \approx N$, for small N). It would be interesting to find a decomposition for DCT which is like the Winograd-Fourier or the Winograd-Hartley transform. In that case the number of multipliers would be reduced to $f(N_1) + f(N_2)$.

The algorithm that we developed here can be extended to the case when N is factorizable into (say) d relatively prime factors. The transform over $N = N_1 N_2 \ldots N_d$ points can then be mapped into a d-dimensional transform over $(N_1 \times N_2 \times \ldots \times N_d)$ points by choosing the input and output index mappings appropriately. The resulting algorithm consists of computing the transform over N_1 points along one dimension, followed by computing the transform over N_2 points along another dimension, and so on for all d dimensions. This algorithm can be realized by an architecture consisting of one-dimensional transform computation units and permutation units. The permutation unit is the bottleneck of this architecture. It would be interesting to design an efficient permutation unit which is easily realizable in VLSI.

Multi-dimensional linear separable transforms:

Multi-dimensional transforms have important applications in the areas of spectrum analysis, signal reconstruction, speech processing, tomography, computer vision and image processing. DFT, DHT and DCT are examples of such transforms. We refer to them as DXT. In Chapter 3 we described a family of optimal architectures with area-time trade-offs for computing $(N \times N \times \ldots \times N)$ d-dimensional DXT. The optimality criteria is as defined by VLSI complexity theory. Our architecture consists of one-dimensional DXT(N) computation units which compute DXT(N) over one index, and permutation units which order data so that in the next iteration DXT(N) can be computed over the next index. The architecture has an area $A = O(N^{d+2a})$ and computation time $T = O(dN^{\frac{d}{2}-a}b)$ for all a in the range $\frac{1}{2}\log_N b \leq a \leq \frac{d}{2}$, where $b = O(\log M)$ is the precision, $M = N + 1$.

The architecture that we developed for d-dimensional DXT can be used as a building block for computing d-dimensional convolution. Recall that the convolution of two sequences $x(n)$ and $y(n)$ can be obtained by computing inverse-DFT of the point-by-point product of DFT($x(n)$) and DFT($y(n)$). Thus the architecture for d-dimensional convolution consists of two d-dimensional DFT units, one d-dimensional inverse-DFT unit, and a multiplier unit to compute point-by-point product. The architecture for d-dimensional inverse-DFT is identical to that of d-dimensional DFT except for the value that is embedded in the multipliers of the DFT(N) computation units.

In our architecture we have assumed that all computations involve fixed point arithmetic with $b = O(\log M)$ bits of precision. The fixed point multipliers and

the adders in the DXT(N) computation unit can be replaced by modulo-M multipliers and adders as in [4] without increasing the asymptotic area and time complexities.

There are a couple of issues regarding the actual implementation of our architecture in VLSI. Though our architecture is optimal according to VLSI complexity theory, it is not so from a VLSI designers' point of view. For instance, we use the mesh of trees network to compute one-dimensional DXT(N). Such a network is costly when it comes to actually laying it out in silicon. One can compromise by replacing the mesh of trees network by a systolic array. This would result in a theoretically non-optimal but an easily realizable architecture. Another design issue concerns the layout of the rotator and the subblock rotator units. Though the layout of both these units is very regular, it contains long wires. Though our circuit is synchronous, the delay through the long wires can affect the performance. It may be necessary to incorporate drivers at the expense of a larger delay and increased circuit complexity.

Template matching:

Template matching is a fundamental operation in many window based image processing tasks such as image location, scene matching, edge detection, filtering. In Chapter 4 we described a semi-systolic linear array architecture which handles the high I/O bandwidth requirement of template matching by storing part of the input on-chip in shift registers in each processor, and by circulating the shift registers so that the processor array can compute on the same input multiple times. Our architecture can be easily extended to compute real-time template matching for any problem specification, since the number of processors is a

function of the frame size, the template size and the internal clock rate. Our architecture achieves optimal speed-up.

In our architecture the input is fed in the line-scan mode. Since template matching is a window based operation and since the computations are now split over K lines, the intermediate results have to stored. This is not the case if the input is fed in the block scan mode. However for block scan mode, the entire input image has to be stored in a frame memory.

The template values in our architecture are broadcast to all the processors. If P is large then the delay in broadcasting may affect the performance of the circuit. One way to get around this problem is to store the template values in each processor and to clock them to the multiplier-accumulator. Another alternative is to store a set of template values in a group of processors and locally broadcast to the processors of that group.

