A Double Precision Floating Point Convolution Processor

Jean François Panisset

B. Eng., (McGill University), 1989

Department of Electrical Engineering

McGill University

Montréal

May, 1994

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Engineering

Name _____

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

SUBJECT TERM

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture	0729
Art History	0377
Cinema	0900
Dance	0378
Fine Arts	0357
Information Science	0723
Journalism	0391
Library Science	0399
Mass Communications	0708
Music	0413
Speech Communication	0459
Theater	0465

EDUCATION

General	0515
Administration	0514
Adult and Continuing	0516
Agricultural	0517
Art	0273
Bilingual and Multicultural	0282
Business	0688
Community College	0275
Curriculum and Instruction	0727
Early Childhood	0518
Elementary	0524
Finance	0277
Guidance and Counseling	0519
Health	0680
Higher	0745
History of	0520
Home Economics	0278
Industrial	0521
Language and Literature	0279
Mathematics	0280
Music	0522
Philosophy of	0998
Physical	0523
Psychology	0525
Reading	0535
Religious	0527
Sciences	0714
Secondary	0533
Social Sciences	0534
Sociology of	0340
Special	0529
Teacher Training	0530
Technology	0710
Tests and Measurements	0288
Vocational	0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language	
General	0679
Ancient	0289
Linguistics	0290
Modern	0291
Literature	
General	0401
Classical	0294
Comparative	0295
Medieval	0297
Modern	0298
African	0316
American	0591
Asian	0305
Canadian (English)	0352
Canadian (French)	0355
English	0593
Germanic	0311
Latin American	0312
Middle Eastern	0315
Romance	0313
Slavic and East European	0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy	0422
Religion	
General	0318
Biblical Studies	0321
Clergy	0319
History of	0320
Philosophy of	0322
Theology	0469

SOCIAL SCIENCES

American Studies	0323
Anthropology	
Archaeology	0324
Cultural	0326
Physical	0327
Business Administration	
General	0310
Accounting	0272
Banking	0770
Management	0454
Marketing	0338
Canadian Studies	0385
Economics	
General	0501
Agricultural	0503
Commerce Business	0505
Finance	0508
History	0509
Labor	0510
Theory	0511
Folklore	0358
Geography	0366
Gerontology	0351
History	
General	0578
Ancient	0579
Medieval	0581
Modern	0582
Black	0328
African	0331
Asia, Australia and Oceania	0332
Canadian	0334
European	0335
Latin American	0336
Middle Eastern	0333
United States	0337
History of Science	0585
Law	0398
Political Science	
General	0615
International Law and Relations	0616
Public Administration	0617
Recreation	0814
Social Work	0452
Sociology	
General	0626
Criminology and Penology	0627
Demography	0938
Ethnic and Racial Studies	0631
Individual and Family Studies	0628
Industrial and Labor Relations	0629
Public and Social Welfare	0630
Social Structure and Development	0700
Theory and Methods	0344
Transportation	0709
Urban and Regional Planning	0999
Women's Studies	0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture	
General	0473
Agronomy	0285
Animal Culture and Nutrition	0475
Animal Pathology	0476
Food Science and Technology	0359
Forestry and Wildlife	0478
Plant Culture	0479
Plant Pathology	0480
Plant Physiology	0817
Range Management	0777
Wood Technology	0746
Biology	
General	0306
Anatomy	0287
Biostatistics	0308
Botany	0309
Cell	0379
Ecology	0329
Entomology	0353
Genetics	0369
Limnology	0793
Microbiology	0410
Molecular	0307
Neuroscience	0317
Oceanography	0416
Physiology	0433
Radiation	0821
Veterinary Science	0778
Zoology	0472
Biophysics	
General	0786
Medical	0760

EARTH SCIENCES

Biogeochemistry	0425
Geochemistry	0996
Geodesy	0370
Geology	0372
Geophysics	0373
Hydrology	0388
Mineralogy	0411
Paleobotany	0345
Paleoecology	0426
Paleontology	0418
Paleozoology	0985
Palynology	0427
Physical Geography	0368
Physical Oceanography	0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences	0768
Health Sciences	
General	0566
Audiology	0300
Chemotherapy	0992
Dentistry	0567
Education	0350
Hospital Management	0769
Human Development	0758
Immunology	0982
Medicine and Surgery	0564
Mental Health	0347
Nursing	0569
Nutrition	0570
Obstetrics and Gynecology	0380
Occupational Health and Therapy	0354
Ophthalmology	0381
Pathology	0571
Pharmacology	0419
Pharmacy	0572
Physical Therapy	0382
Public Health	0573
Radiology	0574
Recreation	0575
Speech Pathology	0460
Toxicology	0383
Home Economics	0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry	
General	0485
Agricultural	0749
Analytical	0486
Biochemistry	0487
Inorganic	0488
Nuclear	0738
Organic	0490
Pharmaceutical	0491
Physical	0494
Polymer	0495
Radiation	0754
Mathematics	0405
Physics	
General	0605
Acoustics	0986
Astronomy and Astrophysics	0606
Atmospheric Science	0608
Atomic	0748
Electronics and Electricity	0607
Elementary Particles and High Energy	0798
Fluid and Plasma	0759
Molecular	0609
Nuclear	0610
Optics	0752
Radiation	0756
Solid State	0611
Statistics	0463

Applied Sciences

Applied Mechanics	0346
Computer Science	0984
Engineering	
General	0537
Aerospace	0538
Agricultural	0539
Automotive	0540
Biomedical	0541
Chemical	0542
Civil	0543
Electronics and Electrical	0544
Heat and Thermodynamics	0348
Hydraulic	0545
Industrial	0546
Marine	0547
Materials Science	0794
Mechanical	0548
Metallurgy	0743
Mining	0551
Nuclear	0552
Packaging	0549
Petroleum	0765
Sanitary and Municipal	0554
System Science	0790
Geotechnology	0428
Operations Research	0796
Plastics Technology	0795
Textile Technology	0994

PSYCHOLOGY

General	0621
Behavioral	0384
Clinical	0622
Developmental	0620
Experimental	0623
Industrial	0624
Personality	0625
Physiological	0989
Psychobiology	0349
Psychometrics	0632
Social	0451

Abstract

Two-dimensional convolution is one of the basic operations in image processing, where it is used as a filtering tool. A kernel of values corresponding to the spatial-domain impulse response of the filter is applied to the original image in order to perform desired operations such as low-pass filtering or edge enhancement. A low pass filter will perform image smoothing by removing high-frequency noise, whereas a high-pass filter will enhance the edges: thus can be used to perform low-level feature extraction in a machine vision application. It is also used in most image resampling and warping algorithms: it thus finds applications in both image processing and computer graphics

Since convolution is basically a two-dimensional multiply and accumulate operation, it is computationally intensive. When applying an M by M kernel to an N by N image, $M^2 \times N^2$ multiplications and additions have to be performed. Furthermore, these basic low-level signal-processing methods are frequently applied many times to large data sets, often in real-time. General-purpose computer architectures are often ill-suited to perform two-dimensional convolutions, since they lack the required processing speed or memory bandwidth. This motivated the project to design and build a specialized device which can compute the convolution operation efficiently for such applications.

This thesis addresses the design and implementation of a specialized processor which can perform two-dimensional convolution using double-precision floating-point operands. The selected architecture is based on the concept of the systolic array. These architectures are reviewed particularly for the constraints which impact their logical and physical design, as well as for the numerous applications for which they have been proposed in the litterature or have been implemented. Aftern outlining the overall system architecture of the convolution processor, the

thesis focusses on the details of the implementation of the bus interface and Direct Memory Access controller. Finally, the performance of the proposed design is evaluated and compared against alternative software implementations of the convolution algorithm on representative architectures.

Résumé

La convolution en deux dimensions est une des opérations de base en traitement d'images où elle est utilisée comme outil de filtrage. Un noyau de valeurs correspondant à la réponse impulsionnelle du filtre dans le domaine spatial est appliqué à l'image originale pour effectuer l'opération désirée. Ainsi, un filtre passe-bas permettra d'adoucir une image en enlevant le bruit à hautes fréquences, alors qu'un filtre passe-haut accentuera les arêtes: ceci peut être utilisé pour les premières étapes de l'extraction d'éléments dans un système de vision informatique. Ces méthodes sont également utilisées dans la plupart des algorithmes de ré-échantillonage et de distorsion d'images: ainsi, elles trouvent des applications en traitement et en synthèse d'images.

Puisque la convolution est à la base une opération de multiplication et d'addition en deux dimensions, elle exige une grande puissance de calcul. Pour convoluer une image de N par N points avec un noyau de M par M coefficients, $M^2 \times N^2$ multiplications et additions sont nécéssaires. De plus ces opérations de traitement de signal de bas niveau doivent souvent être utilisée à maintes reprises sur des quantités importantes de données, et ceci souvent en temps réel. Les architectures informatiques d'usage général sont souvent mal adaptées aux contraintes de la convolution en deux dimensions puisque la puissance de calcul et la rapidité d'accès à la mémoire leur font défaut. Il est donc utile de concevoir et bâtir un système spécialisé qui puisse effectuer des convolutions de façon efficace.

Ce mémoire présente la conception et la réalisation d'un processeur spécialisé qui peut effectuer des convolutions en deux dimensions a partir de données en format point-flottant double précision. Le systême est basé sur le principe de l'architecture systolique. Nous effectons d'abord un survol de ces architectures en s'attardant aux contraintes qui affectent leur conception logique et physique,

ainsi qu'aux nombreuses applications proposées dans les publications. Apres la présentation de l'architecture générale du système suivent les détails de la réalisation de l'interface au bus et le contrôlleur pour l'accès direct à la mémoire (DMA). Enfin, les performances du système sont évaluées et comparées à des réalisation. logicielles de l'algorithme de convolution sur des architectures représentatives

Acknowledgements

I wish to thank my supervisor, Dr. A.S. Malowany for his patience and support throughout the sometimes tortuous process of this degree. I also thank CAE Electronics, my current employer, for providing an opportunity to prove that this knowledge was not acquired in vain and for furnishing access to several of the machines on which the convolution benchmark programs were run. This work has proved to be very interesting which partly explains why it was not finished sooner. Finally, the financial support of NSERC is acknowledged.

Table of Contents

List of Figures

List of Tables

Chapter 1 Systolic Arrays

1.1 Introduction

In this chapter, systolic arrays are examined as a solution to computationally intensive problems. First, the characteristics of a systolic architecture are described. Then, methods are presented for mapping a problem, usually described by a sequential algorithm, into a parallel systolic system This is followed by a look at the issues which face the hardware designer when it comes time to design actual hardware from a systolic algorithm description. Originally proposed as a solution for matrix computations, systolic arrays have been used to solve a wide variety of problems in diverse fields. Although they are still considered somewhat of a research-oriented approach, systolic architectures have nonetheless been implemented in actual hardware in a number of systems, using either custom or off-the-shelf components.

1.2 What is a Systolic Array ?

The term *systolic array* was first used by H.T. Kung and C.E. Leiserson in [Kung and Leiserson, 1979] to describe a new kind of parallel architecture. A systolic array is composed of a grid of interconnected processors which work together to solve a problem faster than a single processor. But the main characteristic of these computational structures comes from the "systolic" part, which means that pipelined computations are performed along all dimensions of the array structure. Data which is read into the array travels (possibly with intermediary results) from processor to processor, thus achieving high computation rates without requiring

correspondingly high Input/Output bandwidth [Fortes and Wah, 1987].

The adjective systolic was used to describe these structures in analogy to the human circulatory system, where at each heartbeat (clock cycle), the heart (the source and destination of data) pumps a small quantity of blood (data) into a network of arteries and veins (the array of processing elements) Another possible analogy for the word is that many of the early systems described as "systolic" alternated between cycles of admission and expulsion of data, which is similar to the way blood flows into and out of the heart.

Systolic architectures are also characterized by regular structures where all the processing elements are similar to each other, except perhaps for boundary elements. Furthermore, the interconnections between the PEs tend to be simple and straightforward. Systolic array PEs are thus a prime candidate for VLSI implementation, where intra-chip bandwidth is very high but inter-chip connections are much more expensive (both in pin count and speed). It is also possible to build scalable systems, where the array can be made progressively larger (and thus able to solve large problems in fewer iterations) by adding extra chips/processing modules

1.3 Algorithm Issues and Software Tools for Systolic Arrays

As it is always the case with parallel architectures, the main challenge often comes in the mapping of an algorithm into the desired parallel structure. Some algorithms are "embarrassingly parallel" and map readily, others require more work. This section will look at systematic methods which have been developed to derive systolic arrays from problem specifications. Some methods are general and can be applied to a wide class of algorithms, others are more specific. Since there are different types of systolic architectures, some methods have been proposed which are oriented towards specific types of systolic arrays. The ultimate goal is

to develop software tools and/or programming languages which would allow the designer to specify the problem in a "natural" form (which is often a sequential algorithm) and derive the corresponding systolic array automatically.

1.3.1 General Mapping Methods

As outlined in section 1.2, one of the principal characteristics of systolic arrays is local communication between the processing elements, often limited to their respective nearest neighbors. In particular, this has the advantages of simplifying VLSI implementation and signal routing on a printed circuit board. Unfortunately, many algorithms contain "broadcast" data dependencies where data needs to be shared between multiple PEs which are not connected to each other. Wong and Delosme derive in [Wong and Delosme, 1988] and [Wong and Delosme, 1992] a method where any such broadcast can be transformed into propagations along the normal connection paths of the systolic array.

Moreno and Lang have developed a method based on the transformation of the dependency graph of the algorithm [Moreno and Lang, 1988] [Moreno and Lang, 1990] called the multi-mesh graph method. The first step is to remove from the graph properties which are incompatible with a systolic implementation such as broadcasts and bi-directional data flows. The graph is then converted into a *G-graph* by collapsing groups of nodes into new nodes (*G-nodes*), which is more suitable to partitioning. Finally, the G-nodes are mapped into an array with m cells by scheduling sets of m neighbor G-nodes (a *G-set*) for concurrent computation. They show how their method can be applied to the transitive closure problem.

Others have attempted to create a formal framework in which to describe and understand the mapping process. For instance, Payer uses the theory of finite state machines to start from a functional description and achieve a systolic array in a for-

mal way [Payer, 1988]. He demonstrates his method on two classes of problems: bit pattern matching and FIR filtering. Bertolazzi, Guerra and Salza propose a method based on the analysis of the data dependencies of the original algorithm and extend it to include the design of non-regular systolic arrays [Bertolazzi *et al.*, 1988]. They apply their method to create systolic arrays to perform 2-dimensional convolutions and solve the shortest path problem on layered graphs. Another formal approach is suggested by Ko and Wing where they formulate the problem and its implementation in an n-dimensional space of integers which allows the implementation to be derived from the algorithm by linear transformation [Ko and Wing, 1988].

Systematic methods for designing systolic arrays lose some of their interest if they result in non-optimal designs (especially if more ad-hoc heuristics are able to do better!). Kothari, Oh and Gannett propose a method which can produce optimal designs for systolic architectures with linear scheduling functions [Kothari *et al.*, 1989]. Their method is based on a combination of linear algebra and a heuristic which exploits special properties of convex sets. This allows them to derive a different method for performing convolutions. Clauss, Mongenet and Perrin are interested in mapping systolic algorithms onto the smallest possible number of processors in a general processor array [Clauss *et al.*, 1990]. They derive two space-optimal mappings for the gaussian eliminination method for solving systems of linear equations. Finally, Zhong and Rajopadhye show how neighboring processors in a systolic array obtained via conventional linear transformation methods can be merged together to obtain fully efficient arrays [Zhong and Rajopadhye, 1991].

1.3.2 Mapping Methods for Specific Algorithm Classes

Numerically-intensive algorithms often spend most of their time in relatively small nested loops, which are thus a prime candidate for parallelization. Much work has been done on the analysis of data dependencies within such loops in order to determine which iterations of the loop can be performed in parallel without

violating these dependencies [Banerjee, 1988] [Wolfe, 1989]. The driving force behind this has been the need for optimizing compilers for "traditional" vector supercomputers, and the success of these methods has generally been judged on the basis of how a compiler manages to parallelize and/or vectorize nested loops in FORTRAN programs. Similar work has also been done to map nested loops to systolic architectures. For instance, Lee and Kedem have derived a method for mapping p-nested *for* loops onto q-dimensional systolic arrays, where $1 \leq q \leq p-1$ [Lee and Kedem, 1989]. Similarly, Bu, Deprettere and Thiele derive a method for mapping nested loop programs where the loop boundaries are allowed to be functions of the previous index variables [Bu *et al.*, 1990b].

Many algorithms can be expressed in terms of systems of linear recurrence equations, which can then be mapped onto systolic arrays. The computations performed by the algorithm can be represented as integral points in some domain of the Euclidian space, and are ordered by means of a linear schedule which must respect the data dependencies between them. In the case of uniform linear recurrences, the dependencies are only local: such problems can be readily mapped onto systolic arrays. Unfortunately, many problems contain global dependencies. Van Dongen and Quinton present a method to transform these non-uniform linear recurrence systems into uniform systems, which can then be mapped directly to systolic arrays [Van Dongen and Quinton, 1988]. Yaacoby and Cappello have approached a subclass of these problems, namely affine recurrence equations and have derived necessary and sufficient conditions for the existence of a schedule which satisfies these problems [Yaacoby and Cappello, 1988].

Steenaart and Zhang take a different approach for the class of recursive algorithms, and derive a methodology for mapping such problems directly onto systolic arrays [Steenarrt and Zhang, 1991]. They are especially interested in recursive filtering algorithms (such as implementations of IIR filters) where the inputs are dependent on the previous outputs.

1.3.3 Mapping to Specific Architectures

Most of the mapping methods seen in the previous sections attempt to solve the problem of designing a systolic array which accurately executes a given algorithm. But in many cases, this mapping cannot be the only constraint on the design of the array, and other factors must be taken into account. For instance, it might be desired to map an algorithm onto an existing, general-purpose array which cannot be tailored exactly to our needs. Also, the size of the array resulting from the optimal solution of a problem might not be practical: thought must be given to partitioning the algorithm onto a fixed-size array. Moldovan and Fortes have proposed a technique which can be used to partition nested loops by dividing the index space of the problem into bands and to map these bands onto the space of the processor array [Moldovan and Fortes, 1986]. Unfortunately, their method cannot deal with nested loops where the iteration bounds are themselves functions of the outer-loop indices. Bu, Deprettere and Dewilde approach the problem in a different way: instead of trying to map an algorithm directly into a fixed-sized array, the problem is first mapped to the "optimal" sized array which is then reduced to the fixed-size array by clustering processing elements [Bu *et al.*, 1990a]. They propose two clustering methods, which have the additional advantages of raising the efficiency of inefficient arrays, balancing local memory and external communications for the processing elements and reducing array dimensionality (the more restrictive case of mapping two-dimensional arrays onto uni-dimensional ones was previously studied by Kumar and Tsai [Kumar and Trai, 1988]).

Other researchers have looked at the problem of mapping systolic algorithms onto specific architectures. For instance, Lin shows how shuffle arrays can be used to implement systolic algorithms [Lin, 1988]. A shuffle array is an array of processing elements interconnected by a shuffle bus. An N-node shuffle bus consists of a master array MA and a slave array SA. Each element of the master array contains a single bit, whereas the slave array contains a k-bit data word. When instructed, any pattern of a 1 followed by a 0 in the master array will cause

the contents of these bits, along with the corresponding registers of the slave array to be swapped. A shuffle array can be configured as a 1-D or 2-D queue, and can also be used for sorting.

Hypercubes can be considered to be generalizations of N dimensional arrays. Ibarra and Sohn show how one-way and two-way linear systolic arrays (i.e. arrays where the processors are connected only in one direction with uni-directional communication paths) can be mapped onto a 64-node NCUBE/7 MIMD hypercube machine [Ibarra and Sohn, 1989]. They used this method to implement 1D FIR filters, matrix multiplication and solve transitive closure problems. The main challenge in this case is to efficiently map the systolic connections onto the structure of the hypercube. Another example of mapping strategies specific to a hardware implementation can be found in the work of Valero-Garcia *et al* , who tackle the issues associated with the use of pipelined functional units as processing elements in systolic arrays [Valero-García *et al.*, 1990]. More specifically, their mapping method improves the efficiency of the array by inserting delays into the data flow between processing elements to take into account the pipelining delays internal to the array elements.

1.3.4 Compilers and Tools

In order to make systolic arrays a truly practical concept and not just an academic curiosity, software tools have to be made available for VLSI designers who want to use this design methodology. Some work has been done to provide tools for specific applications. For instance, Hu, McCanny and Yan have developed a system for designing systolic vector quantization chips for speech and image coding applications [Hu *et al.*, 1990]. Their system consists of a library of cells, silicon assemblers, simulators, test pattern generators and a graphical user interface. Another specific application is the Logic Description Generator which is used to implement systolic algorithms on the SPLASH reconfigurable logic array [Gokhale *et al.*, 1990]. The

LDG accepts as input a programming language which describes the functionality of the cells in the systolic array. Its output is a Xilinx Netlist Format (XNF) file which is fed to the Xilinx design tools which will generate the bit patterns to be downloaded into the Xilinx FPGA devices which make up SPLASH. Due to the very rapid turn-around time possible with this system, most debugging is done directly on the target hardware. Another example of a system-specific tool is the AL programming language for the CMU Warp programmable systolic array [Tseng, 1990] AL is a C-like language where scalar and array objects are duplicated in all the cells whereas distributed array (DARRAY) objects are distributed among cells. The DO statement tells the compiler to distribute loop iterations over the cells instead of duplicating their execution. Using this language to implement matrix computations, 27% of the peak performance of the machine was achieved for matrices of order 300 (which illustrates the problems which can be encountered in using programmable parallel machines efficiently).

More general tools have also been developed. DECOMPOSER is a high-level synthesis tool which takes as inputs a hierarchical description of the computation to be performed and hints as to how it must be performed [Hou *et al.*, 1988]. This description takes the form of a directed acyclic graph (DAG). The output of the system gives the required structure of each processing element, their interconnections and the input and output sequences. The SYSTARS system is capable of performing both analysis and synthesis of systolic arrays [Omtzigt, 1988]. SYSTARS is able to generate both full-size and partitioned systolic arrays. It also includes a graphics display which can animate the structure being designed, which helps the designer visualize the flow of data in the system.

Actual compilers which generate code to implement algorithms have also been proposed. Omtzigt describes the architecture of such a compiler which can handle systems of affine recurrence equations based on the domain flow model [Omtzigt, 1990]. The domain flow graph is an extension of the data flow graph where nodes represent functions (either scalar functions or control flow) or

they can be dependence graphs representing concurrent operators. The input to the compiler is a C language program with extensions (called Domain C). Lengauer, Barnett and Hudson have developed a system-independent compiler which can handle both imperative and functional programs, including non-uniform linear recurrence problems [Lengauer *et al*, 1991]. The output of the compiler is a program in the native language of the target system. Examples of mappings of matrix algorithms to the CMU Warp machine and Occam-based transputer networks are shown

1.4 Hardware Issues for Systolic Arrays

One of the main justifications for the systolic design methodology is the ease with which such designs can be implemented in VLSI. Nevertheless, the physical implementation of systolic arrays poses certain particular challenges, some of which are examined in this section.

1.4.1 Synchronization and Clocking

Systolic arrays are typically structured as synchronous SIMD arrays where all the processing elements execute the same instructions under control of a central clock. While this simplifies the transfer of data between processors and removes the need for synchronizing First-In First-Out (FIFO) memories between them, clock distribution and synchronization can become problematic for large arrays and high clock rates. For one dimensional arrays, Fisher and Kung demonstrate that it is possible to use a pipelined clocking scheme where more than one clock event is propagated at a time [Fisher and Kung, 1985] Although clock skew will occur between processor elements, an upper bound for this skewing can be derived between two adjacent processors and thus correct operation can be ensured (a

probabilistic model which can derive an upper bound for the accumulation of clock skew in synchronous systems is presented in [Kugelmas, 1988]). Unfortunately, this result does not generalize to two-dimensional arrays, where a mixed scheme using clocking and asynchronous elements is used at the expense of additional hardware complexity

One possible solution is to go to a purely asynchronous model based on the concept of the *data flow* machine [Dennis, 1980]. In a data flow computer, an execution unit performs its computation as soon as it has received all of its operands and sends the result on to the unit connected to its output, which in turn "fires" when it has received all of its inputs. The machine is thus self-synchronizing and does not require any global clocking since synchronization occurs implicitly through the detection of inputs This concept can be applied to systolic arrays, which are then usually known as wavefront arrays [Kung *et al.*, 1987] (the term wavefront comes from the analogy of a wave of calculations propagating through the array). Although attractive from a synchronization standpoint, wavefront arrays do require more hardware since buffers must be interposed between the outputs and inputs of processing elements. Furthermore, unless the structure of the array is completely regular, care must be taken to ensure the efficiency of the system (i.e. no single processor must become a bottleneck as it waits for one of its inputs, and thus stalls the output of the processors connected to all of its other inputs· sufficiently deep FIFO memories must be used to prevent this). Finally, as in all asynchronous systems, care must be taken not to fall prey to glitches and parasitic noise which might be generated by surrounding components toggling asynchronously.

1.4.2 Reliability and Fault Tolerance

The designer of any parallel computer system must worry about reliability and fault tolerance, since a large number of processing elements are much more likely

to fail than a single one. A fault-tolerant system must include mechanisms for detecting when errors have occurred, as well as mechanisms for dealing with these errors and ensuring continued operation of the system even in the presence of faulty components. Error detection can be achieved simply by duplicating functional modules and comparing the outputs of two or more units performing the same computations: any discrepancies will indicate a failure in one of the units involved. Several other methods specific to systolic arrays have also been proposed and can be found in [Abraham *et al.*, 1987]. For instance, in a systolic array where not all of the elements are always active, idle elements can be used to duplicate computations and thus provide partial redundancy testing

Instead of duplicating hardware to provide space redundancy, time redundancy can be used where the throughput of the array is kept below its maximum rate and some of the extra time is used for error detection and correction. Antola *et al* show how space and time redundancy can be combined to yield cost-effective fault-tolerant structures in the specific case of arrays used to compute Fast Fourier Transforms [Antola *et al.*, 1988].

Another possibility is to build the fault tolerance into the algorithms implemented by the systolic array [Anfinson, 1988]. For instance, special coding schemes can be used to detect and correct single- or multi-bit errors in computations without having to completely duplicate the functionality of the processing elements (which would be prohibitively expensive in all but the most demanding applications). Bandyopadhyay, Jullien and Sengupta used the residue number system (RNS) to design a systolic array for multi-operand residue addition which can detect and correct errors [Bandyopadhyay *et al.*, 1988].

Although on-line reliability is important, off-line testability is also crucial, and complex systems must be designed to be efficiently and completely testable. Sciuto and Lombardi demonstrate the required conditions to test two-dimensional bilateral arrays (i.e. where data is allowed to flow in both directions between pro-

cessor elements) [Sciuto and Lombardi, 1988]. Kim approaches the more restrictive problem of one-dimensional linear arrays (uni- and bi- directional), with special emphasis on the capability of the array to be reconfigured to bypass module failures without impacting the designed throughput of the system [Kim, 1988]. Array reconfigurability will be further discussed in the following section.

1.4.3 Reconfigurability

Reconfigurability in an array processor can be used both to allow different functionality in a general-purpose system as well as to work around any faulty components which are detected either offline or online. An early reconfigurable array processor named CHiP (for Configurable, Highly Parallel) is described by Snyder in [Snyder, 1982]. CHiP is composed of a collection of homogeneous microprocessors, a switch lattice and a controller. The PEs are connected at regular intervals to the switch lattice, which itself can be configured to connect the PEs together in many different ways. It is thus possible to implement different interconnection schemes, as well as to isolate malfunctioning PEs.

Poli and Bayoumi propose a structure similar to that of CHiP for implementation on a single VLSI device [Poli and Bayoumi, 1988]. The ability to reconfigure on-line the array to work around *transient* problems with a particular PE increases the fault-tolerance of the entire system, whereas the off-line reconfiguration of the array to alleviate a *permanent* PE failure greatly increases the yield of the VLSI device (thus decreasing its cost). Youn and Singh propose a design which can efficiently reconfigure both tree and rectangular structures [Youn and Singh, 1988]. Their main concern is to minimize the extra delay introduced by the reconfiguration path. Their approach is also able to handle clustered defective processing elements (since faults are often not uniformly distributed across the surface of a die or wafer).