Another important issue is what happens when the size of the processor array is so large that the whole array does not fit into one chip. It is interesting to investigate how the processors can be organized into multiple chips such that the inter-chip communication overhead is minimum. Since the shift registers are clocked at a rate K times slower than the inter-processor communication rate, it may be necessary to house the shift registers and the communication units in separate chips. This would be at the expense of an increase in the number of I/O pins.

Block matching:

Block matching is a motion compensation algorithm which is used to remove interframe redundancy in some applications of digital image processing like videophone and teleconferencing. In Chapter 5 we described a semi-systolic linear array architecture which handles the high I/O bandwidth requirement of block matching by storing part of the input from the current frame and the previous frame-memory in shift registers in each processor. This architecture is similar to the architecture that we developed for template matching. Our architecture is versatile since the number of processors is a function of the frame size, the block size and the internal clock rate. The data flow in our architecture is very regular.

As in the case of template matching, the input is fed in the line scan mode and consequently the intermediate results have to be stored in the output ring. The output ring is not necessary if the input is fed in the block scan mode.

We chose shift registers to store data on-chip because they are suitable for regular data flow algorithms like full-search block matching. In hierarchical block matching there is no regularity in the sequence in which data from the previous and the current frames are accessed. For such applications RAMs are better suited than shift registers.

Another interesting problem is how to partition the processor array when it does not fit into a single chip. The processor array then spans over multiple chips. The partitioning should be such that the inter-chip communication overhead is minimum.

Bibliography

[1] N.Ahmed, T.Natarajan, K.R.Rao, "Discrete Cosine Transform," IEEE Trans. on Computers, vol.C-23, 1974, pp.90-93.

[2] R.N.Bracewell, "The Hartley Transform," Oxford University Press, 1986.

[3] R.N.Bracewell, O.Buneman, H.Hao, J.Villasenor, "Fast Two-Dimensional Hartley Transform," Proceedings IEEE, vol.74, no.9, 1976, pp.1282-1283.

[4] G.Bilardi, S.W.Hornick, M.Sarrafzadeh, "Optimal VLSI Architectures for Multidimensional DFT," Proc. Symposium on Parallel Algorithms and Architectures (SPAA), 1989, pp.265-272.

[5] G.Bilardi, M.Sarrafzadeh, "Optimal VLSI circuits for Discrete Fourier Transform," Advances in Computing Research, vol.4, pp.87-101.

[6] R.P.Brent, H.T.Kung, "Area-Time Complexity of Binary Multiplication," Journal of ACM, vol.28, 1981, pp.521-534.

[7] F.Catthoor, H.deMan, "An Efficient Systolic Array for Distance Computation required in a Video-codec based on Motion Estimation," in "Systolic arrays," Adam Hilger, 1987, pp.141-150.

[8] C.Chakrabarti, J.JáJá, "Optimal Systolic Designs for Computing DHT and DCT," IEEE Workshop on VLSI Signal Processing, 1988, pp.411-422.

[9] C.Chakrabarti, J.JáJá, "A Family of Optimal Architectures for Multidimensional Transforms," Proc. 26th Annual Allerton Conf. on Communication, Control and Computing, 1988, pp.1015-1024.

[10] N.U.Chowdary, W.Steenaart, " A high speed two dimensional FFT processor," IEEE Int. Conf. on Acoustics, Speech and Signal Processing, 1984, pp.4.11.1-4.11.4.

[11] W.H.Chen, C.H.Smith, S.C.Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform," IEEE Trans. Comm., vol.COM-25, 1977, pp.1004-1009.

[12] T.C.Chen, M.T.Sun, A.M.Gottlieb, "VLSI Implementation of a 16×16 DCT," Proc. Int. Conf. on Acoustics, Speech and Signal Processing, 1988, pp.1973-1976.

[13] A.M.Chiang, "A Video-Rate CCD Two-Dimensional Cosine Transform Processor", Proc. SPIE vol.845, Visual Communications and Image Processing, 1987, pp.2-5.

[14] N.I.Cho, S.U.Lee, "DCT Algorithms for VLSI Parallel Implementations", IEEE Trans. on Acoustics, Speech, and Signal Processing, vol.ASSP-38, 1990, pp.121-127.

[15] J.F.Coté, C.Collet, D.D.Haule, A.S.Malowany, "A High Performance Convolution Processor," Proc. SPIE vol.1001 Visual Communications and Image Processing, 1988, pp.469-475.

[16] P.Denyer, D.Renshaw, "VLSI Signal Processing: A Bit-Serial Approach", Reading, MA : Addison-Wesley, 1985.

[17] N.Demassieux, F.Jutand, M.Bernard, C.Joanblanq, "A VLSI Architecture for real-time image convolution with large symmetric kernels," Proc. Int. Conf. on Acoustics, Speech, and Signal Processing, 1988, pp.1961-1964.

[18] L.DeVos, M.Stegherr, "Parametrizable VLSI Architectures for the Full-Search Block-Matching Algorithm," IEEE Trans. on Circuits and Systems, vol.36, no.10, 1989, pp.1309-1316.

[19] L.DeVos, M.Stegherr, T.G.Noll, "VLSI Architectures for the Full-Search Block Matching Algorithm," Proc. Int. Conf. on Acoustics, Speech, and Signal Processing, 1989, pp.1687-1690.

[20] R.Dianysian, R.L.Baker, J.L.Salinas, "A VLSI Architecture for Template Matching and Motion Estimation," Presented at Int. Symp. on Circuits and Systems, 1988.

[21] E.R.Dougherty, C.R.Giardina, "Universal Systolic Architecture for Morphological and Convolutional Image Filters," Proc. SPIE vol.1001 Visual Communications and Image Processing, 1988, pp.739-746.

[22] O.Ersoy, "Semisystolic Array Implementation of Circular, Skew circular and Linear Convolutions," IEEE Trans. on Computers, vol.C-34, no.2, 1985, pp.190-196.

[23] D.F.Elliott, K.R.Rao, "Fast Transforms : Algorithms, Analyses, Applications," New York : Academic, 1982.

[24] I.Gertner, M.Shamash, "VLSI Architectures for Multidimensional Fourier Transform Processing," IEEE Trans. on Computers, vol.C-36, no.11, 1987, pp.1265-1274.

[25] H.S.Hou, "The Fast Hartley Transform Algorithm," IEEE Trans. on Computers, vol.C-36, 1987, pp.147-156.

[26] H.S.Hou, "A Fast Recursive Algorithm for Computing the Discrete Cosine Transform," IEEE Trans. Acoustics, Speech, and Signal Processing, vol.ASSP-35, 1987, pp.1455-1461.

[27] F.Jutand, A.Artieri, G.Concordel, N.Demassieux, "VLSI Architecture for Image Filtering," Proc. SPIE vol.1001 Visual Communication and Image Processing, 1988, pp.1108-1115.

[28] F.Jutand, N.Demassieux, A.Artieri, "A new VLSI Architecture for Large Kernel Real-Time Convolution," Proc. Int. Conf. on Acoustics, Speech, and Signal Processing, 1990, pp.921-924.

[29] F.Jutand, N.Demassieux, M.Dana et.al., "A 13.5 MHz. single chip multiformat Discrete Cosine Transform," SPIE vol.845, Visual Communications and Image Processing, 1987, pp.6-12.

[30] T.Komarek, P.Pirsch, "Array Architectures for Block Matching Algorithms," IEEE Trans. on Circuits and Systems, vol.36, no.10, 1989, pp.1301-1308.

[31] R.Kumaresan, P.K.Gupta, "Vector-Radix Algorithm for a 2-D Discrete Hartley Transform," Proc. Int. Conf. on Acoustics, Speech and Signal Processing, 1985, pp.40.6.1-40.6.4.

[32] H.T.Kung, "Why systolic architectures?" IEEE Computer, 1982, pp.37-46.

[33] S.Y.Kung, "VLSI Array Processors," Prentice Hall, 1988.

[34] H.T.Kung, C.E.Leiserson, "Systolic Arrays," Proc. SIAM Sparse Matrix Symposium, 1978, pp.256-282.

[35] H.T.Kung, L.M.Ruane, D.W.L.Yen, "A Two-Level Pipelined Systolic Array for Convolutions," in "VLSI Systems and Computations," Computer Science Press, 1981, pp.255-264.

[36] H.T.Kung, S.W.Song, "A Systolic 2-D Convolution Chip," Tech Rep. CMU-CS-81-110, Carnegie Mellon University, Computer Science Department, March 1981.

[37] H.K.Kwan, T.S.Okulla-Oballa, " Two-dimensional Systolic Arrays for Two-dimensional Convolution," Proc. SPIE vol.1001 Visual Communications and Image Processing, 1988, pp.724-731.

[38] B.G.Lee, "A new Algorithm for the Discrete Cosine Transform," IEEE Trans. Acoustics, Speech, and Signal Processing, vol.ASSP-32, 1984, pp.1243-1245.

[39] B.G.Lee, "Input and Output Index Mappings for a Prime-Factor-Decomposed Computation of Discrete Cosine Transform," IEEE Trans. Acoustics, Speech, and Signal Processing, vol.37, 1989, pp.237-244.