Sha and Steiglitz formulate the problem of array reconfigurability in terms of graph theory and derive a lower bound on the time complexity of any reconfiguration algorithm [Sha and Steiglitz, 1991]. Codenotti and Tamassia on the other hand use a network flow model of the virtual fault-free array composed of the functional cells of the partially defective array [Codenotti and Tamassia, 1991]. A survey of reconfiguration methods for array processors can be found in [Chean and Fortes, 1990].

1.5 Systolic Array Applications

Although systolic arrays were originally proposed by Kung for matrix computations [Mead and Conway, 1980], they have since been used to solve problems in a number of diverse fields. This section looks at a number of such applications, with an emphasis on numerical and signal processing problems.

1.5.1 Matrix Computations

Matrix computations are a natural fit for parallel implementations since they usually make use of fairly simple operations repeated very often. Furthermore, source operands are often used several times, thus making high demands on memory bandwidth. Thus it is hardly surprising that systolic architectures have been suggested to solve a number of matrix algorithms. In [Mead and Conway, 1980], Kung suggests systolic structures for performing matrix-vector inner products, matrix-matrix multiplications and linear system solving using LU decomposition. Another approach to matrix-matrix multiplication is presented in [Peng and Jun, 1988], where a systolic array of m^2 processing elements is used to multiply two m by m arrays in time $\frac{5}{2}m - 1$.

The solution of large systems of linear equations is a problem which comes up

frequently in scientific computing, and it has also been approached using systolic arrays. In [Benaini and Robert, 1990], $\frac{n^2}{4} + O(n)$ processors are used to perform Gaussian elimination on an n^2 problem using time $3n - 1$. Using instead LU decomposition, Wan proposes in [Wan and Evans, 1993] an architecture which can solve the problem $AX = B$ where A is an $n \times n$ matrix, X is an $n \times p$ matrix and B is also $n \times p$ using an array of $np + \frac{n(n+1)}{2}$ processing elements in time $4n + p - 2$ for the first system, $2n$ for each additional system (thus making this structure ideal for a pipelined system) The same array can also be used to compute the inverse of a $n \times n$ matrix in time $5n - 2$. In many scientific applications such as finite element analysis, linear system are very sparse and thus require special solutions in order to achieve high performance. Tseng implemented [Tseng, 1988] a general sparse linear system solver using the incomplete Choleski pre-conditioned conjugate gradient method on the Warp systolic computer [Annaratone *et al.*, 1987].

Another computationally intensive matrix operation which can be solved using systolic arrays is the extraction of eigenvalues/eigenvectors. Although the general QR-decomposition method is not very suitable for parallel implementation, it can be useful in the case of symmetric tridiagonal matrices [Phillips and Robertson, 1988]. Here, an $m \times (n + 1)$ systolic array is able to extract the eigenvalues and eigenvectors of an $n \times n$ symmetric tridiagonal matrix in time $2m + 2n - 1$, with much greater savings if pipelined results are needed. Another popular method is the Jacobi algorithm: systolic arrays for computing eigenvalues/eigenvectors using this method are presented in [Delosme, 1990] and [Lam, 1991].

Linear Least Squares problems are frequently encounted in signal-processing applications. These consist in computing the vector x which minimizes $\|Ax - b\|$. Systolic methods for the solution of this problem are proposed in [Chen and Yao, 1988] and [Torralba and Navarro, 1988], which are both based on QR decomposition. Moonen [Moonen and Vandewalle, 1993] proposes a method to solve the Recursive Least Square (RLS) problem, which consists in recomputing

the least squares solution after appending new data by making use of the results from the previous step. When the effects of finite-precision arithmetic are taken into consideration, some methods yield better results: for instance, Liu [Liu *et al.*, 1990] presents an architecture which performs the Systolic Block Householder Transformation in order to ompute the RLS algorithm. A version of this architecture which can handle complex numbers is presented in [Tai.g *et al.*, 1991]. Liu also proposes a systolic solution to the same problem using the Givens rotation [Liu *et al.*, 1991]

1.5.2 Transform Methods

Systolic arrays have been used to efficiently implement transformation operations. This is a natural application which was also first proposed by Kung in [Mead and Conway, 1980]. He remarks than "an n-point discrete Fourier transform is the matrix-vector multiplication, where the (i, j) entry of the matrix is $\omega^{(i-1)(j-1)}$ and ω is a primitive nth root of unity". Thus the same structures proposed for matrix operations (extended to operate on complex numbers) can be used to compute an n point DFT in $O(n)$ time, as opposed to the $O(n \log n)$ operations required for the FFT algorithm implemented on a sequential processor. Kung then proposes how the roots of unity can be generated internally to the array if each array processing element has at its disposition an extra register: this method decreases the connectivity requirements for each PE Kar proposes in [Kar and Bapeswara Rao, 1993] a scheme which can reduce almost by half the number of multipliers required to compute the DFT algorithm. This is a significant savings since multipliers tend to take up a large amount of silicon real-estate, which is crucial when considering the implementation of a systolic array as a VLSI device. By rewriting the DFT in a recursive form, only $2n + 2$ multipliers are required instead of $4n$, and the cycle time can be reduced from $(t_m + 2t_a)$ to $(t_m + t_a)$ where t_m is the time required to perform a multiplication and t_a an addition.

The Fast Fourier Transform is an algorithm which is used to shorten the amount

of time required to compute an n point DFT on a sequential processor from $O(n^2)$ time to $O(n \log n)$ time It can also be implemented using a systolic array [Choi and Boriakoff, 1992], where it has the advantage of lowering the required number of processing elements from n to $\log n$ (with the addition of some slight overhead, namely $n \log n$ simple single-stage shift registers). Furthermore, this circuit can produce two results at each clock cycle, and does not require ROMs for storing the roots of unity (this can be a factor when implementing such circuits as gate arrays: look-up tables and other such storage elements tend to be very expensive in terms of gate count, and it is usually not practical to go off-chip to access an external memory, in contrast to older designs based on discrete parts where a single ROM lookup table could save considerable amounts of circuitry). In [Johnsson *et al.*, 1988], Johnsson shows how a systolic FFT algorithm can be implemented on a boolean n-cube machine such as the Connection Machine model CM-2. This method makes use of the high storage bandwidth within a node, and is optimized for the communication patterns between the nodes. For a $P = 2^p$ point Decimation in Frequency FFT executed on $N = 2^n$ processors, the first $p - n$ steps are executed locally on each processor and the last $\log_2 N$ steps require interprocessor communication. For a Decimation in Time algorithm, these are reversed. This is made possible by storing $\frac{P}{2N} + nN$ twiddle factors for $\frac{P}{N}$ elements stored in each of the N processors.

Image-processing applications will often require the computation of two-dimensional Discrete Fourier Transforms, since a number of image filtering algorithms can be implemented as simple masking operations in the frequency domain. Sarkar [Sarkar and Majumdar, 1991] presents an architecture which uses two linear arrays of $\sqrt{N}$ processing elements to compute the $\sqrt{N} \times \sqrt{N}$ 2-D DFT in time $O(N)$. The first array of $\sqrt{N}$ processors is used to compute the DFT of the rows, and the second array is used to compute the DFT of the columns. An extra processor is required to generate the roots of unity. All of the PEs are used 100% of the time. The speedup in the computation time over the single processor case is

$\frac{(2\sqrt{N})}{N} = 2\sqrt{N}$, which means that it achieves an optimal linear speedup of $2\sqrt{N}$ using $2\sqrt{N}$ processors. Sarkar also proposes an Instruction Systolic Array to compute the 2D FFT [Sarkar *et al.*, 1991] An ISA is a systolic array where instead of letting the data flow through the array from PE to PE, the data remains stationary whereas instructions flow rhythmically from PE to PE at each clock cycle. For a $\sqrt{N} \times \sqrt{N}$ point FFT, this design uses N processing elements and can complete the operation in time $O(\sqrt{N})$.

A different approach to the problem which uses coordinate rotation digital computer (CORDIC) PEs is proposed by Jones [Jones, 1993]. The conventional DFT sum is first expressed recursively using Horner's rule. Note that multiplication of the input values by the powers of the roots of unity is equivalent to simple phase rotations which can be implemented using the CORDIC algorithm The main advantage of this method is that it does not require multipliers, which are instead replaced by additions and shifts (these are usually more space-efficient in VLSI designs). Based on this method, the FFT of a 2D signal with $N = N_1 N_2$ points can be computed in $O(N)$ time using $N/4$ bit-serial PEs.

Systolic arrays have also been used to compute other transform operations. For instance, Hellwagner proposes in [Hellwagner, 1988] an architecture to perform the one-dimensional Generalized Fourier Transform: this means that the array can be configured to compute a wide class of discrete linear transform including the Walsh-Hadamard and Discrete Fourier Transforms. Another transform method which is useful in signal processing applications is the Discrete Hartley Transform (DHT): its main advantage is that it requires real number arithmetic only as opposed to the complex number arithmetic required by Fourier methods Using a linear systolic array of CORDIC processors, the system presented by Meher, Satapathy and Panda [Meher *et al.*, 1993] can compute the recursive DHT algorithm for a $4N$ sequence in time N using $\frac{(N+1)}{2}$ processing elements. Chakrabarti and JáJá propose a bit-serial systolic array which can be used to compute two-dimensional Discrete Hartley and Discrete Cosine Transforms [Chakrabarti and JáJá, 1990] (the DCT is

also a real-only transform which is used among other applications as the basis for the JPEG image compression algorithm [Wallace, 1991]). This architecture can compute the DHT or the DCT of a $N = N_1 \times N_2$ array of p-bit operands in time $O(p(N_1 + N_2))$, which optimally corresponds to the rate at which the input operands can be shifted into the bit-serial PEs.

1.5.3 Convolution Methods

Whereas the transform methods of the previous section operate on a signal in the frequency domain, it is also possible to operate directly on the signal itself using the time (or spatial) domain representation of the desired filter. For discrete signals, convolution is basically a multiply-and-accumulate array operation: thus it is a natural candidate for systolic implementation. In [Kung, 1982], Kung presents a number of possible alternatives for the design of one-dimensional convolution arrays (there are many possible alternatives, which relate to whether the source data or the filter coefficients are stationary, as well as how partial results are communicated between the processing elements). An implementation of a one-dimensional systolic convolution device from Harris Corporation is shown in [Chester *et al.*, 1991]. Since convolution arrays are usually used to implement linear phase Finite Impulse Response (FIR)filters, the inherent symmetry of the coefficients of these filters can be exploited to reduce the number of required multiplications: Kwan proposes an architecture which exploits these properties [Kwan, 1993].

In [Kwan and Okullo-Oballa, 1990], Kwan approaches two-dimensional convolution from three different perspectives. His first method minimizes the required I/O bandwidth as well as the number of processing cells. The second method compensates for slower processing elements by increased parallelism. Finally, the third method minimizes the number of I/O pins required for a VLSI implementation. For his part, Ersoy approaches the problem of circular and skew-circular convolutions using a semi-systolic array which requires greater communication between PEs but

has a smaller startup time, which is beneficial for small convolutions [Ersoy, 1985].

Although most general one-dimensional digital signal processing concepts [Oppenheim and Schafer, 1989] can be applied to two-dimensional problems, there are nevertheless a number of differences, mostly regarding filter design. Whereas the ideal one-dimensional low-pass filter is a $sin(x)/x$ function, in two dimensions the ideal *circularly-symmetrical* filter is a Bessel function of the first kind of order 1 [Dudgeon and Mersereau, 1984]. Since this function is not separable (i.e. it cannot be expressed as the products of two functions depending only on x or y), this is why two separate one-dimensional convolution operations cannot be used to do low-pass filtering (such attempts yield strongly anisotropic results: in general, the only separable filter is the Gaussian function, which often cannot be used as a filter since it does not roll off quickly enough).

Several methods for designing two-dimensional low-pass filters are outlined in [Lim, 1990] (from a low-pass filter specification, it is simple to generate corresponding band-pass, band-stop or high-pass filters). The most straightforward method is to take the ideal impulse response (which has infinite extent) and truncate it to a reasonable length (given the performance constraints under which the convolution will have to be performed). A simple rectangular window can have a fairly negative impact on the frequency response of the resulting filter (multiplication of the ideal filter in the spatial domain with a windowing function corresponds to convolution with the Fourier transform of the window in the frequency domain) Discussions on the merits of various windows for two-dimensional filters can be found in [Huang, 1972] and in [Speake and Mersereau, 1981].

There are several other possible filter design methods For instance, in the *frequency sampling* method, the frequency response of the desired filter is sampled and the Inverse Discrete Fourier Transform is applied to these samples to obtain the coefficients of the corresponding spatial domain filter. Although this method can be effective, as with the windowing method it does not produce an optimal

filter (that is, a filter with the minimal region of support / number of coefficients).

Frequency transformation methods seek to design optimal two-dimensional filters starting from an optimal one-dimensional design. A popular method for designing such filters is the Parks-McClellan algorithm, which is presented in [Parks and McClellan, 1972]. From this one-dimensional filter, a *frequency transformation* function is used to map the filter into two dimensions. For circularly symmetrical filter designs, the McClellan transformation can be used [Merserau *et al.*, 1976] [Mercklenbrauker and Merserau, 1976] [Psarakis and Moustakides, 1991]. Note that in all of these methods, one of the most important criteria for the "success" of the filter is the preservation of the zero-phase characteristic which ensures that only the magnitude of the image signal is affected and not its phase (those who are skeptical about the need for this are usually shown a demonstration where an image can be reconstructed with little alteration from its phase information only, whereas such an attempt using only the magnitude information fails miserably).

1.5.4 Image Processing and Computer Graphics

Low-level image processing and machine vision algorithms must often perform repetitive computations on large two-dimensional arrays of image pixels. For instance, the Carnegie-Mellon Warp systolic computer has been programmed to efficiently perform convolution, histogramming, Fast Fourier and Hough transforms [Gross *et al.*, 1985]. These algorithms are used as basic building blocks in most machine vision applications and greatly benefit from a system such as Warp which is fully programmable while retaining the high performance of an array processor.

A more recent system is presented by Choudhary and Patel in [Choudhary and Patel, 1988]. Their architecture is called NETRA: it is based

on a large number (100 to 10000) of processing elements which can be organized into clusters of 16 to 64 PEs each, a tree of control processors, a shared global memory and an interconnection network. The PE clusters can operate either in SIMD, systolic or MIMD mode. Implementation overviews are given for data compression, edge detection, feature matching, surface fitting, contour location and surface interpolation algorithms. One possible application would be a 3D stereo vision system, an important part of which is the recovery of depth information from a pair of images. Guerra and Kanade propose a systolic algorithm for this purpose [Guerra and Kanade, 1984], with an eye towards VLSI implementation.

HERMES is a multiprocessor vision system [Bourbakis and Barlos, 1988] consisting of $\frac{N^2}{4^i}, 0 \leq i \leq \log_2 N$ PEs where $N \times N$ is the size of the image to be processed and i is a resolution parameter (i.e. the size of the sub-regions into which the image will be decomposed). Contrarily to most systolic array processors, HERMES is a stand-alone system which does not require a front-end host. Image data is gathered directly from a photosensor array and fed to the PEs which process it in hierarchical fashion. Some of the algorithms implemented on HERMES include:

- General Coding Algorithm (GCA)
- Segmentation Region Analysis Algorithm (SRAA)
- Freeman Coding Algorithm (FCA)
- Simple Transmission Algorithm (STA)
- Order Decoding Algorithm (ODA)

The growth of multimedia applications has created a strong demand for image compression methods. One such scheme is adaptative vector quantization. The image is first decomposed into a set of vectors, from which a subset is chosen to form the basis of a codebook (most codebook generation algorithms attempt to iteratively generate a locally optimum codebook). Once this is done, the image

or set of images can be encoded (or quantized) using the codebook: only the labels of the codewords now need to be stored or transmitted. If many images need to be encoded, Adaptative Vector Quantization (AVQ) attempts to improve results by adjusting the codebook for each new image based on local statistics. Image reconstruction is done using a simple table look-up of the labels on the codebook, thus yielding a compression method where most of the effort is spent on compression: these methods are especially appropriate for digital media storage.

Clearly, codebook generation is an expensive procedure in all but the most trivial cases. Panchanathan and Goldberg propose a systolic array which can perform adaptative VQ [Panchanathan and Goldberg, 1991]. The systolic array is composed of $L \times N$ systolic cells connected in parallel where L is the vector dimension and N is the codeword dimension. Each cell can operate in two modes. In forward mode, it computes the basic *distortion* operation where the distance between a vector and a set of vectors is computed and accumulated. In the reverse mode, it computes the *centroid* operation which is used to average vectors into the new codewords. This architecture achieves a speedup of NL, and has the main advantage that the centroids do not need to be transferred into or out of the array.

Systolic architectures have also been used for computer graphics applications. For instance, a team of IBM scientists built a high-performance graphics system based on a custom-designed chip known as SAGE, the Systolic Array Graphics Engine [Gharachorloo *et al.*, 1988]. The objective of this architecture is to fight the memory bandwidth limitations which plague graphics systems: rendering algorithms such as Z-buffering, texture mapping and multi-sample anti-aliasing require ever larger video buffer bandwidths, while increasingly dense VRAM packages offer diminishing throughputs. SAGE maps a scan line of the display into a linear array of systolic processors, one per pixel. Drawing primitives are decomposed into scan-line fragments which are fed to the array at one end. As the fragment travels down the array, each processor decides whether to render the fragment based on the edge equation and depth information of the fragment (all of which is

computed incrementally). Each PE retains its current color and depth value. Once all the fragments have been fed to the array (one per clock cycle), a refresh token is sent, and the array begins to shift out the resulting pixel values which are used to generate the video signal. The array can then be used to scan-convert the next scanline in the display. Another computer graphics application was suggested by Megson, who uses a systolic array to generate B-Spline patches used in rendering curved surfaces [Megson, 1991].

1.5.5 General Algorithmic Computations

This section looks at general algorithmic problems which have been approached with systolic solutions. One such topic is the Algebraic Path Problem, which shows up under different guises in various fields: *transitive closure* and *shortest path problems* in graph theory, *matrix inversion* in linear algebra and the generation of *regular languages* in automata theory. The algebraic path problem is defined in terms of a weighted directed graph $G = V, E, w(e)$ where V the set of vertices, E the edges and $w(e)$ the weight (or cost) of the edges. If the vertices are numbered from 1 to N, then the objective is to find for each pair of vertices (i, j) the sum of the weights of all distinct paths from i to j. The APP is a $O(N^3)$ problem, and is thus expensive to compute on a serial-execution processor. Benaini and Robert propose a systolic array which requires $\frac{N^2}{3} + O(N)$ processors and can solve this problem in linear time, $5N - 2$ steps [Benaini and Robert, 1990]. A similar solution is presented by Lewis and Kung which uses N^2 processors and also requires $5N - 2$ steps to complete (interestingly enough, both designs claim to be "optimal"...) Scheiman and Cappello perform a rigorous analysis of the complexity of the method proposed by Lewis and Kung and come up with a precise lower bound of $\lceil \frac{N^2}{3} \rceil$ on the number of processors required for time-minimal completion [Scheiman and Cappello, 1992]. Similarly, Cappello analyzes the machine-independent maximum parallelism which can be realized in a systolic implementation of cubical mesh algorithms [Cappello, 1992]

(cubical meshes are used for a variety of algorithms such as finding the longest common subsequence among three strings, L-U factorization of matrices, three-pass transitive closure, matrix triangulation and inversion and two-dimensional tuple comparison)

Another interesting application is a systolic implementation of a Move-To-Front (MTF) text compressor suggested by Thomborson and Wei [Thomborson and Wei, 1991]. MTF compressors work by creating a list of "words". Instead of transmitting the symbol itself, the encoder transmits its current position in the list, and then moves the symbol to the front of the list. An adaptative Huffman or arithmetic code can be used to assign shorter codewords to the positions near the front of the list (which quickly get filled with the symbols which occur with the greatest frequency in the input stream). In the simplest case, the "words" can be the 256 8-bit bytes, although longer words yield better results. This method is suitable for on-the-fly compression and decompression for data transmission.

Priority queues are partially-ordered data structures which support two operations: *insert* which adds a new element to the structure and *deletemin* which deletes from the structure and returns the "smallest" element in the structure. On a sequential processor, both such operations require $3\log(n)$ steps if there are already n elements in the queue. Cheng proposes three alternative designs [Cheng, 1988]. The best solution requires $O(\log(n))$ processors and can implement both *insert* and *deletemin* in $O(1)$ (i.e. constant) time. Priority queues can be used to sort data (by first *inserting* all the data in the structure and then *deleteming* it in order: on a sequential processor, this would yield a $O(n\log(n))$ sort algorithm, whereas here linear time sorting would be achieved. Another application is discrete event simulation, where *events* are inserted into the priority queue according to their *arrival time*. At any iteration, the *deletemin* operation is used to retrieve the next event to occur: processing this even might cause later events to be scheduled and inserted back into the queue. The simulation terminates when the queue is empty. A similar

data structure is the hash table, which stores elements into a number of separate lists indexed by a *hash value* computed from the data element. Panneerselvam et al propose a parallel systolic hashing architecture which can be used, among other applications, to sort values in $O(n)$ time [Panneerselvam *et al* , 1988].

1.5.6 Pattern Recognition and Neural Networks

Pattern recognition often involves a large amount of calculations which have to be repeated many times and which reuse the same data over and over. Thus it is hardly surprising that systolic arrays have been proposed to solve these classes of problems. For instance, Frison and Quinton propose in [Frison and Quinton, 1984] a systolic machine which can perform continuous speech recognition in real-time with a vocabulary of 2000 words In this architecture, 89 processing elements are connected in a 2D array where each PE sends intermediary results to its right and bottom neighbors. This array performs the word spotting step of the process, which consists in detecting the words of the vocabulary in the speech signal The array receives phonemes as input from a phoneme analyzer and computes the probabilities that a word has been recognized given the phoneme string.

McWhirter proposes in [McWhirter *et al* , 1990] a systolic array for multi-dimensional fitting and interpolation using radial basis functions (RBF) The architecture is composed of two parts: the RBF pre-processor is used to determine the coefficients of the basis functions (radially-symmetrical Gaussian functions), whereas the second part is a least-squares processor which fits the data to be recognized using the basis functions from the RBF pre-processor The array operates in two modes: first, it is fed with a set of training data vectors from which it derives interpolation coefficients. This "knowledge" is then frozen and the array can be used on a set of test data vectors which must be recognized. The operation of the array can thus be related to that of a neural network based on the feed-forward multi-layer perceptron (MLP) model.

Since neural networks are intrinsically parallel computing structures, it is hardly surprising that systolic arrays have been used to implement these structures in an efficient manner. Kung proposes in [Kung, 1988] systolic arrays to implement both single-layer feedback networks (i.e. Hopfield neural nets) and multi-layer feed-forward neural nets Hopfield nets are formulated as a consecutive matrix-vector multiplication interleaved with a non-linear activation function. Each PE is used to model a neuron, and behaves differently in the search phase (where the neurons update their activation values) and in the learning phase (where the neurons are "trained") Chinn et al. [Chinn *et al.*, 1990] implemented these systolic arrays on the massively parallel MasPar MP-1 SIMD machine and applied them to speech recognition (the MasPar machine is further discussed in section 4.2). Concerned with the large number of learning iterations required of traditional MLP neural nets, Chiang and Fu [Chiang and Fu, 1990] propose a ring systolic array implementation which requires two orders of magnitude fewer learning iterations than conventional structures. Ramacher and Raab extracted the common computations in neural net models and propose a hardware systolic architecture which can efficiently perform these computations in parallel [Ramacher and Raab, 1990]. Whereas all of the methods proposed above exploit the spatial parallelism and the training set parallelism in neural networks, Chung et al [Chung *et al.*, 1992] propose a systolic structure which exploits the fact that forward and backward passes can be executed in parallel with pipelining of multiple training patterns in backpropagation neural nets. They apply this architecture to the NETtalk text-to-speech neural network which converts English text into phonemes.

1.5.7 Other Scientific Applications

Systolic arrays have also been proposed for other various scientific applications. For instance, a systolic array has been proposed to perform data encryption and decryption in Rivest-Shamir-Adleman (RSA) cryptosystems [Zhang *et al.*, 1988].

The RSA algorithm is a public key encryption method which is based on the difficulty of factoring large integers [Rivest *et al*, 1978]. A user of this system would create his keys in the following way:

1. Choose two random large prime integers p and q
2. Obtain the public modulus $N = pq$
3. Choose a random large integer D such that the greatest common divider $GCD(D, (p-1)(q-1)) = 1$
4. Compute $E = D^{-1} mod (p-1)(q-1)$
5. Publish the public key (E, N) and keep the secret key (D, N)

Thus anyone can use the user's public key to encrypt a message M into a cyphertext C using the equation $C = M^E mod N$, and the user can decrypt the text using $M = C^D mod N$. Although very secure, this method requires a fair amount of integer computations, especially if it is to be used for real-time encrypted communications (such as secure phone lines), hence the need for specialized parallel hardware. Two methods are proposed: one requires $\frac{3}{2}\left|\log E\right|$ processing elements arranged in a linear array, the second method is based on a double linear array with $2n$ PEs where n is the number of bits in E.

Systolic arrays have also found applications in experimental sciences. For instance, Squier and Steiglitz used a custom processor called LGM-1 to perform lattice-gas automata simulations [Squier and Steiglitz, 1990]. This allowed them to compare the results of simulations run on this architecture with results obtained from other methods and investigate the cause of erroneous results. In biology, the DNA sequence comparison problem has been approached using a custom-designed linear systolic array named P-NAC [Lopresti, 1987]. It was able to run two orders of magnitude faster than then-current minicomputers for that specific application. A similar solution to this problem is demonstrated by

Hoang [Hoang, 1992], who implemented his system on the SPLASH programmable logic array [Gokhale *et al.*, 1991]. Since DNA sequence comparison is basically a pattern matching problem, it is hardly surprising that systolic arrays are useful for that application.

Chapter 2 System Architecture

2.1 Introduction

This chapter, looks at the overall architecture of the convolution processor which as has been discussed before, is based on an array of specialized devices connected in a systolic array fashion. Greater emphasis will be put on the description of the structure and operation of some of the more relevant sections of the system; implementation details will only be covered in the following chapter.

2.2 Overall Architecture

Figure 2.1 presents the overall architecture of the convolution processor. Since most image processing research these days is being done on general-purpose UNIX workstations, the system is designed as an *attached processor* [Hwang and Briggs, 1984] which can be used to speed up convolution operations on such a platform. This dictated the choice of a standard and widely-used bus through which the system could communicate with a host processor. As further explained in the following section, a VMEbus interface was chosen for the implementation.

The interface between the host VMEbus and the system local bus implements a programmable Direct Memory Access controller. Source and destination images are stored in the main memory of the host processor. Figure 2.2 illustrates how the source image data is read from host memory by the DMA engine. The data is processed by the systolic array and then written back to host memory, completely

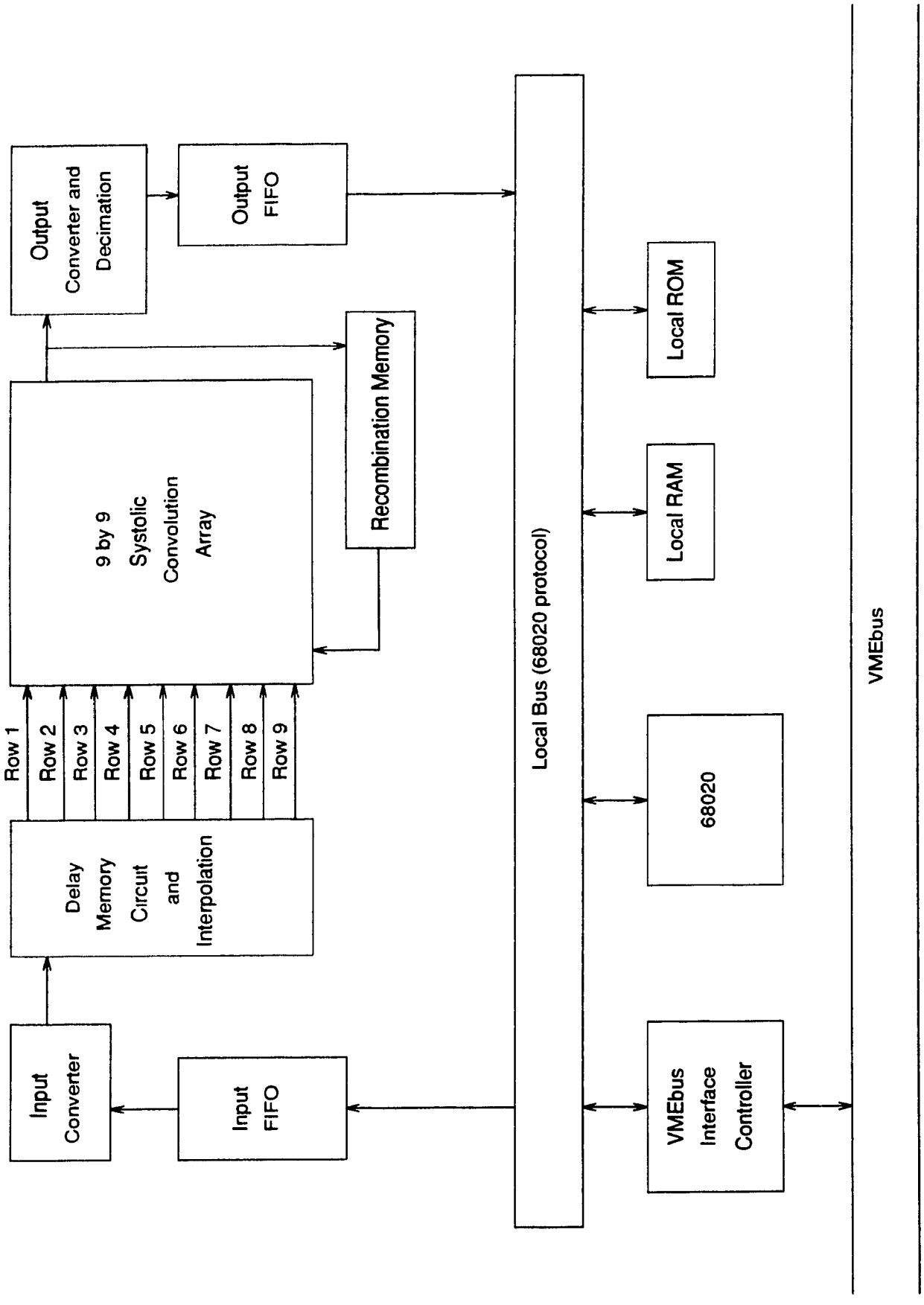

Figure 2.1: Convolution Processor System Architecture

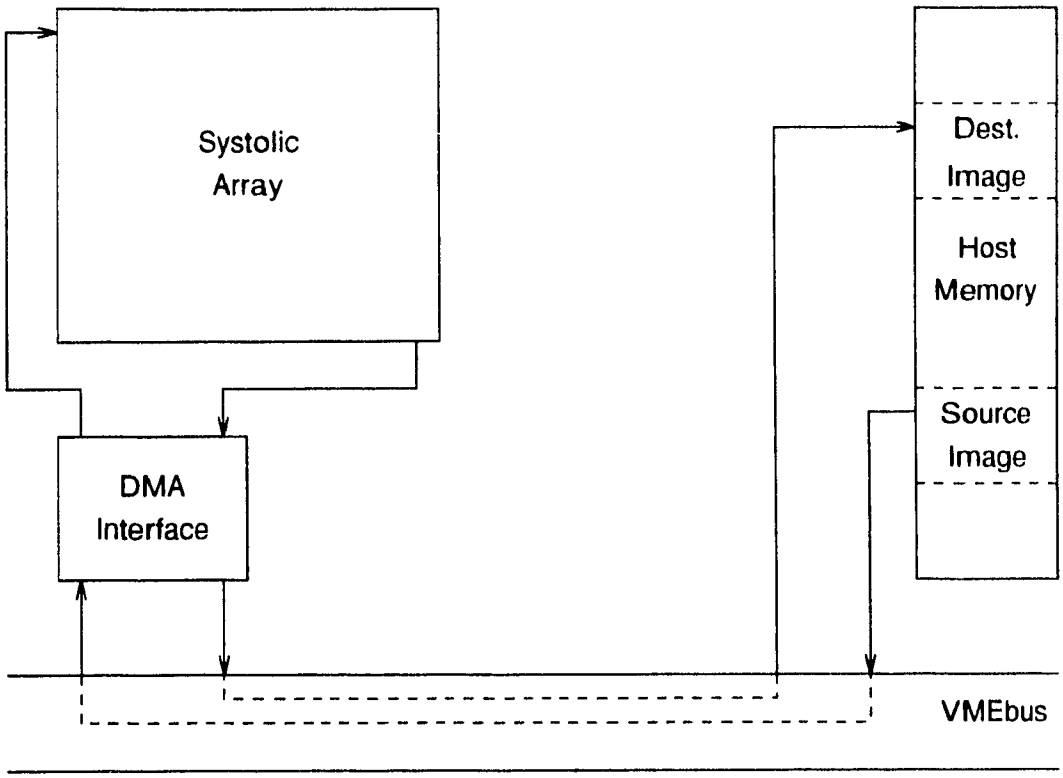

Figure 2.2: Data flow between host memory and systolic array

independently of the host CPU.

The DMA engine is based on a Motorola 68020 CPU and the VTC/Cypress VIC-068 VMEbus Interface Controller. This Application Specific Integrated Circuit (ASIC) implements a complete interface between the VMEbus and a 680x0-style local bus. The combination of these two devices yields an intelligent DMA controller which is fast enough to handle the required data rates, yet retains great flexibility. Since the core of the work performed for this thesis consists in the design and implementation of this VMEbus interface, the next chapter shall be devoted to its study. In the meantime, suffice it to say that the DMA engine is responsible for reading the source data from host memory into the Input First-In First-Out memory (FIFO), and to write the results from the Output FIFO back to host memory.

In most image processing systems, source images will be composed of integer data captured from such sources as cameras or laser range finders. Since the

systolic array operates only on 64-bit IEEE standard floating-point numbers, an input converter takes care of data type conversion. It currently allows both 8-bit and 16-bit input data. In certain cases, it is desired to perform several convolutions on the same image (perhaps interleaved with other processing steps). To minimize errors and loss of precision due to repeated conversions between numeric formats, intermediate results can be kept in floating-point format. Thus the input converter can also accept floating-point data which it passes along to the next stage without modification.

Each line of the image must be fed to the systolic array as many times as there are lines in the convolution kernel. This task is handled by the delay memory circuit which accepts as input the floating-point data from the input converter. It has sufficient memory to buffer the required input image lines, and can feed this data to the lines of the systolic array in the required order. In this way, the lines of the source image do not have to be fetched multiple times from host memory, thus greatly decreasing the required bus bandwidth. The delay memory circuit also implements the image interpolation feature of the system: it does this by inserting zero values between the pixels of the input image in order to raise its sampling rate. Convolving this up-sampled image with a low-pass filter of the proper cutoff frequency and phase will replace the newly introduced zero values by the desired interpolated values while keeping the original pixel values unchanged.

The systolic array performs the actual convolution operation. The lines of the image are fed to each line of the array in turn from the left side. These input values and the partial results which they generate propagate from left to right, and in the case of the partial results, from the output of the right-most processing element of a line to the input of the left-most processing element of the next line. The final result comes out of the output of the processing element at the bottom-right of the array. Note that the kernel coefficients have been pre-stored in each of the processing elements, one coefficient per such device.

The output of the systolic array can go to two destinations. A recombination memory is used to store intermediate values when performing multi-image operations, such as is needed when performing color-recombination [Malowany and Malowany, 1989]. The data can also go the output converter which will convert the resulting floating-point data back into 8-bit or 16-bit integer format. This is often required since the following steps of the image processing algorithm might not require the full precision of floating-point results. If the output of the convolution processor represents the final results of the algorithm, then it might need to be displayed in a frame-buffer, which can usually accept only small (8-bit) integers. The conversion from floating-point to integer format is done by a binary search into a look-up table of interval limits: since this look-up table is programmable, it is possible to select an arbitrary non-linear mapping. Another function of the output converter is that it can selectively ignore output results to implement sampling rate reduction (or *down-sampling*) when combined with the proper filter. As was the case for the input converter, the output converter can be configured to pass data through without modification if floating-point output is desired. The output of the output converter goes into the output FIFO, from where the data is read by the DMA engine to be written back into the host system memory.

2.3 Host Bus Selection

Whenever a peripheral device is to be designed, the selection of the host interface bus is one of the first design decisions which is made. In the commercial world, this choice has a strong effect on the profitability of the design since although it may be technically and economically viable, no one will want to purchase it if it cannot be used with popular computing platforms. Fortunately in this case, there were no such economic pressures. The selection criteria were the following:

1. The host bus has to have enough bandwidth to keep the systolic array from "starving". If floating-point operands are used both for the input and the output, this translates to a required bandwidth of 16 Mb/s.

2. The form factor of the boards has to be large enough to allow implementation of a fairly large and power-hungry design.

3. The bus protocol must allow multiple bus masters so that the DMA engine may take control of the bus while transferring image data.

4. The bus interface has to be simple, or else ASIC solutions must be available which implement a reliable and complete interface.

5. It has to be compatible with the equipment used in our research group.

A previous project where an integer convolution processor was designed and built [Boudreault and Malowany, 1986],[Haule, 1990] had used the Multibus or IEEE-796 standard [Multibus, 1983], but although it is still used in some industrial applications, this platform is now obsolete. A logical successor might have been Multibus-II, but at design time there were still no single-chip interface solutions, and very few systems actually use this standard (although its designer, Intel, appears to be trying to resuscitate it as a platform for high-end PC compatible file-servers: whether this effort will succeed is unknown at this time).

The ubiquitous Industry Standard Architecture (ISA, also known as AT) bus fails to meet criteria 1 to 3. Its successor, Extended Industry Standard Architecture (EISA), as well as the IBM MicroChannel satisfy criteria 1 and 3 but do not offer much board real-estate. Furthermore, at the time of the design there were no off-the-shelf solutions which implemented a bus-mastering interface.

So-called "Mezzanine" buses such as SBus (from Sun Microsystems) or TurboChannel (from Digital Equipment Corporation) offer a lot of bandwidth, but suffer from very small board form factors. This is not so much a problem when

contemplating a high-volume design where surface-mounted components can be used on both side of the board, but it severely restricts the amount of available space in a prototype design such as ours. Furthermore, although their promoters would have us believe otherwise, these are essentially proprietary solutions which find little use outside of the products offered by these companies.

So the choice was made to implement a VMEbus interface. As will be seen in the next chapter, VME satisfies all of our criteria. Its peak bandwidth (40 Mb/sec) is sufficient for our needs and there exists a number of ASIC interface solutions which implement a bus-mastering interface. VME is used by a number of machines at our facility, including the larger Sun and Silicon Graphics workstations, as well as the VME-based Sensory Computing Environment [McRCIM, 1990] being developed here.

Note that in the future, such a design would probably be implemented on Futurebus+, the next generation in general-purpose computer busses [Futurebus+, 1990]. Although FutureBus+ interface silicon is just starting to become available, there is a growing interest in this standard due to the tremendous performance it offers. Recently introduced high-end servers from Digital Equipment Corporation based on the Alpha micro-processor use Futurebus+ to give these machines high inter-processor and I/O bandwidth. The U.S. Navy has standardized on Futurebus+ for all on-ship computing systems (among other considerations, the ability to insert and remove boards from a Futurebus+ backplane without powering down known as "hot" or "live" insertion is appreciated in systems which must maintain very high availability). As costs fall and bandwidth requirements increase, this bus standard might start appearing in lower-end products.

2.4 Input Converter

As outlined previously, the function of the Input Converter is to transform input data formats into the IEEE-754 standard floating-point format which is understood by the convolution array. It then passes this data on to the next stage, the Delay Memory Circuit.

2.4.1 Data Formats

The input converter understands the following input data formats:

1. 8-bit unsigned integer
2. 16-bit unsigned integer, big-endian
3. 64-bit IEEE-754 double precision floating-point format, big-endian

In the case of the second and third formats, big-endian byte order (i.e. the highest byte first) was selected somewhat arbitrarily: since data is read directly from host memory by the DMA Engine, this has to be compatible with the data format used by the host. Since convolver discussed here is to be used with SPARC and Motorola-based hosts, big-endian ordering was a natural. It is also assumed that the host uses IEEE-754 as its floating-point storage format: very few current machines (with the notable exceptions of those based on the VAX, IBM 360 and CRAY architectures) use a different format.

2.4.2 Overall Architecture

Figure 2.3 shows the structure of the Input Converter. Status bits controlled by the local CPU are used to define the data type to expect. It reads its operands from the

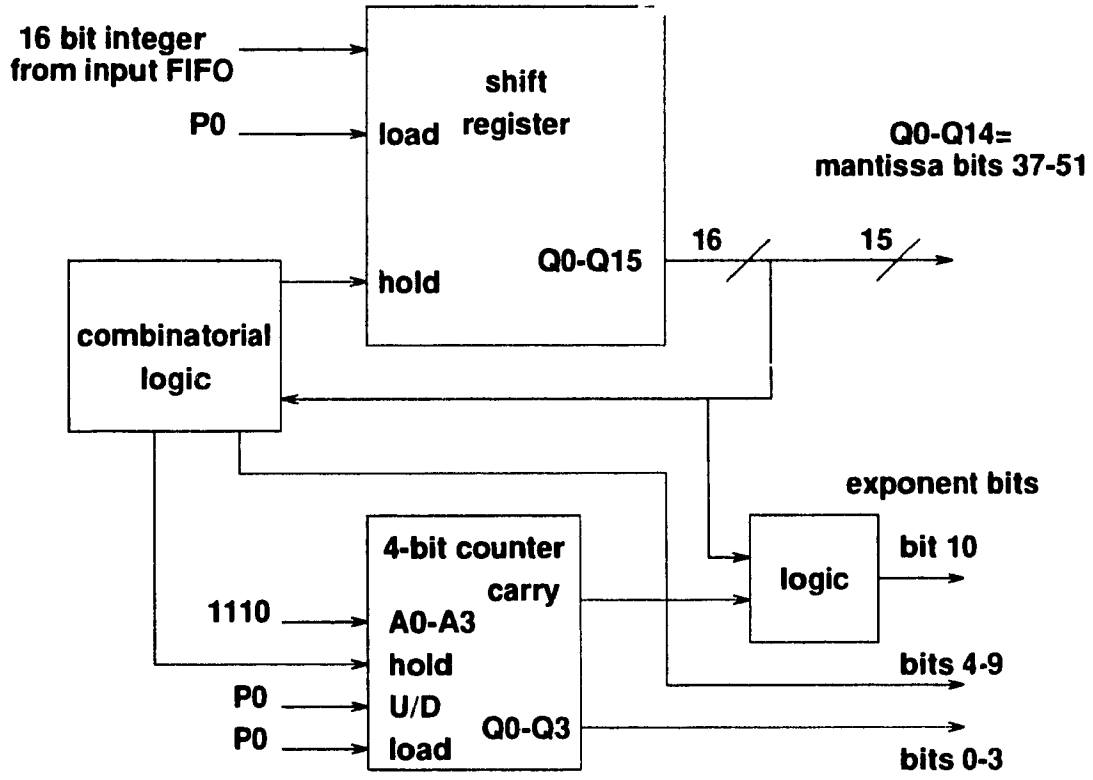

Figure 2.3: Input Converter Block Diagram

outputs of the Input FIFO. Since it is made up of 4 8-bit wide devices, the control logic must decide which outputs to enable in the proper sequence, based on the type of data. Floating-point operands do not need to be processed, so they are passed on directly to the output of the Input Converter. Both 8 bit and 16 bit data are treated as 16 bit data, the 8 upper bits of the former being set to zero.

Conversion of a 16 bit integer operand into floating-point format is done in the following way. The integer is loaded into a 16 bit shift register, while a 4-bit counter is initialized with the desired value (8 or 16). The operand is then shifted left until a 1 appears in the most significant bit of the shift register, at which point the content of the shift register will represent the normalized mantissa of the floating-point number. For every left shift operation, the counter is incremented and yields, in the end, the floating-point exponent. The entire procedure takes at most 16 clock cycles, which corresponds to the cycle of the entire system for an operand (i.e. in most cases, the convolution array requires one input operand and produces one

output data item every 16 cycles).

A detailed description of this procedure is outside the scope of this thesis: interested readers are referred to [IEEE-754, 1985] for details on the floating point format and [Drolet *et al* , 1990],[Drolet, 1992] for operation and implementation details. Suffice it to say for now that a double-precision number uses 52 bits for its mantissa, 11 bits for its exponent (stored in excess-1023 notation) and 1 bit for its sign. The low 37 bits of the mantissa are always set to zero by the converter.

Once the desired floating point exponent and mantissa have been obtained, the Input Converter needs to serialize its output to the next stage, the Delay Memory Circuit, since it expects floating-point operands only 8 bits at a time, starting with the low-order byte.

2.4.3 Implementation Considerations

In most designs, space is an important limiting factor. The VMEbus 6U double-high, double-wide form factor offers 373cm^2 of board space, which is not very much when considering the complexity of this design. Thus an implementation based on Programmable Array Logic (PAL) devices and other random logic would consume too much space. A better solution is the Field Programmable Gate Array (FPGA). As with the more conventional gate arrays, the FPGA is composed of an array of logic blocks which can be connected to generate any desired combinatorial or sequential logic circuit. The main difference is that whereas the interconnections in conventional gate arrays are permanently manufactured into the chip (typically as metalization layers), FPGAs are programmed on power-up by loading a configuration bit pattern into RAM memory locations. Thus FPGAs do not suffer from the long lead times and high non-recurring expense (NRE) of gate array or standard cell devices. On the other hand, their unit cost is significantly higher, so they are typically used for low to mid-volume designs [Mokhoff, 1993].

The very quick turn-around time of these devices means that they are often used for prototyping systems, after which they can be migrated to more conventional devices when production volumes justify the NRE [Egan, 1991] FPGA devices have also been used to design a completely reconfigurable parallel processor named SPLASH [Gokhale *et al* , 1991] Note that FPGAs have somewhat lower performance than gate arrays or standard cell devices. Furthermore, although high-level software which can accept a design in the form of logic equations or standard library parts and map it onto the FPGA logic block architecture exists, it usually produces non-optimal designs which either fail to use all of the available real-estate or generates unacceptable routing delays. For high-performance applications, the designer is often forced to manually specify the partionning of the design into the logic blocks of the part, as well as the routing between these blocks, a tedious task at best Newer Computer Aided Design (CAD) tools as well as better FPGA architectures hope to lessen the burden on the FPGA designer [Bursky, 1992] [Clark, 1992]. The ultimate objective is to be able to automatically synthesize the desired FPGA design from a high-level functional and/or behavioral description in a circuit-description language such as VHDL [VHDL, 1987] [Perry, 1991] or VeriLog [Sternheim *et al.*, 1990]. A survey of current FPGA architectures and programming technologies can be found in [Rose *et al.*, 1993]

The FPGA configuration information can be stored in a ROM device. special 1 bit serial ROMs exist which interface directly to the FPGA device, or standard byte-wide EPROMs can be used with a minimum of "glue" logic. The configuration can also be stored in a disk file and downloaded into the device at power up by a host: this is the approach chosen here, since it allows maximum flexibility This is made possible by the fact that the DMA Engine is there to configure the board after power up, and that none of the bus interface circuitry depends on FPGAs.

Thus the input converter is implemented using a single Xilinx FPGA device, which connects directly to the outputs of the Input FIFOs and to the inputs of the Delay Memory Circuit. Note that in order to allow little-endian 16-bit operands, all

that would be required is to change the order in which the FIFO devices are read to implement the byte-swap operation (all 16-bit operands are assumed to be word-aligned in host memory). This would require changing the state machine which controls the outputs of the FIFOs and the multiplexers at the input of the device: such design changes only generate a different FPGA configuration file which can be downloaded into the device at power up. Similarly, little-endian IEEE-754 operands could also be supported by changing the operation of the serialization circuitry at the output of the device. FPGA devices are used extensively in current industrial designs. In many cases, their flexibility allows working around other problems with the non-programmable sections of a design or adding functionality to an existing design without having to change a single connection.

2.5 Systolic Array

The systolic array used in this design is based on a custom VLSI processor which implements the basic operation $Y_{out} = CX + Y_{in}$ where C is the convolution kernel coefficient, X is the pixel intensity, Y_{in} is the output of the previous processor and Y_{out} is the partial sum to be fed to the next processor. Every operand is in double precision floating point format. Figure 2.4 represents a 3 by 3 systolic array: a similar topology is used for a 9 by 9 array. Each line of the source image is fed pixel by pixel to the left side of the array (in the case of figure 2.4, the first line of the image would be fed to $IN1$, the second to $IN2$, the third to $IN3$). The pixel intensities move from left to right, from processor to processor. When the first line of the image has been completely fed to the first line of the array, the indexing of the source image is incremented and the second line is then fed to the first line of the array. Thus each line of the image must be fed to the array as many times as there are lines of processors. The delay memory circuit is responsible for feeding the image lines in the proper order to the array: its operation is explained in section 2.7.

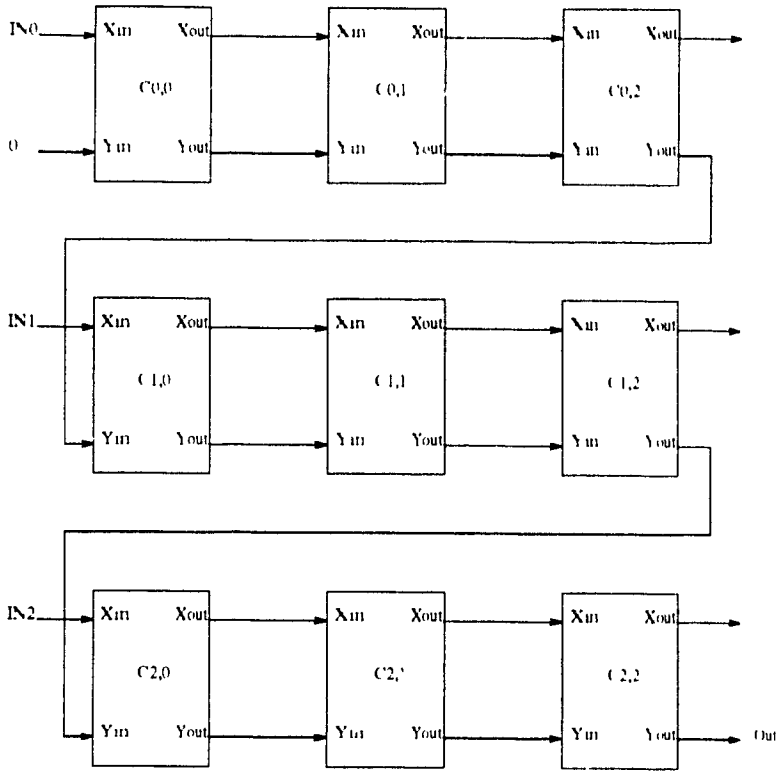

Figure 2.4: 3 by 3 systolic array

Figure 2.5 outlines the architecture of the VLSI convolution chip. The design of this full-custom device is documented in [Larochelle *et al.*, 1989], [Côté, 1990] and [Larochelle, 1991]. Data is transmitted between the chips 4 bits at a time so as to limit the number of required I/O pins (a limitation of the packaging offered at the time for production of these devices by the Canadian Microelectronics Corporation [CMC, 1989]). Since operands are 64 bits wide, 16 clock cycles are required to transfer a complete operand between two processors (note that the pixel value and the partial result are transfered at the same time). The source image pixels are fed to the device on the X_{in} input and stored in a 32 entry deep, 4 bit wide FIFO. The pixels come out unmodified on the X_{out} output which is connected directly to the X_{in} input of the next device As the 4-bit components of the pixel intensity travel down the FIFO, they are also fed to Stage 1 of the device, where they are multiplied by the kernel coefficient stored in the device (note that the coefficients do not change throughout a convolution operation, and remain

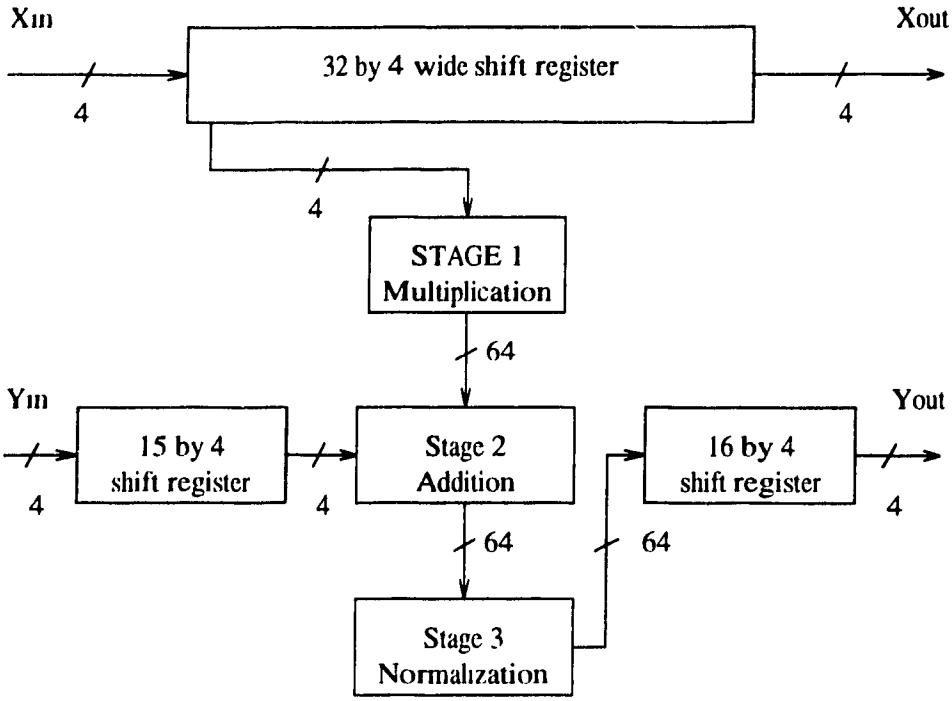

Figure 2.5: Systolic Cell Architecture

fixed inside the device). This multiplication unit is capable of multiplying two 4-bit quantities in one clock period, which means that it operates at the same speed at which the pixels are shifted into the device.

The result of this multiplication is parallel loaded into the stage 2 adder, where it is added in 16 clock cycles to the partial result generated by the previous device in the array. Finally, the result of the addition is parallel loaded into the stage 3 normalization unit, where the mantissa is aligned to generate a valid floating point number, again in 16 clock cycles. A shift register is used to put together the 64 bits of the Y_{in} partial result input and offer them in parallel to the stage 2 adder. Similarly, the output of the stage 3 normalization unit is loaded in parallel into a shift register which will shift it out 4 bits at a time on the Y_{out} output. Thus every 16 clock cycles, a convolved pixel comes out of the convolution array (after a suitable delay required to fill the pipeline).

Note that in order to be *truly* compliant with the IEEE-754 standard, an additional renormalization operation would have to be performed between the multi-

plication and addition steps. Since this is not the case, the convolution processor might yield slightly different results than those that would be obtained when implementing the operation on a machine where renormalization would occur after both the multiplication and addition. CPUs with a multiply-and-accumulate unit often forgo the intermediate renormalization for performance reasons: such is the case with the IBM RS/6000 [Bell, 1990]. If precisely reproducible results are required, RS/6000 compilers can be told not to generate multiply-and-accumulate instructions, but instead generate separate IEEE-compliant multiply and add instructions· of course, performance is greatly reduced in that case.

Figure 2.6 illustrates the pixel values as they travel down a line of the convolution array. Only a single line is shown to make the diagram clearer, but the same principle applies to the two dimensional array. Note that the partial results travel half as fast through the array as the input pixel values. This is due to the insertion of two extra "pixel" delays (i.e. 16 clock cycles, since the 64 bits of a pixel operand are transmitted 4 bits at a time) in the partial sum path for a total of a 4 pixel delay, as opposed to a two pixel delay in the pixel value path. This ensures that the proper partial sums get propagated at the right time. In the last line of our example, the result for the first pixel of a 3 by 1 convolution is ready to come out of the last device in the chain Note that it took 10 system cycles of 16 clock cycles each for the first valid result ($C_0X_0 + C_1X_1 + C_2X_2$) to come out (all previous output was undefined and is ignored by the rest of the system). This is typical of pipelined systems, where there is always a penalty to pay as the pipeline is filled. After that, a valid result will come out of the last device every 16 clock cycles (i.e. $C_0X_1 + C_1X_2 + C_2X_3$, $C_0X_2 + C_1X_3 + C_2X_4$ and so on...).