[40] F.T. Leighton, "New lower bound techniques for VLSI," 22nd Annual Symposium on the Foundations of Computer Science, 1981, pp. 1-12.

[41] Y.C.Liao, "VLSI Architecture for Generalized 2-D Convolution," Proc. SPIE vol.1001 Visual Communications and Image Processing, 1988, pp.450-455.

[42] R.F.Lyon, "Two's Complement Pipeline Multipliers," IEEE Trans. on Comm., vol.COM-24, 1976, pp.418-425.

[43] H.Malvar, "Fast Computation of Discrete Cosine Transform through Fast Hartley Transform," Electron. Lett., vol.22, 1986, pp.353-354.

[44] M.Marchesi, G.Orlandi, F.Piazza, "A Systolic Circuit for fast Hartley Transform," Proc. Int. Symp. on Circuits and Systems, 1988, pp.2685-2688.

[45] D.D.Nath, S.N.Maheshwari, P.C.P.Bhatt, "Efficient VLSI Network for Parallel Processing based on Orthogonal Trees," IEEE Trans. on Computers, vol. C-32, 1983, pp.569-581.

[46] M.J.Narasimha, A.M.Peterson, "On the Computation of Discrete Cosine Transform," IEEE Trans. on Comm., vol.COM-26, 1978, pp.934-936.

[47] R.M.Owens, J.JáJá, "A VLSI Chip for the Winograd/Prime Factor Algorithm to compute the Discrete Fourier Transform," IEEE Trans. Acoustics, Speech, and Signal Processing, vol.ASSP-34, 1986, pp.979-989.

[48] F.Piazza, M.Marchesi, G.Orlandi, "A fast DSP circuit based on FHT," Proc. Int. Symp. on Circuits and Systems, 1989, pp.216-219.

[49] F.P.Preparata, J.E.Vuillemin, "The Cube-Connected Cycles: A versatile network for parallel computation," Communication of ACM, vol.24, 1981, pp.300-309.

[50] J.Riordan, "An Introduction to Combinatorial Analysis," John Wiley and Sons, Inc., New York, 1958.

[51] P.A.Ruetz, R.W.Brodersen, "Architectures and Design Techniques for Real-Time Image-Processing IC's," IEEE Journal of Solid State Circuits, vol.SC-22, no.2, 1987, pp.233-250.

[52] H.V.Sorensen, D.L.Jones, C.S.Burrus, M.T.Heideman, "On Computing the Discrete Hartley Transform," IEEE Trans. Acoustics, Speech, and Signal Processing, vol.ASSP-33, no.4., 1985, pp.1231-1238.

[53] M.T.Sun, L.Wu, M.L.Liou, "A Concurrent Architecture for VLSI implementation of Discrete Cosine Transform," IEEE Trans. on Circuits and Systems, vol.CAS-34, 1987, pp.992-994.

[54] C.D.Thompson, "A Complexity Theory for VLSI," Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1980.

[55] U.Totzek, F.Matthiesen, S.Wohlleben, T.G.Noll, "CMOS VLSI implementation of the 2-D DCT with Linear Processor Arrays," Proc. Int. Conf. on Acoustics, Speech and Signal Processing, 1990, pp.937-940.

[56] M.Vetterli, H.J.Nussbaumer, "Simple FFT and DCT algorithms with reduced number of operations," Signal Process., 1984, pp.267-278.

[57] P.P.N.Yang, M.J.Narasimha, "Prime Factor Decomposition of the Discrete Cosine Transform and its Hardware Realization," Proc. Int. Conf. on Acoustics, Speech, and Signal Processing, 1985, pp.20.5.1-20.5.4.

[58] K.Yang, M.Sun, L.Wu, "VLSI Implementation of Motion Compensation Full-Search Block-Matching Algorithm," Proc. SPIE vol.1001 Visual Communications and Image Processing, 1988, pp.892-899.

[59] S. Winograd, "On Computing the Discrete Fourier Transform", Math. Comput., vol.32, 1978, pp.175-195.

[60] M.Yan, J.V.McCanny, Y.Hu, "VLSI Architectures for Digital Image Coding," Proc. Int. Conf. on Acoustics, Speech, and Signal Processing, 1990, pp.913-916.

[61] K.Yang, M.Sun, L.Wu, "A Family of VLSI designs for the Motion Compensation Block-Matching Algorithm," IEEE Trans. on Circuits and Systems, vol.36, no.10, 1989, pp.1317-1325.

[62] C.N.Zhang, "Multidimensional Systolic Networks for Discrete Fourier Transform," Proc. 11th Annual Int. Symp. Computer Architecture, 1984, pp.21-27.