The current version of the convolution chip is clocked at 16 MHz, which enables the convolution array to produce a convolved pixel every microsecond. For a standard 512 by 512 image, convolution can thus be performed in roughly a quarter of a second (262 msec.). Kernel size depends on the number of devices used to build the array: the current design uses 9 by 9 chips. Since each chip performs a complete

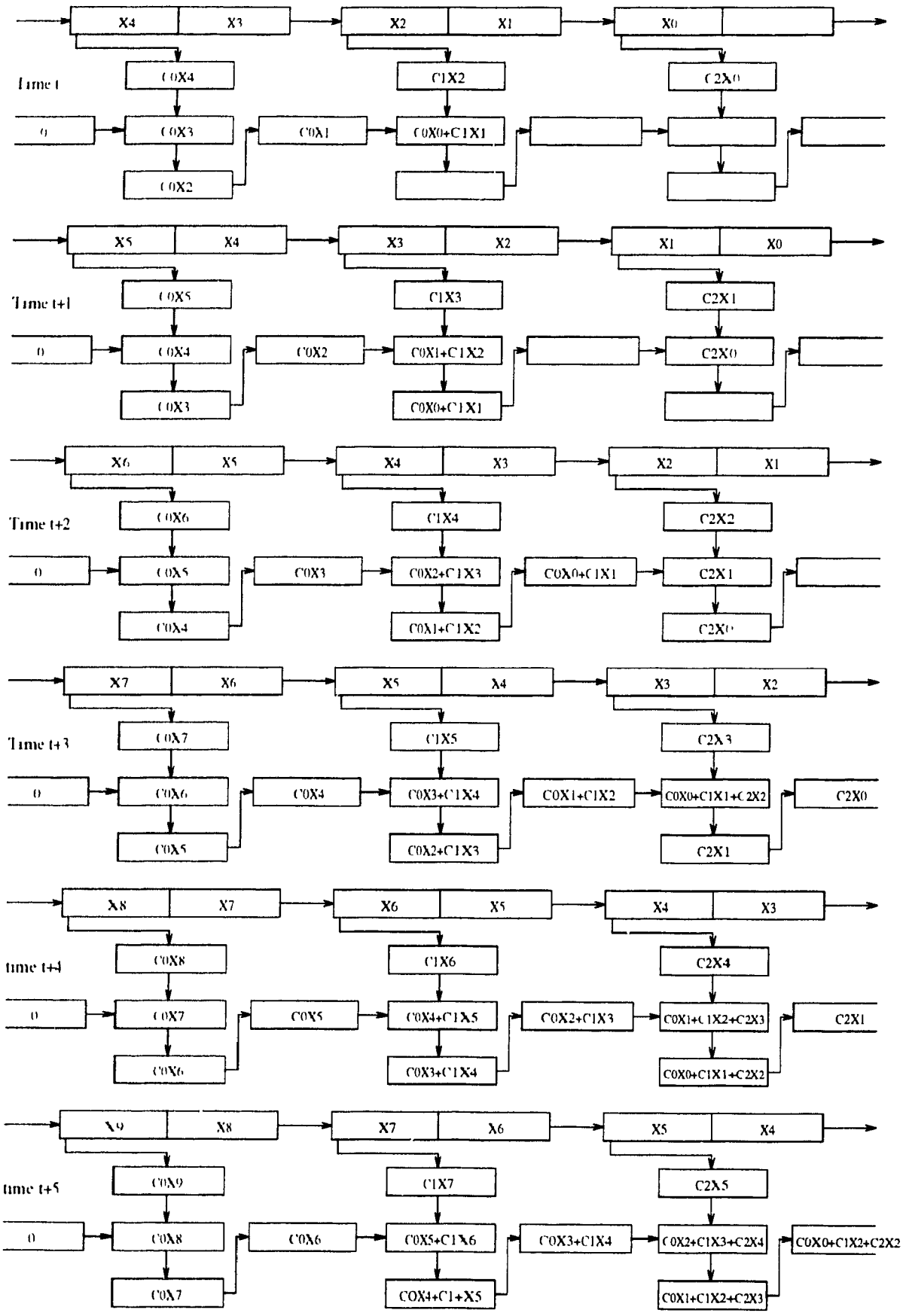

Figure 2.6: Systolic Array Data Flow

floating point multiplication and addition every microsecond, this yields a system performance of 162 MFLOPS, sustainable throughout the convolution operation, as long as data can be read from and written to host memory fast enough. Although the maximum bandwidth required to sustain this rate is 16 Mb/sec, which is well within the 40 Mb/sec maximum bandwidth of the VMEbus, other factors such as the speed of the host memory and contention from other VMEbus masters might reduce the bandwidth available to the convolution processor.

2.6 Recombination memory

As the convolved pixels come out of the array, they are routed to the output converter (see figure 2.7) to be optionally transformed back to integer values before going back to destination memory storage. They can also go to the recombination memory, which is used to buffer an entire image. The content of this memory can then be used to drive the partial sum input of the first device in the convolution array. This allows multi-image operations to be performed, such as image averaging/blending. For instance, a first image could be processed with all of the kernel values scaled by a value α, and then stored into recombination memory. The kernel values would then be reloaded, this time scaled by a value $1 - \alpha$, and the new image sent through the array, using the result of the first convolution as the initial partial sum input, thus effectively blending between the two resulting images. 4 Mb of memory is allocated for this purpose, organized in 4 SIMM modules of 1Mb by 8 bits each. Since 4 bits must be read out for each 16 MHz clock cycle, this means that the memory must have a cycle time of 125 ns (since two operands are read at once from the 8-bit organized memory). This would require rather fast DRAM devices: on the other hand, since memory access is always sequential, the access pattern is trivial to predict. Thus two-way interleaving is used to relax the cycle time requirements to 250 ns, which is fairly easy to satisfy with inexpensive DRAM devices.

2.7 Delay Memory Circuit

One of the greatest obstacles to high performance in convolution implementations on traditional architectures is the high memory bandwidth required by the repeated use of the same operands. In the case of a 9 by 9 convolution, each input datum has to be read 81 times from main memory. Careful design of the algorithm can minimize the number of cache misses by taking into consideration the cache line size [Stone, 1987], but such optimizations are often hardware-dependant and can actually decrease performance on a machine with different architectural characteristics. In order to take full advantage of the systolic architecture of our system as well as to minimize the bandwidth on the VMEbus, a Delay Memory Circuit is implemented.

This circuit takes as its input the unidimensional stream of image pixels in scan-line order from the input converter. Each line of the image has to be fed to the systolic array as many times as there are rows in that array. This is done with 8 circular queues implemented using standard static RAM devices. The first row of the systolic array is fed directly by the pixels coming from the input converter, whereas the other lines receive delayed copies of the rows previously stored into the circular queues (note that pipeline delays have to be taken into consideration by the control logic to determine the exact time at which pixels have to be fed to the array). Since it has a finite amount of storage, the delay memory circuit imposes a practical limit on the length of a raster line. It is currently implemented using 8 8kx8 static RAM devices. Since at that point the operands are in 64-bit double precision format, this means that raster lines can have a maximum width of 1024 pixels. A deeper treatment of the implementation details can be found in [Drolet *et al.*, 1991],[Drolet, 1992].

The Delay Memory Circuit has two other functions. It is responsible for determining whether the convolution array is to be used in 1 or 2 dimensional mode. In 1-D mode, the array implements a single FIR filter with 81 coefficients. The

data is transmitted directly from the input converter to the first line of the array, and 0 values are fed to all the other lines: in that mode, the circuit is basically bypassed. In 2-D mode, it operates as previously described. It can also be used to perform limited interpolations used to increase the sampling rate of an image by a factor of 2 or 4 by inserting appropriately placed 0 values into the data stream (i.e. inserting 1 or 3 zero values between all input pixels, and 1 or 3 lines of zero values between lines of input pixels). The coefficients loaded into the convolution array must then implement a low-pass filter with the appropriate cut-off frequency. Note that multi-rate filters are usually much more efficient for implementing resampling operations (in particular, they avoid the numerous multiplications of zero input values used in our method), but this extra functionality was achieved at the cost of a small amount of additional complexity in the control logic for the Delay Memory Circuit.

2.8 Output Converter

The function of the output converter is to transform the floating-point output of the systolic array back into integer format. This is required for instance when the output is to be viewed on a display device such as a frame buffer. Typically, for a gray-scale image this means that the results must be quantized down to 8 bits of resolution, yielding 256 distinct levels of gray. The recently announced Silicon Graphics RealityEngine graphics subsystem allows up to 12 bits per color component, which in the case of gray-scale images gives 4096 shades of gray (although the actual Digital to Analog Converters which drive the display only have 10 bits of resolution: the extra bits of resolution in the frame buffer are used, among other things, to avoid loss of precision in the lower intensities due to gamma correction).

Index	Content
3	1.9
2	1.3
1	0.5
0	0.2

Table 2.1: Conversion look-up table content

2.8.1 Principle of Operation

The conversion from floating point results into integer format is achieved by means of a look-up table. The principle is that the entries of this table contain the bounds of successive intervals. By using a binary search into this table, the converter determines the interval in which the floating point result falls: the index of the corresponding entry yields the desired integer result. For example, table 2.1 shows the contents of a table with 4 entries.

Using this example, a number between 0 and 0.2 would be mapped to the integer 0, from 0.2 to 0.5 to 1, from 0.5 to 1.3 to 2 and from 1.3 to 1.9 to the integer 3. The only restriction on the values of the interval bounds is that they be monotonically increasing in order for the binary search to be able to find the right interval. This allows for non-linear mappings of floating-point results to integers. This could be used for:

- Dynamic range compression/expansion: when the application is interested in a narrow range of values in the output, all values below that range can be clamped to zero, all those above can be clamped to the maximum index and the full range of integers can be used for the "interesting" values in between. Similarly, a range of values which is not interesting can be compressed to a single interval, again yielding more dynamic range in the output for other intervals of interest.

- Most image processing algorithms assume that the range of possible values

for each image sample is linear. Unfortunately, most output devices (CRTs in particular) do not generate a linear intensity as a function of the value to be displayed. This relationship is usually modelled by a power function and is known as gamma correction [Travis, 1991]. If the display which is to be used does not have gamma correction hardware (i.e. instead of feeding the values to be displayed directly to the digital to analog converters, these are used as the input of a look-up table which implements this correction), then the output converter can be programmed to perform this correction on the image itself. Note that the image then becomes dependant on the particular display it was corrected for, and can only be displayed on another system after being gamma corrected again, at the price of a substantial loss in dynamic range in the low intensities.

- Even if intervals of constant length are used, the table can still implement a simple gain and offset mapping.

2.8.2 Output Converter Architecture

Figure 2.7 presents the block diagram of the converter. The 24 most signicant bits of a floating point output value from the convolution array are stored in a register. This means that only 12 bits of mantissa are retained for the purposes of comparison. Clearly, this imposes a limit on the resolution of the intervals which can be specified. Similarly, the interval bounds are stored as the 24 MSBs of the desired intervals in the output converter look-up table. Note that this table is programmable, but cannot be changed while an image is being processed. The table contains 2^{12} entries, and thus requires 12 bits of address.

As the conversion begins, the 12 bit A register is cleared and the 12 bit B register is set. Their values are added by a carry-lookahead adder and the result is divided by 2 using a hardwire shift. This yields a 12 bit address which points to the middle

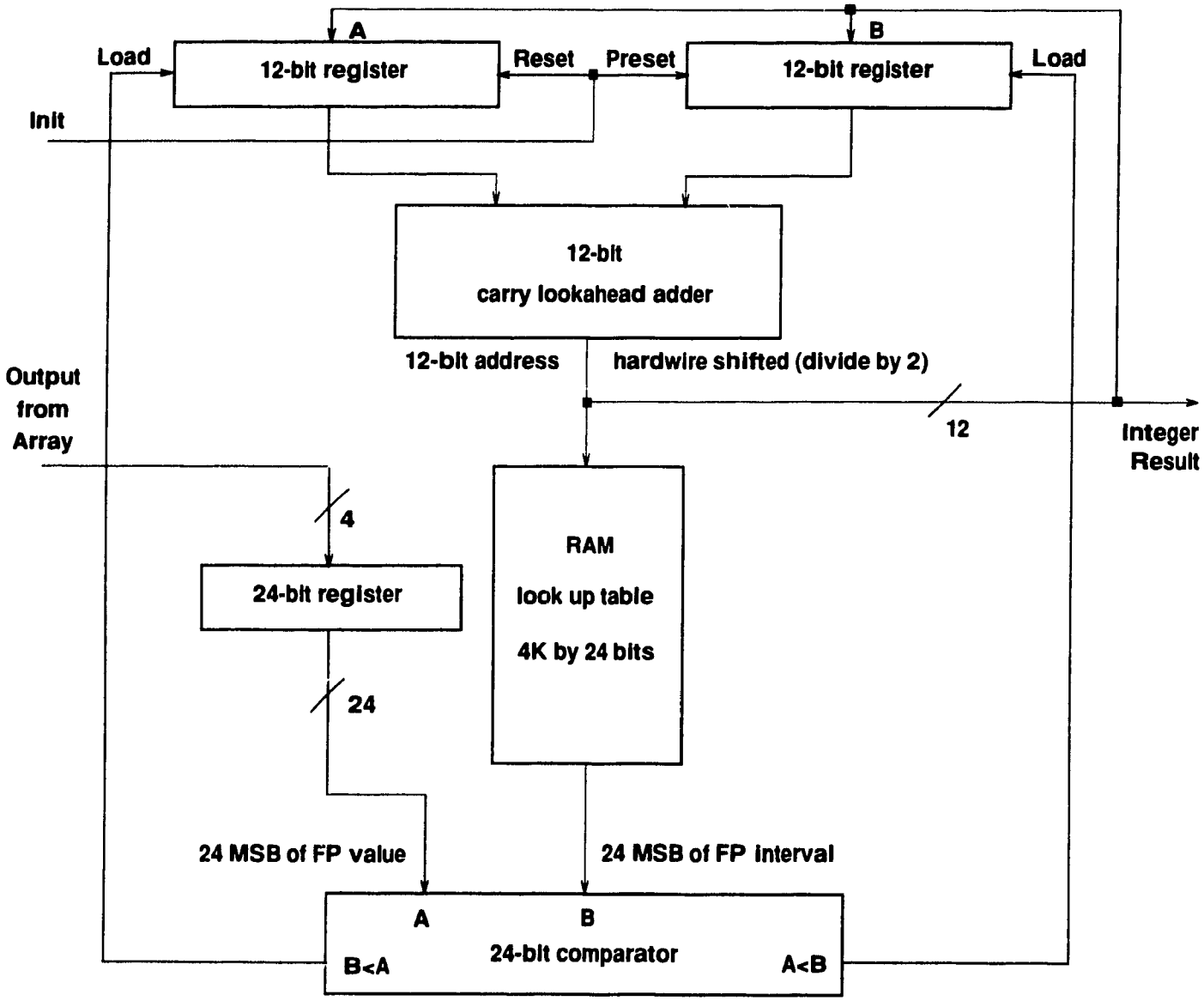

Figure 2.7: Output Converter Block Diagram

of the table. The correspoding interval boundary is fetched and compared with the result to be converted. According to the result of this comparison, either register A or B is loaded with the address previously computed, thus dividing the search interval in half. After 12 iterations, the proper interval has been found, and its address in the look-up table is the desired integer result, which is sent to the output FIFO. Thus the output converter can generate up to 12 bits of precision on its output. Since the convolution array produces a new result every 16 clock cycles, the same clock can be used to drive the output converter since it only requires 12 cycles for a conversion.

As was the case for the input converter, the output converter can be bypassed if floating-point results are desired. In that case, the output of the convolution array is sent directly to the output FIFO. Also, in order to implement the sample rate conversion capabilities of the system, the converter can be programmed to decimate (downsample) the output image by ignoring some of the output samples. In order for this down-sampling to occur without aliasing, the convolution coefficients must have been chosen to implement the proper low-pass filter. Again, a XILINX FPGA device implements the main functionality of the output converter. External static RAMs are used to store the interval look-up table. Further details regarding the implementation of the output converter can be found in [Drolet, 1992].

Chapter 3 DMA Engine Implementation

3.1 Introduction

This chapter covers the design and implementation of the section of the convolution processor known as the DMA engine. The main task of this subsystem is to coordinate the transfer of image data from the memory of the host CPU over the VMEbus to the convolution array, and the writing of results back into host memory. Since it is designed around a general-purpose processor, the DMA engine is flexible and accepts high-level commands from the host CPU. It is also responsible for general initialization and control tasks for the other sections of the convolution processor.

3.2 Design Perspective

The DMA engine of the convolution processor is responsible for the following tasks:

- initialize the system after power-up or reset
- load the kernel coefficients into the systolic array
- select operating modes and perform other control tasks
- interface to the VMEbus
- transfer data over the VMEbus from the memory of the host CPU to the input converter, and from the output converter back into host memory

The initial design approach was to design a special-purpose circuit built around a commercially available DMA controller. After some amount of work, this was rejected for a number of reasons. There were very few available DMA controllers which support 32-bit address and data paths (the entire VMEbus address range must be supported), and those that are available are usually overly complex or have fairly small bandwidth. Furthermore, it was quickly realized that since a VMEbus interface with both master and slave capabilities is needed, it would be beyond the scope of this project to attempt to synthesize this interface from standard logic components (ignoring for a moment such constraints as board area) Thus an off-the-shelf VME interface device had to be used. Unfortunately, all such devices assume that they are connected to a CPU and act as a bridge between the CPU local bus and the VMEbus. A company called PLX makes a set of five devices which can be used for somewhat lower-level interface, but a design using a Motorola DMA controller and these devices was rejected as unduly slow and complex.

The conclusion was that the best approach was to have an on-board CPU, and to use an off-the-shelf ASIC to bridge between its local bus and the VMEbus The VTC/Cypress VIC-068 ASIC was chonse because it seemed like the most viable solution: this device is endorsed by a large group of VME board manufacturers known as VITA, the VMEbus International Trade Association, and is used in a number of commercial products. Since the VIC offers a glue-less interface for a Motorola 68020/68030-style CPU local bus, it was natural to chose to use a 68020 CPU since the virtual-memory capabilities of the 68030 were not needed Having a general-purpose CPU on board means that many of the control tasks can be done in software, which greatly enhances the flexibility of the board, reduces the amount of control logic which has to be designed and lessens the risk of a fatal hardware bug which cannot be fixed in software. Furthermore, since the VIC is able to perform most of the DMA transfer functionality on its own at high-speed, the 68020 does not need to have high performance. This greatly relaxes the design constraints on that section of the circuit, and permits the use of a low-speed part (in this case the 68020

will run at 12.5 MHz). This approach was suggested in a Motorola application note for the 68020 CPU [Motorola, 1987]. It was first presented in [Panisset *et al.*, 1990].

3.3 System Block Diagram

Figure 3.1 shows a high-level block diagram of the DMA engine (note that this block diagram is very similar to the top-level sheet of the circuit schematic). The VIC-068 implements a bridge between the VME system bus and the local bus of the 68020 processor. The only additional logic required are decoders to map the slave interface offered by the VIC into the VME address space as well as additional transceivers and latches to isolate the local bus from the VMEbus. The interface offered by the convolution processor to the host on the VMEbus will be examined in section 3.5. Details on how the VIC and the 68020 interact to perform DMA transfers will be covered in section 3.8.

On the local bus side, the 68020 processor and a small amount of RAM and ROM memory used for its operation are found. The 68020 determines the "personality" of the local bus: its operation will be shown, as well as the bus-control logic which is required to arbitrate between the 68020 and the VIC for accesses to the local bus. The interface to the rest of the convolution processor is also found there, namely the input and output FIFOs and a control register. This register is used by the firmware running on the 68020 to control the convolution array: implementing this control in software yields greater flexibility and further decouples the design of the DMA engine from that of the rest of the system.

3.4 Principle of Operation

The basic principle of operation is that when the host CPU wants to perform a convolution, it sends a high-level command to the convolution processor. This is

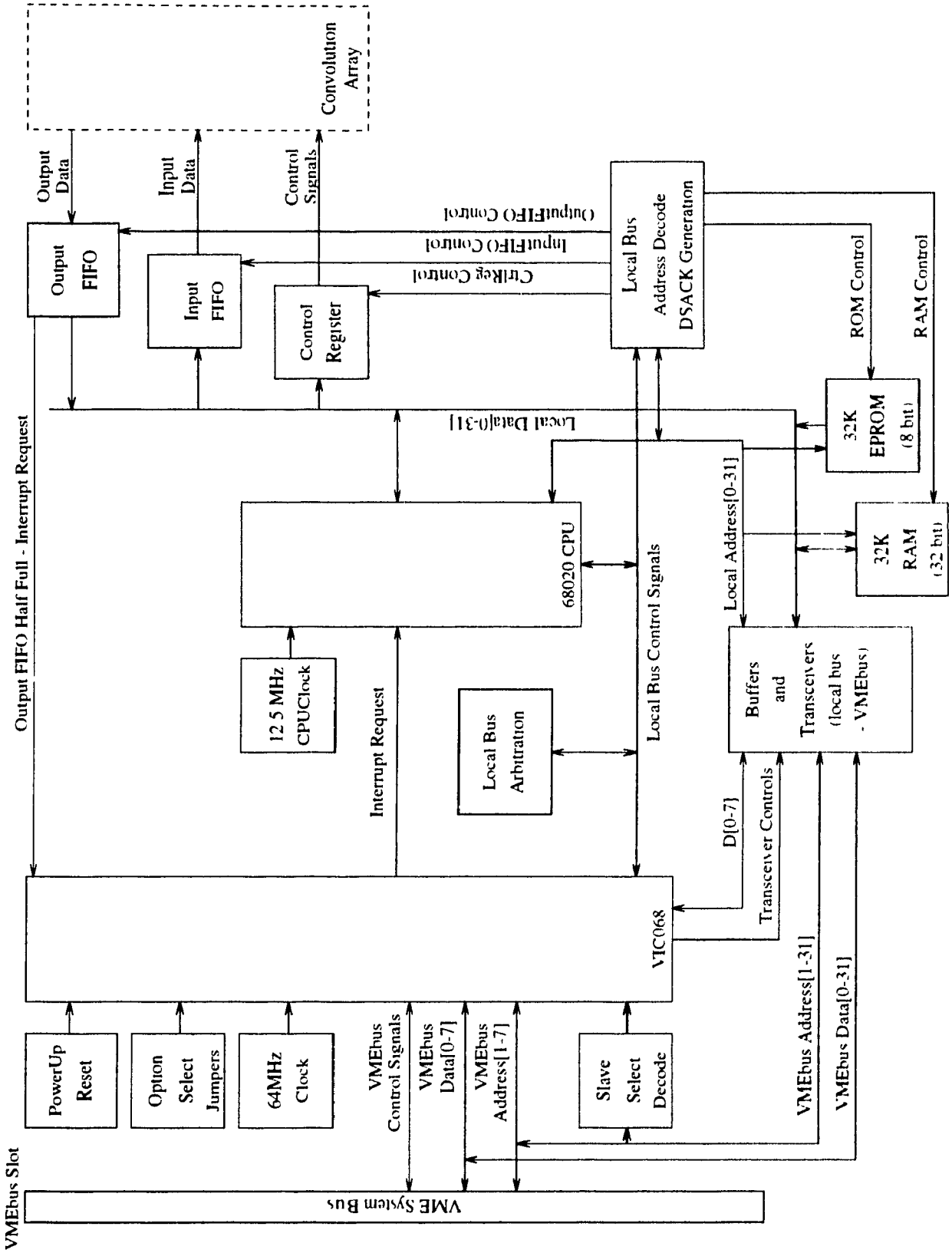

Figure 3.1: Convolution Processor Circuit Block Diagram

done by writing a command into a set of interprocessor communication registers on the VIC which are visible both from the VMEbus and the 68020 local bus. The VIC can then interrupt the 68020 to signal it that a command has arrived. The 68020 reads the command and performs all necessary initialization. In particular, it programs the VIC which will be responsible for performing the DMA transfers from host memory into the input FIFO and from the output FIFO back to host memory. The VIC can become bus master on both the VMEbus and the local bus and can transfer up to 256 bytes of data without external assistance: the 68020 assists it by keeping track of how many such transfers are required and by initiating these transfers.

The inclusion of a local CPU means that most of the complexity of controlling the convolution processor can be implemented in software. Furthermore, the convolution processor is able to respond to high-level commands from the host CPU. Apart from performing convolutions, there are commands to specify the mode of operation and to select the kernel coefficients to be used. The semantics of the software interface presented to the host CPU will be described in section 3.11.

3.5 VMEbus Interface

The convolution processor must be able to interface to a host CPU over a VMEbus. Revision "C" of this standard is documented in [VMEbus, 1982]. A complete description of the operation of this bus is beyond the scope of this thesis: suffice it to say that using a standard interface ASIC such as the VIC hides a lot of the details from the hardware designer. In particular, all of the VMEbus control signals connect directly to pins on the VIC without the need for any kind of buffering or glue logic: this greatly reduces board area requirements, as well as the risk of a design error or of a marginally successful interface implementation which might work with some VMEbus boards and not others. This used to be a common problem when each VME vendor implemented interface circuitry using custom

System Control	SYSCLK
	ACFAIL*
	SYSFAIL*
	SYSRESET*
Bus Arbitration	BR[0-3]*
	BGIN[0-3]*
	BGOUT[0-3]*
	BBSY*
	BCLR*
Interrupts	IACK*
	IACKIN*
	IACKOUT*
	IRQ[1-7]*
Read/Write	AS*
	LWORD*
	DS0*
	DS1*
	WRITE*
	DTACK*
	BERR*
Address	A[00-31]
Address Modifier	AM[0-5]
Data	D[00-31]

Table 3.1: VMEbus signals

logic. Table 3.1 lists the VMEbus signals. The following subsections will cover these signals in further detail and describe the hardware interface presented to the VMEbus by the convolution processor.

3.5.1 Master Interface

VMEbus Arbitration

VMEbus boards can either be bus masters or bus slaves. A bus master initiates transfers (either reads or writes), whereas a slave can only respond to an externally-generated transfer. Some buses such as the Industry Standard Architecture (ISA)

bus used in IBM PC-compatible systems have a fixed bus master (actually, this is an over-simplification: it is possible for an expansion board to take control an ISA bus away from the CPU, but this support is primitive at best). The VMEbus allows any board on the bus to become the bus master. It has 4 Bus Request lines: **BR0***, **BR1***, **BR2*** and **BR3***. These are open collector signals which are shared by all of the potential bus masters. When a bus master wishes to take control of the bus, it asserts one of the Bus Request lines by pulling it low. One of the boards on the VMEbus (usually in the first slot) is configured to be the system controller. When it senses that one of the **BR*** lines is low and that the VMEbus is no longer busy (this is signaled by the fact that the Bus Busy signal **BBSY*** is not being driven), it acknowledges the bus request at a given level by asserting the corresponding Bus Grant signal **BGOUTx***. These granting signals are daisy chained from one board to the next, where the signal enters the board via the **BGINx*** pin and exits it via the **BGOUTx*** pin. Empty slots must have jumpers installed to insure the continuity of the Bus Grant chains.

When a board receives a Bus Grant signal on one of its four **BGINx*** inputs, it determines whether it has requested the bus at that priority level. If it has not, it simply passes the signal along to the next board on its **BGOUTx*** output. If it wanted the bus, it does not pass the signal along and instead drives the **BBSY*** signal low to signal its ownership of the bus. This protocol allows multiple potential masters to request the bus using the same priority level: if two such boards require assert **BRx*** at the same time, the one which is physically closest to the system controller will receive the Bus Grant first and thus take control of the bus first. When it has finished with it and relinquishes it, the system controller will find that the **BRx*** line is still being driven by the second potential bus master and will thus issue a new Bus Grant signal to it. Note that a bus master is permitted to release the **BBSY*** signal before it has completed its last bus cycle, thus allowing the bus arbitration cycle to overlap the current transfer cycle and thus reduce arbitration overhead.

The system controller determines which bus request to service next based on three different schemes. In the prioritized (PRI) arbitration scheme, the **BRx*** lines are prioritized such that the line **BR3*** has highest priority and the line **BR0*** has lowest priority. If two boards request the bus at the same time, the system controller will grant the bus to the device asserting Bus Request on the highest priority line. If a board is currently holding the bus at a given priority level and another board requests the bus at a higher priority, the system controller will assert the Bus Clear **BCLR*** signal to tell the former device to relinquish the bus as soon as possible (although there is no absolute limit to the amount of time a device has to relinquish the bus: it could in theory ignore the **BCLR*** signal).

In the round robin select (RRS) scheme, the system controller assigns the highest priority to each of the 4 Bus Request lines in turn: when a request has been serviced on the highest priority line, it assigns the highest priority to the next one in circular fashion. This ensures a somewhat fairer allocation of bus bandwidth when several boards are capable of becoming bus master in the system. On the other hand, it is usually desirable to assign absolute priorities: for instance, a board which accepts input at high speed and has little buffer space should have a higher priority than one which can afford to wait much longer until it gets access to the bus.

Finally, in the single level (SGL) arbitration scheme, all the boards in the system use the **BR3*** line to request the bus: priority is thus based solely on the proximity of the board to the system controller. This scheme is used on the backplane of the Sun 3/160 system [Sun, 1989a]. The VIC can be programmed to issue Bus Requests on any one of the lines. It can also be configured to act as a system controller supporting any one of these arbitration schemes by permanently asserting its **SCON*** input: this feature is not used on the convolution processor.

Address Size	Operation Type	AM[5:0]
32-Bit Addressing	User Data	0x09
	User Code	0x0A
	Supervisory Data	0x0D
	Supervisory Code	0x0E
	User Block	0x0B
	Supervisory Block	0x0F
24-Bit Addressing	User Data	0x39
	User Code	0x3A
	Supervisory Data	0x3D
	Supervisory Code	0x3E
	User Block	0x3B
	Supervisory Block	0x3F
16-Bit Addressing	User Access	0x29
	Supervisory Access	0x2D

Table 3.2: Address Modifier Values

VMEbus Read/Write Cycles

Once a board has become bus master, it can initiate read and write cycles. It first drives the desired address onto the 32 address lines **A[00-31]**, as well as the 6 address modifier bits onto the **AM[0-5]** lines. The valid address modifier values are listed in table 3.2. They are used to indicate in which address space the transfer is to occur. Note that some of the address spaces do not use all of the 32 address lines. In particular, VMEbus boards which only have a P1 connector (instead of a P1 and P2 connector) only have access to 24 bits of address and 16 bits of data: these boards are known as A24D16 boards. It was decided to implement an A32D32 interface for highest performance and generality, but the VIC can still interface to systems with narrower address and data paths. The **LWORD*** line is asserted to indicate a 32 bit transfer: it it is not, a 16-bit transfer is being requested. If a write is to be performed, the data is put on the **D[0-31]** lines (or on the **D[0-15]** lines for a 16-bit transfer) and the **WRITE*** signal is asserted.

Once all of these signals have been driven and are stable, the Address Strobe **AS*** signal is asserted. Typically, a slave interface address decoder will have already

decoded the address of its module off the address bus lines and will use the **AS*** assertion to begin the transfer. Once it has completed the transfer, the slave module asserts the data acknowledge **DTACK*** line to signal completion to the master. If this was a read cycle, the master can then read the data off the **D[00-31]** data lines. If for some reason the slave could not complete the transfer (for instance, the master tried to address an invalid region of the address space of the slave, or it tried to perform a 32-bit transfer to a device which only has a 16-bit interface), it will instead assert the bus error **BERR*** signal to notify the master that the transfer could not be completed successfully. It is then up to the master to decide what to do (typically, an exception would be raised and signal would be sent to the process which attempt the transfer).

Convolution Processor Master Cycles

The convolution processor DMA engine becomes bus master during DMA transfers to and from the memory of the host CPU. Details of how the local 68020 interacts with the VIC to control these transfers will be presented in section 3.8. For now, suffice it to say that when the 68020 wants to initiate a DMA transfer between one of the FIFO memories and the host CPU memory, it attempts a read or write operation to an address which the on-board address decoding logic maps onto the VMEbus address space. Once this is detected, the VIC attempts to gain control of the VMEbus using the Bus Request/Bus Grant protocol outlined in section 3.5.1. Once it has done that, it then takes control of the local bus away from the 68020 and begins the transfer directly between the host CPU memory and the input or output FIFOs. The arbitration scheme for the local bus is explained in section 3.7.2. The VIC is able to transfer up to 256 bytes of data on its own (i.e. 64 long word transfers), after which it relinquishes control of the local bus to the 68020 which can schedule the next transfer. Note that this model assumes that the host CPU implements a slave interface which allows another processor access to its memory: this is not an unreasonable assumption since most VME-based disk or network controllers have

bus-master interfaces and access buffers directly in host CPU memory.

3.5.2 Slave Interface

Although a slave interface to the convolution processor was not strictly required, it was implemented since most of the required functionality is built into the VIC: furthermore, this feature can be used in a few cases, and increases the generality of the design. The slave select decoder is implemented in the traditional way with PALs and user-setable jumpers which allow the selection of different address ranges. The VIC presents two distinct slave interfaces to the VMEbus. First, it responds to slave A32 transfers (i.e. in the full 4 Gb address space) which map into the local RAM of the convolution processor. A PAL looks at the **A16** to **A31** address lines, which yields a decoding granularity of 64K: this is more than what is needed since there are only 32K bytes of local RAM memory which must be made visible to the host CPU. A bank of 4 DIP switches allows the selection of 4 possible base addresses which means that the 64K window can be mapped at 4 different positions in the 4 Gb A32 VME address space. In order to keep the address decoder as simple as possible (in effect, allow it to fit in a single 20L8 PAL device), these 4 base addresses are hard-coded in the PAL equations. Table 3.3 lists the DIP switch settings and the corresponding base addresses: if none of these are usable in the target VME system, a new address decoding PAL will have to be programmed with a different set of base addresses. Note that the slave interface only has access to the first 64K of the 256K local bus memory map: this is desired, since the host CPU should not try to read/write directly to the board control registers or the input and output FIFOs, and accesses to the VIC internal registers is not allowed from the VMEbus side (section 3.6.2 covers the local bus memory map in greater details). Also note that the slave access decode signal from the PAL is fed to the **SLSEL1*** input of the VIC: there also exists another slave select input, **SLSEL0***, but this input is defective on the revision of the chip used in the system and is thus

S2-4	S2-3	S2-2	S2-1	Base Address
0	0	0	1	0x1FF0xxxx
0	0	1	0	0x5FF0xxxx
0	1	0	0	0x9FF0xxxx
1	0	0	0	0xDFF0xxxx

Table 3.3: Slave Select Base Address

strapped high.

3.5.3 Inter-Processor Communication Registers

The VIC has eight Interprocessor Communications Registers (ICRs). These are accessible from the VMEbus without requiring the VIC to become local bus master, and are accessible from the local bus without requiring VMEbus arbitration. Five of these registers are available for general-purpose use. Furthermore, the VIC has four Interprocessor Communications Global Switches (ICGSs) and four Interprocessor Communications Module Switches (ICMSs). In all cases, these facilities are accessed when the inter-processor communications facilities select **ICFSEL*** input of the VIC is asserted and the address of the register is specified using the VME **A[5-1]**, **LWORD***, **DS1*** and **DS0*** addressing inputs: **ICFSEL*** is decoded in the A16 VMEbus address space (i.e. the short address space). A single 20L8 PAL device decodes this address space: based on the settings of the S1 DIP switches, it decodes a 64-byte memory region which can be based at one of seven different base addresses. These base addresses are hard-coded into the PAL equations and can be changed if none of the pre-defined regions are available in the host system. Table 3.4 lists the current values for these base addresses. On the local-bus side, these registers are addressed in the same way as the other VIC internal registers (see section 3.6). These registers will be used in the following way: when the host CPU wishes to instruct the convolution processor to perform an action, it will write a 5-byte message into the IPC registers: this message will take the form of a 1-byte

S1-7	S1-6	S1-5	S1-4	S1-3	S1-2	S1-1	Base Address
0	0	0	0	0	0	1	0x1FF0xxxx
0	0	0	0	0	1	0	0x5FF0xxxx
0	0	0	0	1	0	0	0x9FF0xxxx
0	0	0	1	0	0	0	0xDFF0xxxx
0	0	1	0	0	0	0	0x9C00
0	1	0	0	0	0	0	0xBC00
1	0	0	0	0	0	0	0xDC00

Table 3.4: Interprocessor Registers Base Address

opcode and a 4-byte pointer to an optional parameter block somewhere in either host or local memory. It will then write to one of the interprocessor communication switches, which is programmed to generate an interrupt to the local 68020. The local CPU can then read the command to be performed from the registers. When it has finished its task, it will write the result code back into the registers and generate a VMEbus interrupt to signal the host CPU that the operation has been completed and that a completion status is available. Thus neither the host CPU nor the local 68020 have to wait for each other: they can proceed asynchronously from each other while awaiting interrupts.

3.5.4 Interrupt Generation

The VMEbus has seven prioritized Interrupt ReQuest lines labelled **IRQ[1-7]*** (level 7 has the highest priority). These are open-collector lines which are shared by all of the boards in the system. Whenever a board wishes to generate an interrupt, it asserts the corresponding **IRQx*** line. In a manner very similar to the bus master arbitration scheme described in section 3.5.1, a board in the first slot acts as system controller: when it detects an interrupt request, it asserts its Interrupt Acknowledge **IACK*** output. It also asserts its **IACKOUT*** output, which is connected to the **IACKIN*** input of the its neighbor. Finally, it puts the encoded level of the interrupt it is responding to on the **A3-A1** address lines. When the interrupter sees **IACK***

and **IACKIN*** asserted, it compares the encoded interrupt level on the address lines with the level of the interrupt it has generated. If they match, it drives a status ID value onto the low **D7-D0** data lines. If they don't match (or if a board gets **IACKIN*** and it has not generated an interrupt), the board propagates the signal via its **IACKOUT*** output to the **IACKIN*** input of the next board in daisy-chain fashion. Empty slots must have a jumper installed to insure the continuity of the chain.

Interrupters can release the **IRQx*** line either when they get the interrupt acknowledge (this scheme is known as Release On AcKnowledge, ROAK), or they can wait for the system controller to read a status register (Read On Register Access, RORA). When acting as system controller, the VIC can deal with both types of interrupters: this capability is unused in this system, since the board will not be the system controller. Rather, the VIC will be used to generate interrupts. Its internal registers are programmed to specify which interrupt line to use to generate interrupts for the host based on the interrupt levels used by the other board in the system. Note that several boards can share an interrupt line since these are level-triggered: the board which is geographically closest to the system controller will have higher priority in that case. Interrupts will be used to signal the host that the convolution processor has completed the requested action, as explained in section 3.5.3.

3.6 Local CPU Bus

3.6.1 Local Bus Structure

The local bus is simply that of the Motorola 68020 CPU, which minimizes the amount of glue logic since the VIC is designed to interface directly to such a bus. The 68020 has 32 address lines and 32 data lines. It supports virtual memory

through an external 68851 Memory Management Unit which is incorporated into the 68030. Since a 68020 is used as a control processor, the lack of memory management actually simplifies the task at hand. Real addresses are used throughout the board. Host memory is also accessed using real addresses in most cases, which means that the operating system running on the host needs to be able to lock the source and destination image buffers into contiguous physical memory. Some Sun computers implement a scheme called DVMA (Direct Virtual Memory Access) [Sun, 1989b] which allows addresses coming from the VMEbus to be mapped into virtual addresses in host memory: this relaxes the constraint that the image buffers be mapped into contiguous memory regions, although it is still desirable to lock these buffers into physical memory to prevent a large performance loss if pages are not resident when they are accessed and need to be paged in from disk.

The 68020 local bus is addressable in byte (8 bits), word (16 bits) or long word (32 bit) increments. There are no restrictions on data alignment, or on the size of memory devices which can be attached to the bus. For instance, an 8 bit wide memory (such as the EPROM which holds the bootstrap code) can be connected to the bus. The 68020 encodes the size of the data transfer onto its **SIZ1** and **SIZ0** outputs, and encodes the operand alignment on the two low-order address lines (**A1-A0**). The decoding logic for the addressed module looks at these inputs and, based on the size of the port, it signals how many bytes of the transfer it was able to accept/deliver when it acknowledges the completion of the transfer using the **DSACK1*** and **DSACK0*** lines (note that, as for the VMEbus, the 68020 bus uses an asynchronous bus protocol). In the worst case of an unaligned long-word transfer to a byte-sized port, a single read or write instruction can generate four separate bus cycles. Although this yields a lot of flexibility for the programmer, it adds a lot of complexity to the 68020 bus interface circuitry. Current RISC architectures such as the MIPS R3000/4000 are much less forgiving: they impose strict requirements on operand alignment, and compilers for these architectures will frequently pad C language structures in order to align structure elements

on word or long-word boundaries [MIPSASM, 1987]. Furthermore, it has been suggested that portable C code should be debugged on machines which impose strict alignment constraints, since such code will then work on machines which are more forgiving. Further details on the operation of the 68020 bus can be found in section 7 of [MC68020, 1989].

3.6.2 Local Bus Memory Map

Table 3.5 describes the on-board memory map for the 68020 local bus. The total address space is 256K, replicated throughout the 68020 4Gb physical address space. The 32K byte EPROM is mapped at address 0 since when the 68020 CPU first powers up, it reads its Reset Initial Interrupt Stack Pointer from address 0x00000000 and its Reset Initial Program Counter from address 0x00000004 The former is initialized to the top of the 32K static RAM (mapped next from 32K to 64K) since the 68020 stack grows downwards. The latter is initialized to the beginning of the power-up code sequence in the EPROM. Note that the 68020 attempts to perform 32-bit reads for these values: since the EPROM only acknowledges an 8-bit transfer, the 68020 must then perform 3 extra reads to get the remainder of the operand Furthermore, since the EPROM is a relatively slow device with an access time of 150ns, the bus control logic which generates the **DSACKx*** signals inserts a delay (wait state) whenever the EPROM is accessed. For all of these reasons, after the initialization of the board is completed, the remainder of the code (in particular the code which must execute rapidly during DMA transfers) is copied into the much faster 32 bit wide static RAM where the 68020 will be able to access it at full speed

The static RAM is composed of 4 8Kx8 devices which are connected to the 32 data lines to form a 32-bit wide path. These memories have an access time of 55ns, which means that the local bus control logic can acknowledge transfers as soon as they are decoded, thus allowing the 68020 to operate at full speed when accessing RAM. Both the EPROM and the RAM are accessible from the VMEbus during VME

Function	LA17	LA16	LA15	Comment
Boot EPROM	0	0	0	32K length
Local RAM	0	0	1	32K length
VIC registers	0	1	0	256 locations, mirrored 128 times
Board Control Register	0	1	1	1 location, mirrored 32K times
Input FIFO	1	0	0	1 location mirrored 32K times, write cycles
Output FIFO	1	0	0	1 location mirrored 32K times, read cycles

Table 3.5: Local Bus Address Space

slave cycles. Section 3.7.2 explains of how the local bus is arbitrated between the 68020, the VIC and the VMEbus.

The 58 VIC internal registers are mapped from 64K to 96K. Only the lower 8 bits of the address is significant, so the registers are mapped 128 times within this 32K addressing region To signal a register access, the bus control logic asserts the VIC Chip Select **CS*** input. Although the VIC supports other access modes, the current design always uses longword accesses aligned on longword boundaries (i.e. address lines A1 and A0 are both zero). The VIC will acknowledge a 32-bit transfer, even though only the 8 least significant bits of the transfer are relevant (the registers are all 8 bits wide). Note that the registers are only accessible from the local bus. hence the need for a local CPU if only to initialize the VIC after reset. Section 3.9 will go into further details with respect to the programming and operation of the VIC.

A 32-bit wide, write-only control register is mapped from 96K to 128K (the use of general-purpose PAL devices for address decoding prevented a finer grain of address decoding). This register is used to control the rest of the convolution processor: for instance, some of its bits signal the size of the data in the input FIFO. The register is write-only since this was the easiest way to implement it. A copy of the value of this register is kept in one of the 68020 internal registers at all times. Thus when a single bit needs to be set or cleared, a masking operation is performed on that register and the new value is then written to the external control register. In this way, the logic needed to control the rest of the convolution processor

can be implemented entirely in software: this design approach was chosen to increase modularity and to allow different people to work on the components of the system with maximum independance. The devices used to implement this register (74F374s) are fast enough to allow full-speed access by the CPU.

The input and output FIFO memories which are connected to the convolution array are mapped from 128K to 256K. They can be viewed as a single 32-bit wide port replicated 128K times in that address space. A write cycle will write into the input FIFO, a read cycle will read from the output FIFO. Four 2Kx9 devices are used to implement each of the two FIFO memories (only 8 bits out of the nine are used), which means that there is 8K bytes of buffering both at the input and at the output of the convolution array. The devices have an access time of 65ns, which means that they can be accessed at the full speed of the local bus In the worst case of 64-bit floating point input to and output from the array, a transfer rate of 12.5 Mb/sec is required to prevent the convolution array from stalling. This also means that given an input FIFO full of data, the array will stall if the convolution processor is locked out of the bus for more than 655 microseconds, which is quite a short bus period. On the other hand, it is anticipated that in most cases the system will be running with 8-bit input and output, which yields a much more comfortable buffering interval of 5.25 msec. Furthermore, the devices were selected a couple of years ago: since then, the same manufacturer (Cypress) has come out with pin-compatible devices with up to 32Kx9 capability· it would thus be easy to increase the capacity of the FIFOs if needed.

3.6.3 Local Bus Control Logic

In figure 3.1, the block labelled "Local Bus Address Decode DSACK Generation" implements most of the local bus control logic. Section 3.6.2 listed the contents of the local bus memory map: figure 3.2 shows how this is implemented Two 20L8 PAL devices are used to decode local address lines **LA15-17**, giving the 256K

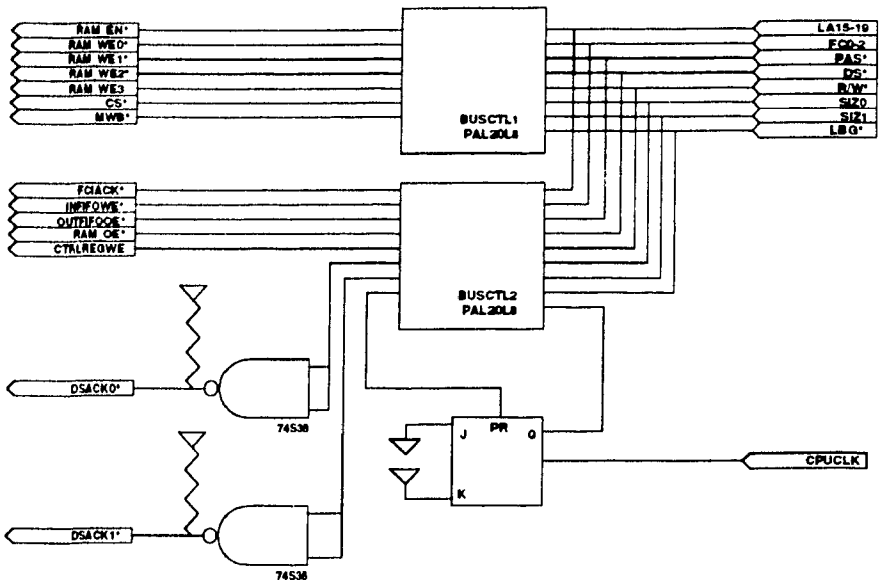

Figure 3.2: Local Bus Control Logic

total address space of the on-board bus. The **A16-A19** 68020 address lines are also used to indicate which address space the CPU is accessing during interrupt acknowledge cycles. Note that it would not have been possible to save wiring by leaving the **A20-A31** address lines unconnected since these are used to specify the VMEbus base address for DMA transfers, as explained in section 3.8. Since a larger address space was not needed, it was decided that an address decoding scheme using a single PAL device (and hence introducing a single device delay) would be preferable in order to gain more performance.

The local bus static RAM is controlled by the signals **RAM_EN***, **RAM_OE*** and **RAM_WE0-3***. The former is used to select the RAM when the proper address range is present on the local bus address lines. This signal is gated by the **PAS*** or Physical Address Strobe signal which is generated either by the 68020 or by the VIC, depending on which device is currently local bus master. Other terms are used to restrict access to either 68020 accesses or VMEbus slave accesses. Full details of the PAL equations are beyond the scope of this document.

The **RAM_OE*** is used to signal to the RAM whether the cycle is a read or a write: it is simply derived from the local bus **R/W*** line. Note that all READ accesses

to the RAM are 32 bits wide and will be signalled as such by the transfer acknowledge generation logic. Finally, the **RAM_WE0-3*** signals are used to individually select each of the four 8-bit devices which make up the 32-bit wide memory. This complexity is required by the 68020 bus protocol which allows unaligned writes.

The **ROM_EN*** signal is used to decode read cycles for the 32K EPROM which contains the startup code. Since the EPROM interface is only 8 bits wide, the EPROM is only used for power-on code: the necessary instructions will then be copied to RAM from which the 68020 will be able to execute at full speed The **CTRLREGWE** is used to latch the current value of the local bus data lines into the Array Control Register: this register is write-only (in order to minimize the amount of logic) and a software image is kept in a 68020 register to perform the necessary masking operations.

The **CS*** signal is generated when the 68020 tries to access the portion of the address space into which the VIC control registers are mapped When the 68020 attemps a transfer in address space 0x03 (as indicated by its Function Code **FC0-2** outputs), this is used to assert the **MWB*** signal which signals the VIC that the 68020 wishes to perform a VMEbus master cycle: this is further explained in section 3.7.2. The **FCIACK*** signal is generated when the 68020 **FC0-2** Function Code lines indicated a CPU Space access and the 68020 is performing an Interrupt Acknowledge cycle. this signal is used as an input to the VIC, which requires it in order to perform its function as on-board interrupt controller.

The **INFIFOWE*** and **OUTFIFOOE*** signal are used to respectively write the current value of the local bus data lines into the Input FIFO or remove a word from the output FIFO and put in on the data lines. These two signals can be generated either during VMEbus DMA transfer cycles or during normal cycles initiated by the 68020: this allows the local CPU to initialize the input FIFO with the proper amount of 0 values required to "fill the pipeline", as well as clear the output FIFO of the initial invalid results which are generated before the array has been completely

initialized.

Finally, the **DSACK0*** and **DSACK1*** signals are used to acknowledge the completetion of all local bus cycles. Most devices on the bus are fast enough not to required any "wait states" (of course, a wait state is somewhat of a fuzzy concept in an asynchronous bus architecture): the 68020 allows accesses to fast devices to be acknowledged in advance, and in this case this is done by simply NANDing all of the individual device enable signals. This is possible in part since the 68020 is running at a fairly slow speed of 12.5 MHz: a faster CPU clock speed might have required the insertion of additional delays. The only exception are accesses to the EPROM, which is a fairly slow device with an access time of 150ns. A JK latch driven by the CPU clock is used to add an extra clock cycle of delay, which is sufficient to insure that the outputs of the EPROM have stabilized on the local bus data lines. Since the **DSACK0*** and **DSACK1*** signals are open-collector, 74S38 open-collector NAND gates configured as inverters are used to drive these signals: thus in the case of VIC register or VME bus accesses, the **DSACK*** signals are generated by the VIC and not by this control logic.

3.6.4 Local Bus Arbitration, Deadlock Resolution and Reset Logic

The block labelled "Deadlock Arbitration Bus Control" in figure 3.1 implements the remaining local bus control functionality not covered in the previous section. Combinatorial logic is handled by a 20L8 PAL device whereas sequential logic is housed in a 16V8 Lattice GAL device (a GAL is basically an electrically reprogrammable PAL). Figure 3.3 is a block diagram of this circuit.

When the VIC senses a VMEbus **BERR*** signal in response to an attempted VMEbus transaction, it will assert the local bus **LBERR*** signal: note that **LBERR*** is also an input for the VIC to allow it to detect a bus error for any local bus transactions and pass this signal on to the VMEbus if required. The 20L8 PAL

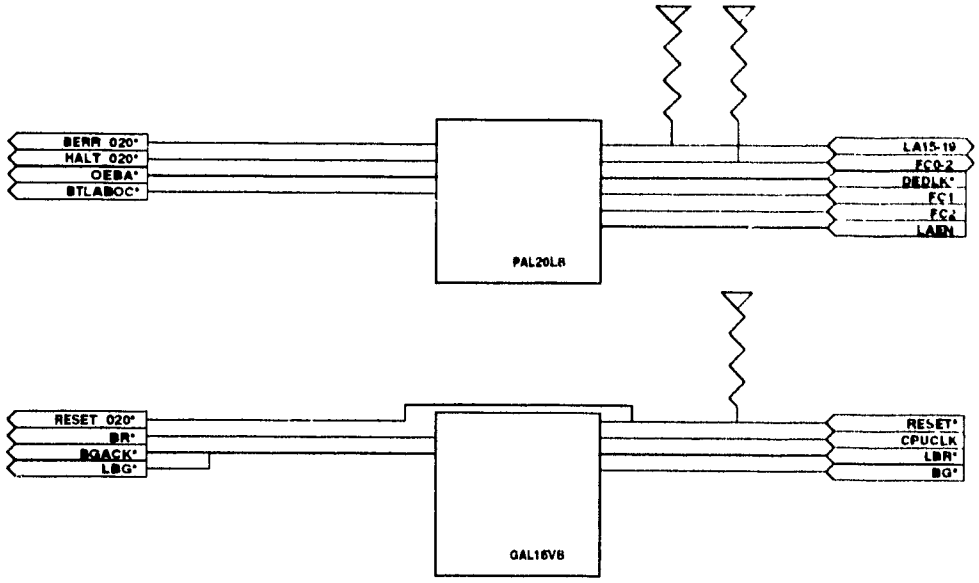

Figure 3.3: Local Bus Arbitration, Deadlock Resolution and Reset Logic

passes this signal on to the 68020 as **BERR_020***, since for the 68020 this signal is an input only. Similarly, the VIC **HALT*** bi-directional signal is passed on as **HALT_020*** to the corresponding 68020 input. When the VIC detects a VMEbus slave access request concurrent with a local CPU request for the VMEbus, it asserts its **DEDLK*** output: the bus control logic asserts both **BERR_020*** and **HALT_020***, which signals the 68020 that it should back off from the bus cycle it is attempting and should retry it when these signals are no longer asserted. Note that since VMEbus slave cycles (where the host CPU is trying to access the on-board memory of the convolution processor) only occur during the initialization phase, it is unlikely that such deadlock situations will ever occur. Nevertheless, since the VIC already offers this functionality, it was included into the design. The 20L8 also generates a couple of other signals, **OEBA*** and **BTLABOC*** which are used to control the transceivers and address latches during VIC-controlled VMEbus DMA transfer cycles.

The 16V8 implements a simple state machine which is used for arbitration of the local bus between the 68020 and the VIC. When the VIC requires the local CPU bus, it asserts its Local Bus Request **LBR*** signal. In response, the state machine asserts the 68020 Bus Request **BR*** signal. When the 68020 detects this and completes its

currently executing bus cycle, it asserts its Bus Grant **BG*** output. This is passed on to the VIC Local Bus Grant **LBG*** input. At the same time, the 68020 Bus Grant ACKnowledge **BGACK*** signal is asserted and its **BR*** input is negated. At that point, the VIC owns the local bus When it no longer requires it, it will negate its **LBR*** output. the arbitration logic then negates the 68020 **BGACK*** input and the 68020 regains ownership of the local bus.

3.7 VMEbus-Local Bus Interface

3.7.1 Bus Transceivers

Although the VMEbus is quite similar to the local bus of a Motorola processor, there is nevertheless a significant amount of interface circuitry which must be added between the two. For one thing, the bus drivers of a 68020 are not powerful enough to drive the VMEbus lines directly, and thus must be buffered. Also, the 68020 local bus is more lenient about unaligned transfers than the VMEbus is. Finally, the two buses must be isolated from each other to allow concurrent operation, but must also be connected together when required. For all these reasons, a series of transceivers are used to connect the two.

First of all, the lower 8 VMEbus data lines (**D0-7**) and the lower 7 address lines (**A1-7**) go directly through the VIC, which handles all of the necessary buffering and arbitration. The 24 upper address lines are connected through 74F543 octal latching transceivers. The latching capability is required to support write posting, a technique where the local CPU can do a single write to the VMEbus without having to wait for the completion of the write on the VMEbus (the address and data having been captured in the latching transceivers). Three octal latches (74F373) with their inputs connected to the **LD8-31** local data lines and their outputs connected to the **LA8-31** address lines are used to implement block transfers. The VMEbus **D8-31**

data lines are connected to the local **D8-31** data lines also using 74F543 octal latching transceivers. Additionally, 74F245 octal transceivers are used on the local side to implement byte-swapping functionality required to allow un-aligned transfers. All in all, 13 latches and transceivers are required to implement the interface between the VMEbus and the 68020 local bus, which requires a fair amount of board area due to the use of large DIP (Dual In-line Package) devices. Fortunately, the VIC provides all of the control signals needed to drive the control inputs of these devices, so this minimizes the amount of extra logic required. Since this design was completed, a companion device to the VIC called the VMEbus Address Controller (VAC) was introduced: the VAC incorporates all of the address bus transceivers and latches, as well as address decoding circuitry, two serial ports and other useful features. Had the VAC been available at the time of the design, it would have been used.

3.7.2 Local Bus Arbitration

Slave Accesses from the VMEbus

When a master on the VMEbus accesses the address range decoded by the slave access decoder, this asserts the **SLSEL1*** input on the VIC. This signals the VIC that the external bus master wishes to access resources which are on the local CPU bus. The VIC then asserts its Local Bus Request (**LBR***) which is connected to the Bus Request (**BR***) input of the 68020. When the 68020 senses its **BR*** input go low, it completes the current bus cycle, tri-states all of its outputs which control the local bus and then asserts its Bus Grant (**BG***) output. The arbitration control logic uses this signal to generate the Local Bus Grant (**LBG**) signal to the VIC, the Bus Grant Acknowledge (**BGACK***) signal to the 68020 and to negate (**BR***) to the 68020 The VIC interprets the assertion of **LBG*** as the signal that it now owns the local bus: it then connects the local bus address and data lines to the VMEbus address and data lines through the control inputs of the address and data transceivers.

The local bus decoding logic then decodes the slave access to the proper on-board module (in this case, only the EPROM and RAM are accessible). When the on-board acknowledges **DSACK0-1*** are generated by the local bus control logic, the VIC senses these and generates a VMEbus **DTACK*** to signal the VMEbus master that the transfer has been completed. The VIC then deasserts **LBR***, which causes the local bus arbitration logic to deassert **BGACK*** to the 68020, which takes back control of the local bus.

Master Accesses to VMEbus

When the 68020 wants to access a memory location on the VMEbus, it first loads the function code 0x03 into either the Source Function Code (**SFC**) register or the Destination Function Code (**DFC**) registers using the Move Control Register (**MOVEC**) instruction. It then issues the Move Address Space (**MOVES**) instruction which transfers data between an internal 68020 register and a memory location in the address space specified by the code previously loaded into **SFC** or **DFC**. Address space 0x03 is reserved by Motorola for user expansion, and in this case the on-board address decoding logic maps it onto the VMEbus A32 address space. The VIC Module Wants Bus (**MWB**) is then asserted, and the VIC proceeds to become bus master on the VMEbus (if it does not already own it) using one of the arbitration protocols outlined in section 3.5.1. Once it has obtained ownership of the VMEbus, it connects the local address and data lines to the VMEbus address and data lines. It can derive the values to be driven onto the VMEbus Address Modifier (**AM0-5**) lines based on the 68020 **FC0-2** outputs, or since in this case **FC0-2** will always have value 3 for VMEbus accesses, it can take this value from a previously programmed internal register.

Once the VIC has received **DTACK*** from the VMEbus module to acknowledge the transfer, it asserts **DSACK0*** and/or **DSACK1*** to signal the 68020 that the transfer has been completed. At that point, it can either relinquish control of

the VMEbus or keep it in anticipation of a next cycle based on the way it has been programmed. Note that the VIC does not need to acknowledge to the 68020 that it has acquired control of the VMEbus. to the 68020, a read or write cycle over the VMEbus is completely transparent (although much longer than an access to a function local to the board). If the 68020 needs to do a single write to a VMEbus module, the VIC can be programmed to implement write posting, where the values driven onto the 68020 data and address lines are captured by the latching transceivers and **DSACK0***, **DSACK1*** are returned right away to the 68020. The VIC the performs the VMEbus write cycle on its own, while allowing the 68020 to continue to issue cycles which affect only local-bus modules. If the write-posted cycle ends in a VMEbus Bus ERRor (**BERR***), the VIC will issue an interrupt to the 68020 to signal this occurrence: since this can come several local bus cycles after the write was posted, the 68020 software must be careful in keeping track of which posted writes are outstanding and might possibly be signaled as having terminated with an error.

3.8 VMEbus DMA Transfers

As outlined previously, most of the work required to transfer data from host memory to the input FIFO and from the output FIFO back to host memory is done by the VIC, with the assistance of the 68020. After a period during which the convolution array has to be filled, results begin to come out of the array into the output FIFO. The Half Full (**HF***) output of the output FIFO is connected to the Local Interrupt ReQuest 7 (**LIRQ7***) input of the VIC: when this FIFO becomes half-full, the VIC will detect a HIGH to LOW transition on that input and will generate an interrupt to the 68020. The 68020 will acknowledge that interrupt by initiating a CPUSPACE cycle (i.e. a read cycle where the Function Code (**FC2-0**) outputs are all 1). The local bus control logic decodes this to assert the VIC Function Code Interrupt ACKnowledge (**FCIACK***) input. The VIC responds by driving an inter-

rupt vector onto the 8 lower data lines **D0-7** and asserting **DSACK1*, DSACK0***. The 68020 will then execute an interrupt service routine (it will fetch the address of this routine from the entry in the interrupt vector table corresponding to the interrupt vector supplied by the VIC) to start a DMA transfer to empty the output FIFO by writing the results back into host memory over the VMEbus. Although the interrupt procedure introduces a bit of delay, this is not a problem since the output FIFO is only half full when the interrupt is generated. Since the convolution array operates synchronously (i.e. every time a result is written to the output FIFO, a datum is removed from the input FIFO), there is no need to generate interrupts when the input FIFO becomes empty: all that is required is that as many operands are written into the input FIFO as are removed from the output FIFO. The 68020 computes how much data must be transfered based on the operand sizes used for the source and destination images.

The first action of the 68020 interrupt handler is to initiate a 32-bit write to the VMEbus destination address. This will cause the local bus arbitration logic to assert Module Wants Bus *MWB** to the VIC: having been properly configured beforehand, the VIC will interpret this assertion as an indication that it must perform a VME block transfer with local DMA. It will simultaneously arbitrate for control of both the on-board local bus and the VMEbus. The VIC interprets the address of the triggering write cycle as the source/destination address on the VMEbus and the data as the address on the local bus: this is made possible by the use of latching transceivers and extra latches which are used to capture these values. Since the VIC drives directly the lower 8 bits of the address and data buses on each side, it can "count" up to 256 (one of its registers is used to determine the number of cycles in a block transfer, from 1 to 64). With additional counters, it is possible to get it to perform transfers up to 64K in length without any outside intervention: unfortunately, this feature was documented as not working properly in version of the device. Again, had the VAC companion chip been available at the time of device, it would have been selected since it takes care of everything that is needed

for longer transfers.

Note that since all local transfers will be performed to the FIFOs, the data portion of that triggering cycle does not need to change. Furthermore, although the VIC will be incrementing the lower 8 address bits on the local bus, this does not cause any problems since, as shown in section 3.6.2, the FIFOs are replicated throughout a 32K section of the local bus address space. In order to simplify the control software, these 256 byte transfers will always start on 256-byte boundaries. Since this might be an unreasonnable restriction to place on the location of image buffers in host memory (it might be difficult to guarantee such aligment with certain operating systems), the first and last transfers for an image can be shorter and handled as a special case.

Once the VIC has obtained ownership of both buses, it connects the two through the latching transceivers and begins to transfer data using 32-bit wide local bus cycles and VMEbus block transfers. The length of these VMEbus bursts can be programmed to prevent other devices on the bus from being locked out for too long. Ideally, it would be desirable to be able to lower the arbitration overhead as much as possible by keeping these bursts fairly long: a certain amount of tuning of this parameter will be required for every system in which the convolution processor will be installed. If the VIC is programmed for bursts of less than 64 transfers (i.e. 256 bytes), it will relinquish the VMebus and re-acquire it between bursts. Since there is not much else for the 68020 to do during that time, the VIC will not bother releasing control of the local bus.

While the VIC has control of both the VMEbus and the local bus, the 68020 is able to keep on executing since it has a 256 byte internal instruction cache. The DMA transfer control loop is coded to fit entirely within this cache, where it will remain after the first iteration of the loop has been executed. As long as the CPU does not need to perform any external bus cycles, it can keep on executing instructions even though it has relinquished control of the bus. In this case, all it needs to do is

increment the VMEbus DMA base address by 256, decrement a counter indicating how many transfers are left in order to empty 4K of data from the output FIFO (remember that this operation is triggered by the Half Full flag on the 8K output FIFO) and initiate the next triggering 32-bit write to VMEbus address space. At that point, the 68020 will stall since it does not have access to its bus.

When the VIC has finished its 64 cycles, it relinquishes control of the VMEbus and the local bus The 68020 can then complete its stalled triggering cycle, thus starting the process over again. After 16 of these DMA transfers from the output FIFO to host memory, the control software proceeds to read fill the input FIFO correspondingly, i.e. read as many operands into the input FIFO as were taken out of the output FIFO. This might end up corresponding to differing amounts of memory based on input and output operand size: in the case where input operand size is larger than output operand size, fewer output than input DMA block transfers might be performed.

3.9 VIC Controls

The Cypress VIC-068 VME Interface Controller is an integrated VMEbus interface device which greatly simplifies the design of a master/slave VMEbus interface. Section 3.5 showed how it is used to implement the interface between the VMEbus and the 68020 local bus. this section looks deeper into its operation.

The VIC is controlled by 58 byte-wide registers which must be programmed from the local-bus side· thus although it might be possible to build a state machine to do this, the VIC is really meant for applications where a CPU is present on the board. Since every aspect of its operation can be configured in software, it is very flexible and can be configured for most applications. Each group of registers will be looked at and their use with respect to this design will be discussed.

3.9.1 Interrupt Registers

The VIC can act as an interrupt generator/controller for both the local CPU bus and the VMEbus. It can receive interrupts from the following sources:

1. interprocessor communication registers (see further)
2. *ACFAIL** (power fail) on the VMEbus
3. *SYSFAIL** (system failure) on the VMEbus
4. arbitration timeout
5. failure of a posted write cycle
6. handshaking with a VMEbus interrupter
7. 7 local interrupt inputs
8. DMA completion

Since the board will not serve as system controller, sources 2, 3 and 6 are not used Posted write cycles will not be used, and the 68020 does not need to be notified of DMA completion (as explained in section 3.8). Arbitration timeout interrupts will be handled as an error condition which might lead to the aborting of the current operation. The interprocessor communication registers will be used by the host CPU to transmit commands to the system· when the VIC detects that host has written to these registers, it will interrupt the 68020 to signal it that a command is waiting for it. One of the 7 local interrupt inputs will be used to generate interrupts to the 68020 triggered by transitions on the Half Full output of the output FIFO (the polarity and edge-triggered versus level-triggered nature of these inputs is also configurable). Finally, the 68020 will be able to generate interrupts to the host CPU by writing values into the proper interrupt control register of the VIC to signal the completion of a requested operation

3.9.2 Inter-processor communication registers

As outlined in section 3.5.3, the VIC has inter-processor communication registers which can be used to implement efficient protocols which do not require any of the processors involved to busy-loop waiting for the other to complete an operation. There are 5 usable 8-bit registers plus another register which can be used for semaphore functions. The host will write a 1-byte command and an optional 4-byte optional parameter buffer pointer into these five registers. It will then write to one of the four Interprocessor Communications Module Switches, which can be configured to generate an interrupt to the 68020: its interrupt handler can fetch the operation code and optional parameter pointer from the registers. A VMEbus master cycle can then be used to retrieve the parameter values at the address contained in the pointer. Note that there are also four Interprocessor Communications Global Switches which are read-only from the local bus: these are not used in the system.

3.9.3 Block transfers control registers

These registers include the Block Transfer Definition Register, which is used to enable the VIC to perform block transfers longer than 256 bytes: as explained earlier, this capability is not used in this design. The Block Transfer Control Register contains bits which are used to enable VMEbus block transfers with local DMA when Module Wants Bus *MWB** is asserted to the VIC: this is the mechanism which is used by the 68020 to start 256 byte block transfers. Another bit is used to signal the direction of the transfer, i.e from local memory to the VMEbus or vice-versa. The Release Control Register is used to set the maximum burst length, which may be shorter than the 64 cycle block transfer length: this is used in a system where other potential bus masters cannot be locked out of the bus for 64 cycles due to real time constraints (such as limited buffer space on a disk or network controller)

Finally, the Block Transfer Length Register is used to specify the number of bytes (in increments of four, since only long-word transfers are supported) to be transfered during a block move in this case, this will always be 256, except perhaps for the first and/or last transfers

3.9.4 Slave select control registers

Two registers are used to configure each of the Slave Select inputs: in this case, only the second one (*SLSEL1**) is used since in the current version of the VIC the first one does not work properly. These registers are used to set the address and data size of transfers which are supported by the slave interface, i.e. A32, D32 in this case. The timing between the assertion of *DSACK1** by the bus master and the acknowledgement of the end of the transfer by the VIC by asserting the *DTACK** signal is also configured in software: this delay is a function of the speed of the on-board devices which are accessible from the VMEbus via the slave interface and the delay introduced by the transceivers between the VMEbus and the local bus.

3.9.5 Arbitration control registers

The Arbiter/Requester Configuration register determines how the VIC will request ownership of the VMEbus. In particular, it selects which Bus Request line the VIC will be using (see section 3.5.1 for more details on VMEbus arbitration) and how "aggressive" it will be at requesting the bus (a "fairness" timer can be configured to slow down the pace at which the VIC might request control of the VMEbus). Since the board will not be used as a VMEbus system controller, none of the bus arbitration functions are used. Finally, the Release Control Register is used to determine the release protocol used by the VIC. It can support any of the protocols outlined in section 3.5.1.

3.9.6 VMEbus and local bus configuration registers

A number of registers are used to set such operating parameters as the speed of the memories on the local bus (in the Local Bus Timing Register) as well as the values of the timers used to determine error conditions such as timeouts and bus errors. Again, the VIC shows great flexibility in that most of these parameters can be changed in software, making it fairly painless to interface to a number of devices of varying speeds and capabilities.

3.10 68020 operation

3.10.1 Booting

Initialization

When the 68020 powers up, the first thing it does is a long-word read at address 0x000000, from which it reads the Initial Interrupt Stack Pointer. It then does a second long-word read at address 0x00000004 to get the Reset Initial Program Counter this value is placed into the Program Counter and the CPU begins executing instructions from there. Since the local EPROM is mapped at address 0x00000000 in local address space, both of these values are pre-programmed at the beginning of the EPROM. The Initial Interrupt Stack Pointer is initialized to address 0x0000FFFF, which corresponds to the top of the 32K local RAM address space: stacks grow downwards in 680x0 processors. The Reset Initial Program Counter is initialized to the start of the initialization routine stored in the EPROM.

The first task of the initialization routine is to push onto the interrupt stack initial values for the Status Register, Master Stack Pointer and Program Counter and execute a ReTurn from Exception **RTE** instruction to exit the reset exception

handler and return to supervisor state. The Master Stack Pointer is initialized 2K below the address of the Interrupt Stack Pointer: this reserves more than enough stack space for exception processing. Note that all of the code on the 68020 will run in supervisor or exception mode, none in user mode, since there are no other users or operating system resources to protect from the program. Finally, the 256 byte instruction cache is enabled: this will important in order to obtain maximum performance during DMA transfer cycles.

Self-Test

In order to insure that the on-board resources are operating properly, built-in self-test procedures are executed next. These verify the following operations:

- EPROM read cycles: a checksum value stored at the end of the device matches with the checksum computed by the CPU
- RAM read and write cycles: since there is only 32K of RAM, a fairly extensive test of the device can be performed
- pushing and popping on the stack
- exception handling
- reading and writing to VIC control registers
- writing to the board control register

Note that at the time the board was designed, components with support for built-in self-test where not as readily available as they are now. If this board were to be redesigned, it would use components which include JTAG boundary-scan functionality.

The rest of the board is initialized next. In particular, the control registers of the VIC are set as outlined in section 3.9. The write-only control register is initialized

to the proper value. If one of the self-tests fails, an error code is deposited into one of the VIC inter-processor communication registers: the host CPU can read this code and present to the user the reason for which the board failed its self-test

Copy to RAM

Since EPROM accesses require extra wait states to compensate for the slow speed of the device, the next step is to copy the code which is going to be used for the main loop of the control program from EPROM to RAM. Instead of trying to write relocatable code, since it is never executed from EPROM but only from its new base address in RAM, this code is assembled using its base address in RAM. Furthermore, since an extra delay in exception processing is not desirable (especially when handling interrupts generated by the status of the FIFO memories), the exception vector table is also copied to RAM and the Vector Base Register is initialized to point to the new base address of this table. Once all this is done, the board is ready to operate and accept commands from the host.

3.11 Host Software Interface

As explained in section 3.5.3, the host CPU instructs the convolution processor to perform actions by writing a 5-byte code (a 1 bytes opcode plus an optional 4 bytes operand) into the VIC inter-processor communication registers and setting one of the inter-processor communication switches. The main loop of the on-board control software sits idly awaiting interrupts generated by the IPC switch registers. Based on the opcode requested by the host, the control software executes the desired function. When it has completed the requested operation, it will instruct the VIC to generate a VMEbus interrupt to signal the host. A result code is also placed into the IPC registers: this can be used to signal abnormal completion of a requested operation.

The software recognizes the following commands from the host:

- **Load New Coefficients**: the host will have previously written into a fixed-address buffer in local RAM (using VMEbus slave cycles) the floating-point values of the coefficients to be loaded into each of the processors which make up the convolution array The optional operand is not used.
- **Load Output Converter Table**: as in the previous command, the host will have written the 4K entries which make up the output converter look-up table into local RAM. The local CPU will load these into the look-up table via the array processor control register. The optional operand is not used.
- **Set Image Source Address**: the operand contains the source address in host memory for the image.
- **Set Image Destination Address**: the operand contains the destination address in host memory for the results of the convolution.
- **Perform Convolution**: the operand contains the following fields: 2 bits each for input and output data format, specifying either 8 bit, 16 bit or 64 bit operands; 2 bits each for specifying whether the input and output data streams should be upsampled/downsampled by factors of 1, 2 or 4; 12 bits each for the x and y size of the input image. Note that the size of the delay memory circuit memories imposes a practical limit on the size of the image lines.

Note that the **Perform Convolution** operation requires that a valid set of coefficients must have been previously loaded into the array, that the output converter look-up table must have been initialized and that a source and destination address must have been specified, although it is possible for the host to issue multiple convolutions without changing these base addresses. When no upsampling is performed, it is also possible to have the source and destination addresses point to the same region in memory since the convolution processor will read the source image

before writing the results of the convolution, thus allowing in-place operations (this is obviously not possible if the input image is being upsampled, since the output data stream would quickly begin to overwrite regions of the image which would not have been processed yet).

The actual operation of the 68020 during the convolution is explained in section 3.8. The coding of this loop is quite critical, since some crucial portions must fit in the 256-byte instruction cache to allow the 68020 to continue processing (i.e. computing the base address of the next transfer) during the time it has relinquised its local bus to the VIC and be ready to start the next 256 byte DMA transfers as soon as the current one has finished

3.12 Host Software

Although the convolution processor does most of the work, a certain amount of software needs to run on the host processor in order to interface with the device. This software can be separated into three levels:

1. device driver level
2. library level
3. application level

Only the device driver is of concern here: it implements all of the functionality required to write a C or C++ application which interfaces with the convolver. A library can be used to supply higher-level functionality, such as a single *convolve()* routine. It can also be included into a library of signal processing functions. Finally, support for the convolver could be included in a signal or image processing application, especially one with a modular design. For instance, the Khoros system [Khoros, 1991] implements different operations as separate programs which

can be connected using a graphical user interface: the application which performs convolutions could be replaced by a version which knows how to take advantage of the convolver device.

3.12.1 Host Device Driver

Writing a device driver for a Unix-like operating system has often been considered somewhat of a black art only to be mastered by the most seasoned wizards. This is due in part to the design of traditional Unix systems as monolithic kernels where the device driver is a C routine which is linked to the rest of the system. Thus the only way to debug a driver is to reboot the machine with the new kernel, try out the new driver, and most likely crash the machine since the kernel operates in the privileged mode of the CPU. In most cases, the driver developper is left to scratch his head with nothing but the output of a few debugging print statements to figure out what went wrong. Fortunately, this has begun to change. Most new kernels such as those of Sun's Solaris 2 or IBM's AIX 3 have support for dynamically loadable device drivers which are easier to debug.

Another problem traditionaly encountered was the lack of adequate documentation: the one available from the OS vendor was often sketchy on details and short on examples, if any were provided. This has also changed: there are now good references on writing device drivers [Egan and Teixeria, 1992], [Pajari, 1992]. Furthermore, there are now several versions of Unix available with full source code at prices accessible to others than large corporations, and in some cases at no cost. For instance, the 386BSD / NetBSD / FreeBSD systems are based on 4.3 BSD, which is documented in detail in [Leffler *et al.*, 1989]. Another example is Linux, which is also free and available with full source. Thus the driver developper is free to study the source code of all of the other drivers available for the system, as well as to obtain assistance from the many other developpers who use USENET as a forum for exchanging information. In this case, since the primary environment is

SunOS/Solaris, the documentation provided by Sun has to suffice.

Note that in many cases, it is not necessary to write an actual device driver, since most of the work can be done in an application. For instance, an application used here at McRCIM interfaces to a VME-based frame grabber by mapping its memory into the address space of the process using the *mmap()* system call and accessing that memory as a normal C array. Unfortunately, in this case this option is not possible since the convolver will be generating interrupts and the only part of the system which is able to respond to interrupts is the interrupt service routine which is part of a device driver.

The normal semantics for a Unix device driver is to support the *read()*, *write()* and *ioctl()* system calls. In this case, it would be impractical to use *read()* and *write()* although it would be possible for an application to *"write()"* the source image to **/dev/convolver** and *"read()"* back the result from the same device, this would result in unnecessary copying of data which would make the system very inefficient Instead, the convolver operations outlined in section 3.11 are simply mapped into corresponding *ioctl()* operations. Note that the **Set Image Source Address** and **Set Image Destination Address** operations will take care of locking the appropriate image ranges into physical memory. As for the **Perform Convolution** operation, it is provided both in blocking and non-blocking versions.

Chapter 4 Comparison with General Purpose Systems

4.1 Introduction

In their study of computer graphics hardware, Myer and Sutherland identified more than 25 years ago what they called the "Wheel of Reincarnation" [Myer and Sutherland, 1968]. Simply put, this means that as a problem is identified which requires more computing power than available using general-purpose systems, the temptation to design and implement special-purpose hardware to solve this problem grows. In most cases, the system will perform as expected and provide a viable solution. But this has been done at the expense of flexibility: the specialized system can do one thing only (albeit very well), and it is generally harder to use (and program) than a general purpose machine. As the speeds of the latter increase and begin to overtake the speed of the specialized hardware, it will become tempting to migrate the application back to the general-purpose machine until the entire cycle can be repeated again. This cyclical migration is no less true today, although we seem to be in the "moving to a general-purpose architecture" phase. UNIX workstations have been doubling in performance every 18 months for the last few years and their prices have been falling steadily. In many cases, the cost of the hardware is being dwarfed by the cost of software development, so it makes sense to move to an environment which enhances software productivity.

This project has by no means been immune to this phenomenon. When it first started, most workstations offered floating-point performance in the range of a few hundred kiloFLOPS, which meant that our system had almost three orders of magnitude more performance. Nowadays, many workstations can deliver a

few dozen MFLOPS without even having to resort to hand-coded assembly language (optimizing compiler technology has greatly improved, and on many RISC architectures there is little additional performance to be gained by programming in assembler instead of in a high-level language such a FORTRAN or C [Bell, 1990]) This chapter will present implementations of the floating-point convolution algorithm on a number of general-purpose machines. This will allow us to compare the performance which can be obtained from a C program (with a bit of care, but no extraordinary feats of hand-optimization). We will also be able to compare a few different architectures:

- an SIMD machine, the MasPar MP-1
- RISC processors arranged in MIMD fashion, the Silicon Graphics 4D/240
- single-processor RISC workstations such as the SPARC-based Sun SS10/30, the IBM RS/6000 model 360 and the R4000-based Silicon Graphics Indigo R4000

4.2 An SIMD machine, the MasPar MP-1

4.2.1 System Hardware

The MasPar system is a Single Instruction Multiple Data computer which is oriented towards high-speed scientific computing involving array operations Typical applications are low-level image processing, computational fluid mechanics and finite element analysis. It consists of two main parts: a Front End (FE) workstation which handles all interactions with the user and the Data Parallel Unit (DPU) which contains the actual SIMD machine. A block diagram of the architecture of the system is presented in figure 4.1. A thorough treatment of the MasPar system hardware and I/O subsystems can be found respectively in [MasPOp, 1990] and

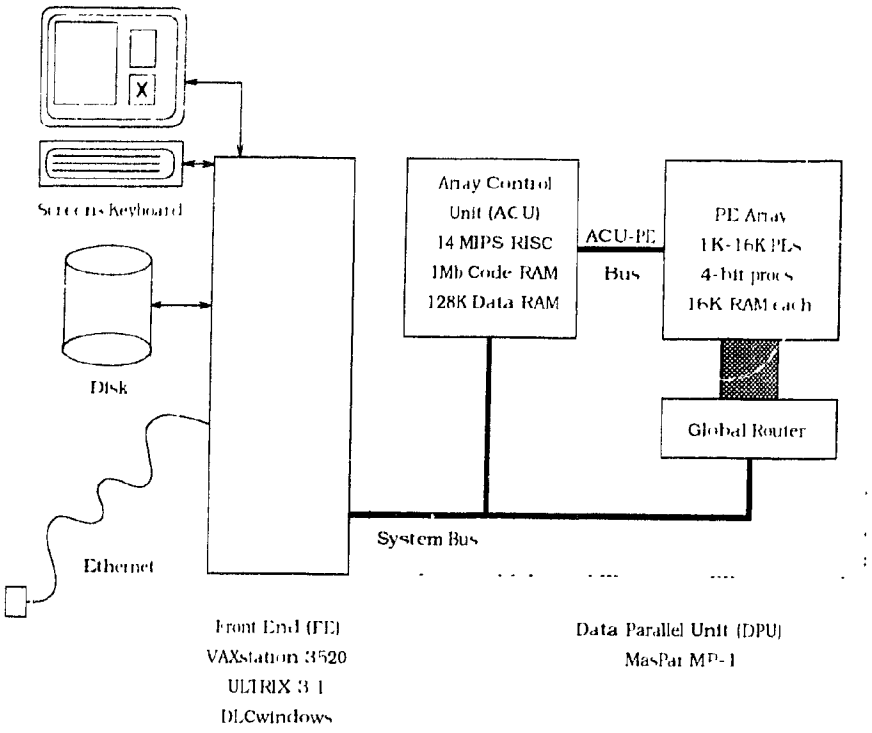

Figure 4.1: MasPar MP-1 System Block Diagram

[MasIO, 1990] Note that the architecture of the MP-1 is quite similar to that of the CM-200 from Thinking Machines Corporation [Ramanathan and Oren, 1993].

Front End Host

The MasPar is controlled by a Front End (FE) host, a VAXstation 3250 from Digital Equipment Corporation. All user interaction with the MP-1 is done through the Front End, which also acts as a disk and communications server. The primary task of the Front End is to handle all sequential code in a application, in particular all user interface functions. Parallel operations are performed on the Data Parallel Unit (DPU). This communication is handled by a high-speed interface: a number of registers and First-In First-Out queues are mapped into the address space of the Front End and let it transfer data to and from the DPU. Among other things, this interface allows the Front End direct access to the internal bus of the DPU (a

variation on the standard VMEbus), thus allowing Direct Memory Access (DMA) transfers.

The Data Parallel Unit

The Data Parallel Unit is composed of two main components: the Array Control Unit (ACU) and the Processing Element (PE) array. The Array Control unit is composed of a 14 MIPS RISC processor with thirty-two 32-bit registers and a Harvard-style architecture. It has 1Mb of memory for code and 128K of memory for data, which is sufficient since most data will reside in PE memory (the ACU can page out to the Front End disk if necessary). The primary purpose of the Array Control Unit is to act as a sequencer for the Processing Elements of the PE array. As such, it communicates with the PE array over the ACU-PE bus: it uses this bus to broadcast instructions to be executed by all the currently active PEs as well as broadcast data values from its own address space to the PEs. The ACU-PE bus can also be used to read values back from the PEs to the ACU: the outputs of all the PEs are then connected together in wired-OR fashion. Thus if more than one PE responds to a read request from the ACU, the ACU will receive the bitwise-OR of the data values sent by all the responding PEs. The other function of the ACU is to execute all non-parallel code running in the DPU: this includes of course all control statements which dictate which PEs will participate in which instructions, but also all operations on variables which are stored in the data memory of the ACU.

The PE array can contain from 1K to 16K processing elements. Each PE is a load/store arithmetic processor with dedicated register space and RAM. Each PE has a 1.6 MIPS control processor, forty 32-bit registers (32 of which are available to the programmer and 8 which are reserved for the system micro-code) and 16K of data memory (recall that no instruction memory is required since instructions are broadcast by the ACU). The PEs are physically implemented as a full custom CMOS VLSI device containing 32 such PEs. Since the PEs on a chip share the data

RAM area (16K of which is reserved for each PE), access to this RAM is much slower than access to the private registers. Thus register allocation must be carefully done to ensure maximum efficiency in parallel programs

Each PE has both a sequential ID (assigned starting from 0, with no gaps), as well as x and y ID numbers which identify its position in the array. Thus the PE array can be viewed either as a one-dimensional array or a two- dimensional array depending on the needs of the parallel algorithm. A number of status bits on each PE specify whether the PE will participate in the current instruction being broadcast by the ACU. The set of PEs which are currently enabled to execute the next instruction is known as the *active set*. Conditional statements broadcast by the ACU can modify the active set by disabling or enabling PEs. These tests can either be performed on data that is local to the PEs or on the PE index variables (in the case where we want to exclude a geometric portion of the array).

Each PE is connected with its eight nearest neighbors by the "XNET", and the 2D PE array wraps toroidally at the edges A status bit in each PE determines its participation in XNET transfers controlled by the ACU, which can cause each enabled PE to transfer the value stored at a given address to a neighbor in a given direction: all such transfers occur at once, thus achieving large I/O bandwidth (a total of over 2.2 Gb/sec). A mode called "Pipelined XNET" allows XNET transfers at a distance of more than 1 PE in a more efficient manner than several 1 PE distance transfers.

A Global Router allows any PE to communicate with any other PE in the array. For the purposes of Global Router communication, the PEs are grouped into clusters of 16 PEs which share a bi-directional serial line to the serial router. Access to these serial lines is arbitrated in microcode: since the bandwidth of the Global Router is much lower than that of the XNET (around 50 Mb/sec), care must be taken to avoid contention as much as possible by distributing data cleverly in the PE array.

4.2.2 System Software

The programming model of the MasPar is that of two tightly coupled programs running together, one of the Front End and one on the Data Parallel Unit. The program running on the Front End is a purely sequential program, typically written in a traditional language such as FORTRAN or C using the standard UNIX compilers for these languages. The parallel part of the program executes on the DPU, and is written in a parallel language. When a call is made across this boundary, data values have to be copied over the FE to DPU bus, since the FE and the ACU have different address spaces. Single values can be passed via FIFO queues, blocks of data can be transferred using DMA directly to and from the memory of the PEs. These function calls can be either synchronous or asynchronous.

Athought the DPU can be programmed in assembly language, most users will use instead the MasPar Parallel application Language (MPL). MPL is basically "old style" Kernighan and Ritchie C enriched with a new data type modifier, *plural* modifier. In MPL, any variable which is declared normally resides in the data address space of the ACU. In the context of this parallel environment, these variables are known as *singular* variables. Plural variables, on the other hand, are allocated at the same memory location on every PE (this is dictated by the SIMD nature of the PE array). Whenever an operation involves only singular variables, the ACU performs this operation on its own. As soon as an operation involves a plural variable, the result of the operation is a plural result and all of the PEs which are part of the active set take part in this operation. For instance:

```
int i;
plural int j,k;
k = i+j;    /* This is a plural operation */
```

In this case, the ACU will broadcast the content of its i variable to all of the PEs. The active PEs will then add this value to the content of the j variable and store the

result in their *k* variable.

C control also accept plural arguments: in this case, they influence the size of the current active set. For instance, the following code avoids divisions by zero:

```
plural double i,j,k;
if(i!=0.0)
{
    k = j / i;  /* Avoid division by zero */
}
```

This piece of code will cause every active PE to test the value of its *i* variable: those that find it equal to zero are temporarily excluded from the active set for the duration of the if compound statement. Thus the active set can only be reduced by MPL control structures. When such a structure exits, the active set is restored to its previous state. Note that in the following code:

```
plural double i,j,k;
if(i!=0.0)
{
    k = j / i;  /* Avoid division by zero */
}
else
{
   i = -1.0;    /* Make sure i is no longer zero */
}
```

some PEs which have *i* originally non-zero will execute the *if* path of the statement, whereas others which have *i* set to 0 will execute the *then* part of the statement. The original semantics of the if-then-else construct are respected by every PE individually, but not when we consider the complete array. Greater detail about MPL can be found in [MPLref, 1990] and [MPLguide, 1990].

Inter-PE communication is implemented using the *xnet* and *router* pseudo variables. For instance, in the following code

```
plural int i;
i = xnetW[1].i;
```

we can say the every PE "retrieves" the value of the i variable from its westerly neighbor, 1 PE away (i.e. its direct neighbor to the west) and copies it into its own i variable. Remember that XNET communications occur all at once: the net effect of this action is thus to shift the values of the i variables by one position to the east, keeping in mind the toroidal wrapping property of the PE array. Similarly, a PE can use the Global Router to get a variable from any other PE

```
plural int i,j,k;
i = router[j].k;
```

Here, every active PE would retrieve the value of the k variable stored on the PE whose number is in its j variable and store it in its i variable. There are 6 pseudo-variables which help PEs make decisions as to whether to participate in an operation:

- *nproc* the total number of PEs in the array
- *nxproc* the width of the PE array
- *nyproc* the height of the PE array
- *iproc* the index of the PE (viewing the PE array as linear)
- *ixproc* the column index of the PE in the array
- *iyproc* the row index of the PE in the array

4.2.3 Implementation and Results

Our implementation of the convolution algorithm on the MasPar follows the following steps: first, the image is read off the disk by the Front End and remapped

into a format compatible with the layout of the Processing Elements in the PE array. It is then transfered to PE memory, together with the values of the kernel coefficients. The convolution is then performed by the DPU under the control of the ACU. Once the operation is finished, the results are read back into the FE. Since we are only really interested in the floating-point performance of the MasPar, we will only benchmark the amount of time required to perform the convolution by the DPU and exclude the overhead of transfering the image. Since the MasPar machine which was available had only 64 by 32 PEs, most of the work was taken up by coming up with schemes for mapping the 512x256 test image into the PE array. Three such schemes are considered here.

Implementation Method 1

This first method has been proposed in [Jacobsen, 1990]. The principle is to split the image into blocks which are of the size of the PE array. These blocks are then copied into the PE memories in an array of pixels: for instance, on PE (0,0), this array contains the top-left-most pixel of every block in the image. Every PE first multiplies its pixel value with the first kernel coefficient, then transmits the partial result to its east neighbor. These partial results are accumulated and transmitted to the east until a row of coefficients has been used up: the partial results are then transmitted to the south. After a number of iterations equal to the number of coefficients in the convolution kernel, the partial result will contain the resulting convolved pixel (although not for the pixel on the PE on which it resides: it will have to be moved back to the PE containing the original pixel). This process is illustrated in figure 4.2 for a 2 by 2 kernel operating on a 3 by 3 image.

XNET communication is a natural candidate to communicate the partial results between adjacent PEs since in this method, PEs only need to communicate with their immediate neighbors. Since all the PEs communicate at once and they are all enabled, very high bandwidth is attained. Note that the operation of this algorithm

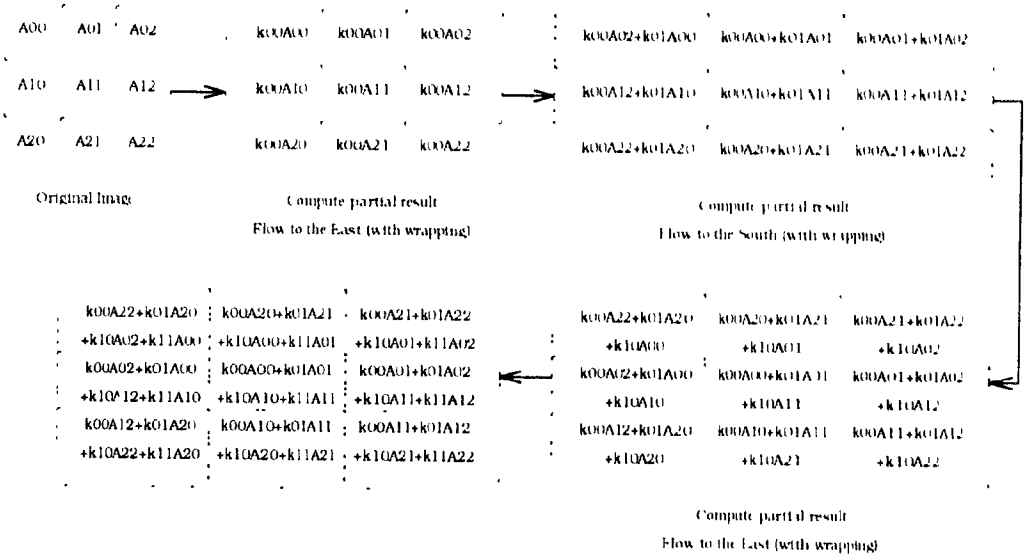

Figure 4.2: Method 1 - 2x2 kernel, 3x3 image

is similar to the systolic array method used to implement our hardware convolution processor.

This partial convolution is performed on every block in the original image: if the original image had the same size as the PE array, then this algorithm could rely only on the toroidal wrap property of the PE array to handle boundary conditions. Unfortunately, this is not the case here, since the image is larger. Although it has been done for the other two methods, the code required to handle these boundary conditions has not been implemented in this case.

Implementation Method 2

In the second method, the image partitioning is the same as for the first method. Each PE computes the convolution result for the pixel stored in its memory by implementing the convolution equation directly:

$$Pixel[i][j] = \sum_{x=-\frac{u}{2}}^{\frac{u}{2}} \sum_{y=-\frac{h}{2}}^{\frac{h}{2}} Pixel[i+x][j+y]k[x][y] \qquad (4.1)$$

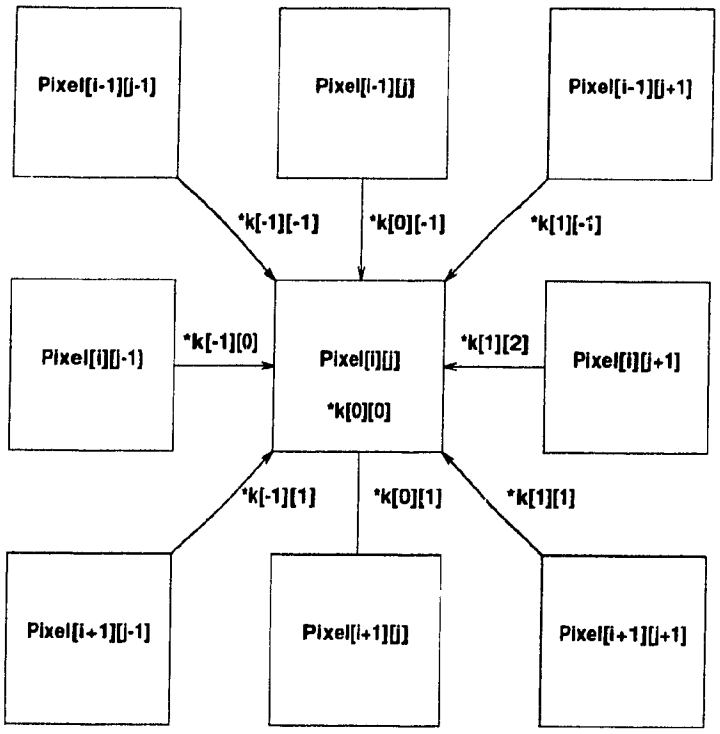

Figure 4.3: Method 2 - 3x3 kernel example

The kernel coefficients, stored on the ACU, are broadcast to all the PEs when the multiplications are done. The neighboring pixels are read from the neighboring PEs using XNET communication. Figure 4.3 demonstrates the operation of method 2 for a 3 by 3 convolution kernel.

Implementation Method 3

In this third implementation, the image is broken up into as many contiguous, rectangular regions as there are PEs. Each of these image blocks is stored on a PE, and each PE operates on its own region of the image, implementing equation 4.1 directly. With this method, the communication between PEs is minimized, since apart from the pixels on the edge of the image block, the PEs will be able to compute the convolution without requiring any data from their neighbors. This method is especially attractive when the image is much larger than the size of the PE array,

since in this case there will be very little communication required between the PEs (the proportion of "edge" pixels to "interior" pixels being small). On the other hand, if the region stored on each PE is very small (especially if it is not much larger than the kernel size), then this method does not have many advantages since every pixel will be an "edge" pixel and communication with neighboring PEs will be required.

Implementation Performance

Figure 4.4 represents the performance of the three methods when taking into account only the time required to execute the convolution algorithm on the DPU. First note that the results for a 3 by 3 kernel are not very significant: since the *clock*() function call used to time this function has a resolution of around 10 msec and the execution time was around 80 to 90 msec, large errors can have crept in (although all of the figures plotted in the graphs represent averages over 10 runs of the program). A top performance of around 37 MFLOPS is obtained in all three cases for a 9 by 9 kernel. Note that as the kernel gets larger, the performance increases: this is due to the fact that as more floating point computations need to be performed, the overhead due to address computations and loop index checking becomes comparatively smaller, thus yielding a higher perceived floating point throughput. When comparing this somewhat dissappointing result with the performance of machines in the following sections, one must keep in mind that these results where obtained early in 1991: since then, the MasPar machine has been upgraded several times with new microcode and hardware which have reportedly increased its performance (the author was not able to successfullyrun these tests again after the upgrades).

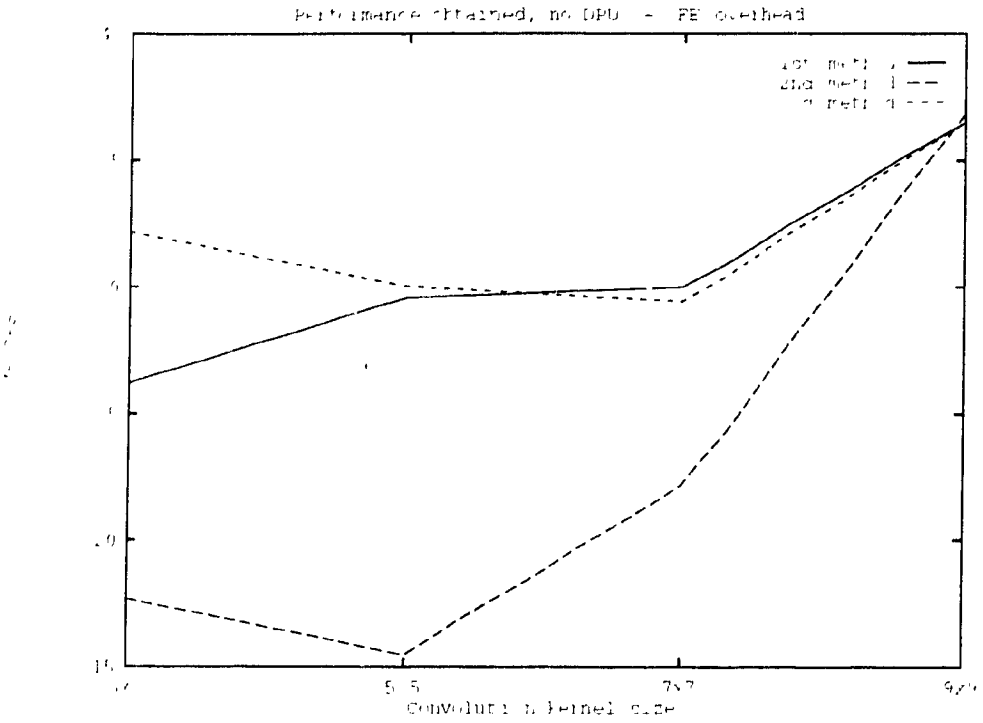

Figure 4.4: MasPar Implementation Performance

4.3 An MIMD Machine, the Silicon Graphics 4D/240

4.3.1 System Hardware

For the last few years, Silicon Graphics has offered a line of "symmetric multi-processing" systems called the POWER Series based on MIPS (now owned by SGI) R3000 micro-processors. These processors are configured in Multiple Instruction, Multiple Data (MIMD) fashion. They all share a common system memory which is accessed over an inter-processor communication bus (to which is also connected the graphics subsystem). In order to decrease traffic over this bus and reduce contention between the processors, each CPU has a private local cache memory for both instruction and data. Cache coherency hardware ensures that no cache ever holds a stale copy of data which has been updated by another processor: this hardware approach to cache coherency means that "traditional", single-threaded

applications usually need not be aware that they are running on a multi-CPU system. On the other hand, this additional hardware adds both complexity and cost to the machine As in most MIMD machines, the main bottleneck is the interprocessor bus which can quickly become saturated, since it offers only 64 MB/sec of throughput. Although POWER Series machines can be configured with up to 8 CPUs, in many applications little performance is gained by going above 4 processors (especially when heavy use is made of the graphics subsystem, which can require a significant portion of the bus bandwidth to be fed with enough data to run at full speed). Recently, Silicon Graphics has announced the new Onyx and Challenge lines of multi-processing machines. These retain the same basic architecture, but the bandwidth of the multiprocessor bus has been raised to 1.2 GB/sec, allowing up to 36 MIPS R4400 processors running at 75MHz externally (150 MHz internally) to share the bus [SGISMP, 1993]. "True" performance of these systems is not known at this point

The 4D/240 system on which the convolution algorithm was coded is not as recent. It is based on 4 25MHz MIPS R3000 CPUs, each having 2x64Kb of primary cache memory for instructions and data and 256Kb of secondary cache. There are 128Mb of shared system memory. Although this is by no means the fastest of the POWER Series system, the performance obtained can be scaled quite closely with clock frequency (the fastest Power Series machines have R3000 processors running at 40MHz).

4.3.2 System Software

SGI machines run a version of the UNIX operating system called IRIX It includes several extensions, notably in the areas of graphics, real-time capabilities and "symmetric multiprocessing". Basically, most parallel processing on these systems is very coarse grained and occurs at the UNIX process level One of the processors on the system runs the IRIX kernel, which is responsible for dispatching processes

to the available CPUs. In this way, programs can be completely unaware that there are several CPUs in the system: if there are 4 runnable processes at one time, they each get the benefit of a "full" CPU (assuming that their memory access patterns don't conflict too much with each other, and in particular that they don't "bust" the local CPU cache too often) All of the CPUs are usually kept fairly busy in a multi-user environment: four processes could run at full speed with little interference between each other In visual simulation applications (such as a low-cost flight simulator), one of the CPUs could be used to traverse the visual database and determine the visible polygons, the second one could be used to feed the polygons to the graphics pipe, the third one could be used to run the actual simulation (i.e. compute the flight equations) and the fourth one could be used to interface to external peripherals (such as the cockpit controls).

It is also possible for a single process to take advantage of more than one CPU. The *m_fork()* system call creates a copy of the process which calls it on each available CPU, and calls the same function of that process on each CPU. Each instance of the function has its own stack and local variables, but contrary to the standard UNIX call *fork()*, all the copies of the process share the same addressing space (i.e. global variables and dynamically allocated memory). When all of the instances of the function have completed, the *m_fork()* call returns and the process continues running on a single CPU.

4.3.3 Implementation and Results

The implementation of the convolution algorithm on the SGI 4D/240 works roughly along these lines: the main process takes care of of reading the image from disk and converting it to floating-point format. It then stores it in a 2D array in C-style row-major format In order to avoid border effects, the image is extended by replicating a band of width equal to half the kernel size around its perimeter. Although this method requires a few extra floating-point computations, it will save a lot of time

by greatly simplifying the addressing computations. An early implementation of this algorithm which used the modulo operation to obtain the proper wrap-around behavior got disastrous performance on all architectures on which it was compiled.

The main process then uses *m_fork*() to start four instances of a function which computes a straight-forward 2D convolution sum. Each of these functions operates on a horizontal quarter-image to minimize contention for main memory access. The only time when the CPUs try to access the same regions of memory are for the convolution coefficients (which will remain in CPU-local cache memory after the first time they are read) and for a thin region along the boundary between the quarter images. It would have been possible to duplicate these regions in order to eliminate this contention, but a closer study of memory access patterns would have been required to justify the effort. Unfortunately, SGI does not provide any tools for monitoring these patterns.

Test results were generated for 1, 2 and 4 CPUs Also, the inner loop of the convolution sum was explictly unrolled, since the SGI compiler was not smart enough to do it on its own. Loop unrolling is a technique whereby small loops with fixed boundaries are replaced with as many instances of the loop body as there would have been iterations in the loop. All array indices are replaced by constants correponding to the iteration index. In heavily pipelined processors, unrolling prevents pipeline stalls due to branch instructions and simplifies the work of the optimizer which can find the optimal scheduling for the instruction stream.

Table 4.1 lists the number of MFLOPS obtained for varying kernel sizes and number of CPUs used. First note that due to the lack of an accurate timing function and to the fact that it was not possible to bring down the system to single-user mode to run these tests, some of the values can be off by significant amounts. Clearly, the performance of the convolution algorithm seems to scale linearly with the number of CPUs installed in the system, which means that the algorithm exhibits strong locality of reference, allowing the individual CPUs to run at full speed without

3 × 3	5 × 5	7 × 7	9 × 9	kernel size
1.69	2.58	2.67	3.48	1 CPU (no unrolling)
3.22	4.96	5.19	6.96	2 CPUs (no unrolling)
6.41	9.77	10.35	14.02	4 CPUs (no unrolling)
6.54	6 77	6.71	6.66	1 CPU (unrolling)
12.82	13.48	13.48	13.27	2 CPUs (unrolling)
24.32	26.53	26.54	26.44	4 CPUs (unrolling)

Table 4.1: MFLOPS Results for the SGI 4D/240

interfering with each other when accessing main memory. Also, in the case where the convolution sum loop was not unrolled, performance increases as the kernel size increases, which reflects the fact that loop overhead becomes less of an issue as the size of the loop increases. When the loop is unrolled manually, the performance remains mostly constant across kernel size (it even seems to decrease for the 9×9 kernel size, which might suggest that we are starting to have problems with the cache at that point). Finally, these results also show that in most cases, compilers still need to be given a hint (in this case explicit loop unrolling) to allow them to generate code which makes full use of the capabilities of the machine.

4.4 Single Processor RISC Machines

4.4.1 Motivation

Single-processor UNIX RISC workstations are still the most widely used systems for scientific computations. A properly-written C or FORTRAN program can be recompiled without modification on most such machines. Instead of trying to take advantage of specialized hardware through hardware-specific code, one can either run a program on many machines at once (thus supporting the claim by Sun Microsystems that "The Network is the Computer"), or one can wait for the performance of workstations to increase to a point where one's application runs

in a reasonable amount of time. At the rate at which workstation performance is increasing these days, one might not have to wait that long! Another advantage in favor of single-processor machines is that these architectures are well understood by compiler writters who are able to write fairly efficient compilers. as a general rule, the more specialized the architecture is, the more difficult the software development tools are to use. For instance, if one does not like the C compiler provided with Sun workstations, there are many other alternatives available (including the excellent and free GNU C Compiler).

4.4.2 Implementation and Results

The convolution algorithm implemented as a C program was tested on three widely-used UNIX workstations:

- the Sun SparcStation 10/30, based on a SPARC processor
- the Silicon Graphics Indigo, based on a MIPS R4000 processor
- the IBM RS/6000 Model 360, based on a POWER processor

The vendor-supplied compiler was used in all three cases with maximum optimization enabled. The program follows basically the same lines as the implementation on the SGI POWER Series: the image is read from disk and stored in memory as a 2D array. The image border is explictly replicated to avoid costly wrap-around address calculations. The following loop performs the actual convolution:

```
kx_middle = k_width / 2;
ky_middle = k_height / 2;
for(i=ky_middle;i<height;i++)
  {
    for(j=kx_middle;j<width;j++)
      {
        result = 0.0;
        for(k=0;k<k_height;k++)
          for(l=0;l<k_width;l++)
            result+=srcimage[i+k-ky_middle][j+l-kx_middle]*
                    kernel[k][l];
        destimage[i][j]=result;
      }
  }
```

Once again, none of the compilers were able to unroll the two inner loops, even though the loop boundaries *k_width* and *k_height* were explictly declared as constants. Thus a second version of the code was also compiled and run were the two inner loops were explictly unrolled.

Table 4.2 lists the MFLOPS performance obtained both without and with loop unrolling. When the inner loops are not unrolled, performance increases with the kernel size since this minimizes loop overhead. When the loops have been explictly unrolled, the performance remains fairly constant with kernel size, except in the case of the SPARCStation 10 where performance decreases markedly: this is probably due to the memory access patterns conflicting with the mapping of memory to the cache (which is fairly large at 1Mb). The IBM RS/6000 Model 360 has clearly superior performance: this is in part due to the fact that its CPU implements a single-cycle multiply-and-accumulate instruction which is being used by the code generated by the C compiler. Note that although the RS/6000 CPU implements IEEE floating-point semantics, no result renormalization is done between the multiplication and addition operations in the multiply-and-add unit. This could lead to results which are different than those obtained on a machine which lacks such a functional unit and where the results would be renormalized after both the multiply and the add operations. In order to duplicate these results, the IBM XL C

3×3	5×5	7×7	9×9	kernel size
3.21	4.41	5.24	5.83	SUN SS10/30 (no unrolling)
5.22	7.96	9.15	10.83	SGI Indigo (no unrolling)
19.03	27.08	30.95	33.76	IBM RS/6000 360 (no unrolling)
12.31	10.08	9.54	9.42	SUN SS10/30 (unrolling)
14.04	15.79	17.96	17.67	SGI Indigo (unrolling)
40.68	45.83	45.88	45.47	IBM RS/6000 360 (unrolling)

Table 4.2: Single Processor MFLOPS Results

compiler offers a switch to disable the generation of multiply-and-add instructions, at a significant penalty in performance.

4.5 Possible Implementation on a Vector Processor

Since convolution is basically a two-dimensional multiply and accumulate operation, it could be implemented on any pipelined vector processor which supports this operation. Since this dot product operation is a mainstay of many scientific computations, most vector processors implement it in hardware. The kernel coefficients would be stored in linear fashion in one of the processor vector registers, and the appropriate pixels would be stored in another vector register. This second vector is basically the neighboring pixels of the pixel presently under consideration

A single vector instruction would then compute the resulting pixel. Unfortunately, this has two problems. If the kernel size is small, the length of the vectors will be short (9 elements for a 3x3 kernel), and this may not be enough to justify the overhead of a vector instruction, although most vector machines nowadays have vector instructions which are faster than their scalar counterparts for anything but the shortest vectors. The other problem is that we must still deal with the fact that each pixel in the source image will have to be read several times from memory (as many times as there are kernel coefficients). Thus the overhead of forming the "neighborhood" vectors to feed to the vector unit might also reduce performance

significantly.

4.6 Discussion of Results and Recommendationds for Future Work

With 81 VLSI devices forming a 9 by 9 array, the convolution array discussed in this thesis performs 162 double-precision floating-point operations per clock cycle: at a design speed of 16 MHz, this corresponds to a sustained rate of 162 MFLOPS. The actual rate obtained on a physical computer depends on how fast operands can be transfered between the array and the memory of the host over the VMEbus. To operate at its peak design rate, the array must receive a new input operand every microsecond and produces results at the same rate. If both the input and the output data streams are in floating-point format, this translates to a bus transfer bandwidth of 16 Mb/s. In a typical system with other boards requiring a portion of the VMEbus bandwidth, this maximum convolver rate might not be supported continuously. On the other hand, when dealing with 8-bit fixed precision operands, only 2 Mb/s of bus bandwidth is required, which is well within the capabilities of the VME bus. At this point, the DMA engine of the convolution processor has been built and partially tested. OrCAD design tools were used to draw the schematics, produce netlists, generate the PAL fuse maps and simulate critical parts of the design (most importantly the local bus control logic). A high-quality prototyping wire-wrap board was used to build the system: this board has a VMEbus 6U form factor, Pin Grid Array (PGA) areas for the 68020 and the VIC and built-in decoupling capacitors for all the power connections. A utility program has been written to combine the netlist generated by the OrCAD schematic entry tool and the layout of the components on the prototype board to generate a detailed wiring list: the output of this program was very useful in minimizing wire-wrapping errors.

In order to achieve maximum performance (especially in the case of the tight DMA-control loop which must fit in the 128 byte instruction cache), the 68020 has been programmed in assembly language. This control program as well as the

self-test routines were assembled on a UNIX host using a 68020 cross assembler. The output of the assembler (a binary file in Motorola S-Record format) is used to program the 32K EPROM By observing the state of the local bus with a logic analyzer, the 68020 has been observed to initialize itself and the VIC interface ASIC and execute simple test code sequences such as accesses to the ROM and RAM. The principal work which remains to be done is to complete the integration of the board with both the convolution array itself and the VMEbus host. The fabrication and performance tests on the custom VLSI convolver chip constitutes a separate research project which is being carried out concurrently. Because of this, fully functional convolver units were not available for integrated testing with the DMA system.

After investing significant efforts in the design, construction and testing of our convolution processor, it is somewhat disappointing to see that we were able to obtain almost 30% of its performance (46 versus 162 MFLOPS) with a C program running on a general-purpose UNIX workstation such as the IBM RS/6000 model 360 (see section 4.4.2). Nevertheless, several points which serve to justify our design have to be kept in mind:

- Whereas IBM has access to the latest technology when implementing its workstations, this was not the case for our convolution processor. In particular, the 3 micron CMOS process used to implement the systolic array chip is completely obsolete. A major recommendation to obtain a much higher performance would involve re-implementation using a higher density CMOS process.

- It is much easier to integrate a VMEbus board in a real-time image-processing system. Workstations usually lack high-speed I/O connections and are thus difficult to connect to external hardware which requires high bandwidth, unless one is ready to deal with the often proprietary expansion bus, and thus go back to designing dedicated hardware. Several VMEbus boards can be

connected together (possibly using higher-speed point-to-point connections for image data) to achieve results which are not possible otherwise. Of course, given enough performance in a UNIX workstation, the entire system could be implemented in software at a much lower cost and with less difficulty, but most image-processing algorithms are still starved for computational power and will remain so for the next few years.

- The RS/6000 Model 360 is by no means a low-cost solution: fully configured, its price runs up to well over $50,000. If it were produced in even modest quantities, the proposed convolution processor system would cost much less than that, especially if the systolic array device were re-implemented using a lower-cost methodology such as a gate array instead of a full-custom device (the price differential coming mostly from the non-recurring expenses).

As indicated above, an obvious strategy for increasing the performance of the system would be to re-implement the VLSI systolic array devices using a higher density CMOS process This would have the effect of both raising the operational frequency and reducing the silicon area required by the device: thus either a smaller die could be used or several devices could be combined on a single chip (thus reducing the physical size of the array). Another possible approach would be to re-implement the systolic cell using a gate array or standard cell methodology, which although not as fast or dense as a full-custom device has the advantage that it allows easier access to the higher-performance processes available from the manufacturers (the full-custom process offered by the Canadian Microelectronics Center is not the latest process available from commercial manufacturers).

Of course, any increase in the computational capacity of the array would mean an additional burden on the VMEbus DMA interface. In order to alleviate this problem, the on-board recombination memory could be made more general and mapped into the on-board local bus address space, as well as being made accessible to the host CPU when the convolution processor responds to slave VMEbus cycles.

Thus the host CPU could manipulate an image stored directly on the convolution processor to perform operations which cannot be done by the convolution array. The DMA interface would then be used only to transfer the input data from its source (possibly a frame grabber) and to transfer the final result to its destination (a frame buffer for instance).

Another possible avenue of work would be to reuse the DMA engine to drive other types of dedicated processors which operate on a high-speed data stream. For instance, the growing interest in "video on demand" systems is fueling the need for high-performance video image compression and decompression engines which can operate on digital video streams. Since the DMA engine is already capable of dealing with different bandwidths at the input and output of the convolution processor (which is required when operating with different input and output data types), it could be adapted for compression/decompression applications. Digital video systems are likely to be the next area to demand specialized designs which can perform more computations and handle larger amounts of data than a CPU is capable of.

Conclusion

This thesis presents the design of a double-precision floating point convolution processor which can be used as an attached processor in a VMEbus-based system. Typical applications could be low-level computer vision and image processing tasks in a real-time environment. It also reviews the relevant literature and shows how this design relates to other systolic solutions proposed or implemented in the past. It discusses the trade-offs between general-purpose and specialized architectures, between which system designers seem to be continually oscillating. Performance results are presented and evaluated, and suggestions for future work are also made.

General-purpose architectures seem to be currently favored by many, and this can be seen in the number of manufacturers of high-end "supercomputers" who seem to be abandoning dedicated designs in favor of large numbers of general-purpose RISC processors connected in parallel. Certainly the capabilities of hardware have been growing at a much faster rate than those of software, and it thus makes sense to design a system which can reuse software developped for "traditional" architectures. In many projects, the cost of software easily outweighs that of hardware. On the other hand, there are applications where general-purpose hardware simply cannot be used: for instance, image generation systems for commercial and military flight simulators (the author's current area of work). Although some workstation manufacturers would have us believe otherwise, there is currently no viable substitute to dedicated image generators unless a severe degradation of the training value of the simulator is accepted. There will always be applications where performance is the premier criterion, and for those applications, dedicated hardware systems will continue to be designed and built.

References

[Abraham *et al.*, 1987] J. A. Abraham, P. Banerjee, C.-Y. Chen, W. K. Fuchs, S.-Y Kuo, and A. L. N. Reddy, "Fault tolerance techniques for systolic arrays," *IEEE Computer*, vol 20, pp 65–75, July 1987.

[Anfinson, 1988] C J. Anfinson, "A linear algebraic model of algorithm-based fault tolerance," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 483–493, May 1988.

[Annaratone *et al.*, 1987] M. Annaratone, E. Arnould, T. Gross, H. Kung, M. Lam, O. Menzilcioglu, and J. Webb, "The Warp Computer: Architecture, implementation and performance," *IEEE Transactions on Computers*, vol. 36, pp. 1523–1538, December 1987.

[Antola *et al.*, 1988] A. Antola, R. Negrini, M. G. Sami, and N. Scarabottolo, "Policies for fault-tolerance throgh mixed space- and time- redundancy in semi-systolic FFT arrays," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 565–576, May 1988.

[Bandyopadhyay *et al.*, 1988] S. Bandyopadhyay, G. A. Jullien, and A. Sengupta, "A systolic array for fault tolerant digital signal processing using a residue number system approach," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 577–586, May 1988.

[Banerjee, 1988] U. Banerjee, *Dependence Analysis for Supercomputing* Norwell MA: Kluwer Academic Publishers, 1988

[Bell, 1990] R. Bell, *IBM RISC System/6000 Performance Tuning for Numerically Intensive FORTRAN and C Programs.* International Business Machines, Armonk, NY, 1990.

[Benaini and Robert, 1990] A. Benaini and Y. Robert, "Spacetime-minimal systolic architectures for gaussian elimination and the algebraic path problem," in *Proceedings of the International Conference on Applications Specific Array Processors*, (Princeton, NJ), pp. 746–757, September 1990.

[Bertolazzi *et al.*, 1988] P. Bertolazzi, C. Guerra, and S. Salza, "A systematic approach to the design of modular systolic arrays," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp 453–462, May 1988

[Boudreault and Malowany, 1986] Y. Boudreault and A. S. Malowany, "A VLSI convolver for a robot vision system," in *Proceedings of the Canadian Conference on Very Large Scale Integration*, 1986.

[Bourbakis and Barlos, 1988] N. Bourbakis and F. Barlos, "Performance evaluation of the Hermes multibit systolic array architecture for low level processing tasks," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 113–124, May 1988.

[Bu *et al.*, 1990a] J. Bu, E. F. Deprettere, and P. Dewilde, "A design methodology for fixed-size systolic arrays," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 591–602, September 1990.

[Bu *et al.*, 1990b] J. Bu, E. F. Deprettere, and L. Thiele, "Systolic array implementation of nested loop programs," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 31–42, September 1990.

[Bursky, 1992] D. Bursky, "FPGA advances cut delays, add flexibility," *Electronic Design*, vol. 40, pp. 35–43, October 1992.

[Cappello, 1992] P. Cappello, "A processor-time-minimal systolic array for cubical mesh algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 4–13, January 1992.

[Chakrabarti and JáJá, 1990] C. Chakrabarti and J. JáJá, "Systolic architectures for the computation of the discrete Hartley and the discrete cosine transforms based on prime factor decomposition," *IEEE Transactions on Computers*, vol. 39, pp. 1359–1368, November 1990.

[Chean and Fortes, 1990] M. Chean and J. A. B. Fortes, "A taxonomy of reconfiguration techniques for fault-tolerant processor arrays," *IEEE Computer*, vol. 23, pp. 55–69, January 1990.

[Chen and Yao, 1988] M.-J. Chen and K. Yao, "Linear systolic array for least-squares estimation," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 83–92, May 1988.

[Cheng, 1988] K. H. Cheng, "Efficient designs of priority queues," in *Proceedings of the 1988 International Conference on Parallel Processing - Vol. I*, (Penn State University, Pa), pp. 363–366, August 1988.

[Chester *et al.*, 1991] D. B. Chester, W. R. Young, and M. Petrowski, "A fully systolic adaptive filter implementation," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing ICASSP*, (Toronto, On), pp. 2109–2212, 1991.

[Chiang and Fu, 1990] C. C. Chiang and H. C. Fu, "An improved multilayer neural model and array processor implementation," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 389–400, September 1990.

[Chinn *et al.*, 1990] G. Chinn, K. A. Grajski, C. Chen, C. Kuszmaul, and S. Tomboulian, "Systolic array implementations of neural nets on the MasPar MP-1 massively parallel processor," in *Proceedings of the International Joint Conference on Neural Networks*, (San Diego, CA), June 1990.

[Choi and Boriakoff, 1992] J. Choi and V. Boriakoff, "A new linear systolic array for FFT computation," *IEEE Transactions on circuits and systems - II: Analog and Digital Signal Processing*, vol. 39, pp. 236–239, April 1992.

[Choudhary and Patel, 1988] A. N. Choudhary and J. H. Patel, "A parallel processing architecture for an integrated vision system," in *Proceedings of the 1988 International Conference on Parallel Processing - Vol. I*, (Penn State University, Pa), pp. 383–387, August 1988.

[Chung *et al.*, 1992] J.-H. Chung, H. Yoon, and S. R. Maeng, "A systolic array exploiting the inherent parallelisms of artificial neural networks," *The EUROMICRO Journal, Microprocessing and Microprogramming*, vol. 33, pp. 145–159, May 1992.

[Clark, 1992] T. R. Clark, "Tom Clark on: FPGA design," *Computer Design*, vol. 31, pp. 31–32, December 1992.

[Clauss *et al.*, 1990] P. Clauss, C. Mongenet, and G. R. Perrin, "Calculus of space-optimal mappings of systolic algorithms on processor arrays," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 4–18, September 1990.

[CMC, 1989] Canadian Microelectronics Corporation, Carruthers Hall, Queen's University, Kingston, Canada, *Guide to the Integrated Circuit Implementation Services of the Canadian Microelectronics Corporation*, GICIS version 4:0 ed., March 1989.

[Codenotti and Tamassia, 1991] B. Codenotti and R. Tamassia, "A network flow approach to reconfiguration in VLSI arrays," *IEEE Transactions on Computers*, vol. 40, pp. 118–121, January 1991.

[Côté, 1990] J.-F. Côté, "The design of a testable floating point convolution processor," Master's thesis, McGill University, Montréal, Qc, November 1990.

[Delosme, 1990] J.-M. Delosme, "Bit-level systolic algorithm for the symmetric eigenvalue problem," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 770–781, September 1990.

[Dennis, 1980] J. B. Dennis, "Data flow supercomputers," *IEEE Computer*, vol. 13, pp. 48–56, November 1980.

[Drolet *et al.*, 1990] J. Drolet, J.-F. Panisset, J.-F. Côté, F. Larochelle, and A. Malowany, "A double precision floating point convolution system," in *Proceedings of*

the ASME International Computers in Engineering Conference, vol. 2, (Boston, MA), pp. 1–6, American Society of Mechanical Engineers, August 1990.

[Drolet *et al.*, 1991] J. Drolet, J.-F. Panisset, and A. Malowany, "Design of a floating point convolution processor," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, (Quebec City, Canada), pp. 13.5.1–13.5.4, Canadian Society for Electrical and Computer Engineering, September 1991.

[Drolet, 1992] J. Drolet, "The design of a floating-point convolution system," Master's thesis, McGill University, Montréal, Qc, November 1992.

[Dudgeon and Mersereau, 1984] D. E. Dudgeon and R. M. Mersereau, *Multidimensional Digital Signal Processing*. Englewood Cliffs NJ: Prentice Hall, 1984.

[Egan and Teixeria, 1992] J. Egan and T. Teixeria, *Writing A Unix Device Driver*. New-York NY: John Wiley & Sons, 2nd ed., 1992.

[Egan, 1991] B. T. Egan, "Designers search for the secret to ease ASIC migration," *Computer Design*, vol. 30, pp. 78–94, December 1991.

[Ersoy, 1985] O. Ersoy, "Semisystolic array implementation of circular, skew circular and linear convolutions," *IEEE Transactions on Computers*, vol. C-34, pp. 190–196, February 1985.

[Fisher and Kung, 1985] A. L. Fisher and H. T. Kung, "Synchronizing large VLSI processor arrays," *IEEE Transactions on Computers*, vol. C-34, pp. 734–740, August 1985.

[Fortes and Wah, 1987] J. A. Fortes and B. W. Wah, "Systolic arrays - from concept to implementation," *IEEE Computer*, vol. 20, pp. 12–17, July 1987.

[Frison and Quinton, 1984] P. Frison and P. Quinton, "An integrated systolic machine for speech recognition," in *VLSI: Algorithms & Architecture. Proceedings of the International Workshop on Parallel Computing & VLSI.*, (Amalfi, Italy), pp. 175–186, May 1984.

[Futurebus+, 1990] Institute of Electrical and Electronics Engineers, *IEEE Std 896 - 1990, IEEE Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus+*, 1990.

[Gharachorloo *et al.*, 1988] N. Gharachorloo, S. Gupta, E. Hokenek, P. Balasubramanian, B. Bogholtz, C. Mathieu, and C. Zoulas, "Subnanosecond pixel rendering with million transistor chips," in *Computer Graphics (SIGGRAPH)*, (Atlanta, GA), pp. 41–49, August 1988.

[Gokhale *et al.*, 1990] M. B. Gokhale, A. Kopser, S. P. Lucas, and R. G. Minnich, "The logic description generator," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 111–120, September 1990.

[Gokhale *et al.*, 1991] M. Gokhale, W. Holmes, A. Kopser, S Lucas, R Minnich, D. Sweely, and D. Lopresti, "Building and using a highly parallel programmable logic array," *IEEE Computer*, vol. 24, pp. 81–89, January 1991.

[Gross *et al.*, 1985] T. Gross, H. T. Kung, M. Lam, and J Webb, "Warp as a machine for low-level vision," in *Proceedings of the 1985 IEEE International Conference on Robotics & Automation*, (St-Louis, Missouri), pp. 790–800, March 1985

[Guerra and Kanade, 1984] C. Guerra and T. Kanade, "A systolic algorithm for stereo matching," in *VLSI: Algorithms & Architecture. Proceedings of the International Workshop on Parallel Computing & VLSI*, (Amalfi, Italy), pp 103–112, May 1984.

[Haule, 1990] D. D. Haule, "Design of a VLSI system for image processing," Master's thesis, McGill University. Montréal, Qc, March 1990.

[Hellwagner, 1988] H. Hellwagner, "A systolic array with constant I/O bandwidth for the generalized Fourier transform," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 207–216, May 1988.

[Hoang, 1992] D. T. Hoang, "A systolic array for the sequence alignment problem," Tech. Rep. CS-92-22, Department of Computer Science, Brown University, Providence, RI, April 1992.

[Hou *et al.*, 1988] P.-P. Hou, R. M. Owens, and M J. Irwin, "A high level synthesis tool for systolic designs," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 665–673, May 1988.

[Hu *et al.*, 1990] Y Hu, J. V. McCanny, and M. Yan, "Systolic VLSI compiler (SVC) for high performance vector quantization chips," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 145–155, September 1990.

[Huang, 1972] T. Huang, "Two-dimensionals windows," *IEEE Transactions on Audio and Electroacoustics*, pp. 88–89, March 1972.

[Hwang and Briggs, 1984] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York NY: McGraw-Hill Book Company, 1984.

[Ibarra and Sohn, 1989] O. H. Ibarra and S. M. Sohn, "On mapping systolic algorithms onto the hypercube," in *Proceedings of the 1989 International Conference on Parallel Processing - Vol. I: Architecture*, (Penn State University, Pa), pp. 121–124, August 1989.

[IEEE-754, 1985] Institute of Electrical and Electronics Engineers, *IEEE Std 754 - 1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[Jacobsen, 1990] K. Jacobsen, *Image Convolutions, Application Note*. MasPar Corporation, Sunnyvale, CA, 1990.

[Johnsson *et al.*, 1988] L Johnsson, C.-T. Ho, M. Jacquemin, and A. Ruttenberg, "Systolic FFT algorithms on boolean cube networks," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp 151–162, May 1988.

[Jones, 1993] K Jones, "Parallel DFT computation on bit-serial systolic processor arrays," *IEE Proceedings-E. Computers & Digital Techniques*, vol. 140, pp. 10–18, January 1993.

[Kar and Bapeswara Rao, 1993] D. C. Kar and V. Bapeswara Rao, "A new systolic realization for the discrete fourier transform," *IEEE Transactions on Signal Processing*, vol. 41, pp 2008–2010, May 1993.

[Khoros, 1991] The Khoros Group, Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM, *Khoros Manual Release 1.0*, 1991.

[Kim, 1988] J. H Kim, "On the design of easily testable and reconfigurable systolic arrays," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 505–514, May 1988.

[Ko and Wing, 1988] C. K. Ko and O. Wing, "Mapping strategy for automatic design of systolic arrays," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 285–294, May 1988

[Kothari *et al.*, 1989] S.C. Kothari, H. Oh, and E. Gannet, "Optimal designs of linear flow systolic architectures," in *Proceedings of the 1989 International Conference on Parallel Processing - Vol. I: Architecture*, (Penn State University, Pa), pp 247–256, August 1989.

[Kugelmas, 1988] S. D. Kugelmas, "A probabilistic model for clock skew," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 545–554, May 1988.

[Kumar and Trai, 1988] V. K. P. Kumar and Y.-C. Trai, "Mapping two dimensional systolic arrays to one dimensional arrays and applications," in *Proceedings of the 1988 International Conference on Parallel Processing - Vol. I*, (Penn State University, Pa), pp. 39–46, August 1988.

[Kung and Leiserson, 1979] H. Kung and C. Leiserson, "Systolic Arrays (for VLSI)," in *Sparse Matrix Proceedings 1978*, *Society for Industrial and Applied Mathematics*, pp. 256–282, 1979.

[Kung *et al.*, 1987] S. Y. Kung, S. C. Lo, S. N. Jean, and J. N. Hwang, "Wavefront array processors - concept to implementation," *IEEE Computer*, vol. 20, pp. 18–33, July 1987.

[Kung, 1982] H. Kung, "Why systolic architectures?," *IEEE Computer*, vol. 15, pp. 37–46, January 1982.

[Kung, 1988] S. Y. Kung, "Parallel architectures for artificial neural nets," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 163–174, May 1988.

[Kwan and Okullo-Oballa, 1990] H.-K. Kwan and T. S Okullo-Oballa, "2-D systolic arrays for realization of 2-D convolutions," *IEEE Transactions on Circuits and Systems*, vol. 37, pp. 267–273, February 1990.

[Kwan, 1993] H.-K. Kwan, "Systolic realisation of delayed two-path linear phase FIR digital filters," *IEE Proceedings-G: Circuits, Devices & Systems*, vol. 140, pp. 75–80, February 1993.

[Lam, 1991] S. P. S Lam, "A systolic implementation of the Jacobi algorithm," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing ICASSP*, (Toronto, On), pp. 1021–1024, 1991.

[Larochelle *et al.*, 1989] F. Larochelle, J.-F. Côté, and A. Malowany, "A floating point convolution systolic cell," in *Proceedings of the Vision Interface '89 Conference*, (London, Ont.), pp. 77–80, June 1989.

[Larochelle, 1991] F. Larochelle, "VLSI design of a double precision floating point convolution systolic cell," Master's thesis, McGill University, Montreal, Qc, March 1991.

[Lee and Kedem, 1989] P. Lee and Z. M Kedem, "Mapping nested loop algorithms into multi-dimensional systolic arrays," in *Proceedings of the 1989 International Conference on Parallel Processing - Vol III Algorithms & Applications*, (Penn State University, Pa), pp. 206–210, August 1989

[Leffler *et al.*, 1989] S. Leffler, M. K McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System* Reading MA: Addison-Wesley Publishing Company, 1989.

[Lengauer *et al.*, 1991] C Lengauer, M Barnett, and D. G. Hudson, "Towards systolizing compilation," *Distributed Computing*, vol. 5, pp. 7–24, June 1991

[Lim, 1990] J. S. Lim, *Two-Dimensional Signal and Image Processing*. Englewood Cliffs NJ: Prentice Hall, 1990.

[Lin, 1988] W.-T. Lin, "Mapping systolic algorithms into shuffle arrays," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 351–360, May 1988.

[Liu *et al.*, 1990] K. R. Liu, S.-F Hsieh, and K Yao, "Two-level pipelined implementation for systolic block Householder transformation with application to RLS algorithm," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 758–769, September 1990

[Liu *et al.*, 1991] K. R. Liu, S.-F. Hsieh, K. Yao, and C.-T. Chiu, "Dynamic range, stability, and fault-tolerant capability of finite-precision RLS systolic array based on Givens rotations," *IEEE Transcations on Circuits and Systems*, vol. 38, pp. 625–636, June 1991.

[Lopresti, 1987] D. P. Lopresti, "P-NAC: a systolic array for comparing nucleic acid sequences," *IEEE Computer*, vol. 20, pp. 98–99, July 1987.

[Malowany and Malowany, 1989] M. Malowany and A. Malowany, "Color-edge detection algorithm for a high-performance convolution processor," in *Proceedings of the Vision Interface Conference*, (London Ont.), June 1989.

[MasIO, 1990] MasPar Corporation, Sunnyvale, CA, *MasPar System I/O Manual*, 1990.

[MasPOp, 1990] MasPar Corporation, Sunnyvale, CA, *MasPar MP-1 Principles of Operation*, 1990.

[MC68020, 1989] Motorola Inc., Englewood Cliffs, NJ, *MC68020 32-Bit Microprocessor User's Manual, Third Edition*, 1989.

[McRCIM, 1990] McRCIM, "Mcgill research center for intelligent machines annual report," tech. rep., McGill University, Montréal, Qc, 1990.

[McWhirter *et al.*, 1990] J. G. McWhirter, D. S. Broomhead, and T. J. Shepherd, "A systolic array for nonlinear adaptive filtering and pattern recognition," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 700–711, September 1990.

[Mead and Conway, 1980] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading MA: Addison-Wesley Publishing Company, 1980.

[Megson, 1991] G. Megson, "Systolic algorithm for B-spline patch generation," *Journal of Paralle and Distributed Computing*, vol. 11, pp. 231–238, March 1991.

[Meher *et al.*, 1993] P. Meher, J. Satapathy, and G. Panda, "Efficient systolic solution for a new prime factor discrete Hartley transform algorithm," *IEE Proceedings-G: Circuits, Devices & Systems*, vol. 140, pp. 135–139, April 1993.

[Mercklenbrauker and Merserau, 1976] W. F. Mercklenbrauker and R. M. Merserau, "McClellan transformations for two-dimensional digital filtering: II - implementation," *IEEE Transactions on Circuits and Systems*, vol. CAS-23, pp. 414–422, July 1976.

[Merserau *et al.*, 1976] R. M. Merserau, W. F. Mercklenbrauker, and J. Thomas F Quatieri, "McClellan transformations for two-dimensional digital filtering: I - design," *IEEE Transactions on Circuits and Systems*, vol. CAS-23, pp. 405–414, July 1976

[MIPSASM, 1987] Silicon Graphics Computer Systems, Mountain View, CA, *Assembly Language Programmer's Guide*, 1987.

[Mokhoff, 1993] N. Mokhoff, "Technology trends: PLDs/FPGAs," *Electronic Engineering Times*, pp. T1–T48, March 22 1993.

[Moldovan and Fortes, 1986] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Transactions on Computers*, vol. C-35, pp. 1–12, January 1986.

[Moonen and Vandewalle, 1993] M. Moonen and J. Vandewalle, "A systolic array for recursive least squares computations," *IEEE Transactions on Signal Processing*, vol. 41, pp. 906–912, February 1993.

[Moreno and Lang, 1988] J. H. Moreno and T. Lang, "Graph-based partitioning of matrix algorithms for systolic arrays. application to transitive closure," in *Proceedings of the 1988 International Conference on Parallel Processing - Vol I*, (Penn State University, Pa), pp. 28–31, August 1988.

[Moreno and Lang, 1990] J. H. Moreno and T. Lang, "Matrix computations on systolic-type meshes - an introduction to the multimesh graph method," *IEEE Computer*, vol. 23, pp. 32–51, April 1990.

[Motorola, 1987] Motorola Inc., *Using the MC68020 as a Dedicated DMA Controller*, 1987. Application Note DC003.

[MPLguide, 1990] MasPar Corporation, Sunnyvale, CA, *MasPar Parallel Application Language (MPL) User Guide*, 1990.

[MPLref, 1990] MasPar Corporation, Sunnyvale, CA, *MasPar Parallel Application Language (MPL) Reference Manual*, 1990.

[Multibus, 1983] Intel Corporation, *Multibus Data Book*, 1983.

[Myer and Sutherland, 1968] T. Myer and I. Sutherland, "On the design of display processors," *Communications of the ACM*, vol. 11, pp. 410–414, June 1968.

[Omtzigt, 1988] E. T. L. Omtzigt, "SYSTARS: a CAD tool for the synthesis and analysis of VLSI systolic/wavefront arrays," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 383–391, May 1988.

[Omtzigt, 1990] E. T. L. Omtzigt, "Domain flow and streaming architectures," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 438–447, September 1990.

[Oppenheim and Schafer, 1989] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*. Englewood Cliffs NJ: Prentice Hall, 1989.

[Pajari, 1992] G. Pajari, *Writing Unix Device Drivers*. Reading MA: Addison-Wesley Publishing Company, 1992.

[Panchanathan and Goldberg, 1991] S. Panchanathan and M. Goldberg, "A systolic array architecture for image coding using adaptive vector quantization," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 1, pp. 229–229, June 1991.

[Panisset *et al.*, 1990] J.-F. Panisset, J. Drolet, J.-F. Côté, F. Larochelle, and A. S. Malowany, "A floating point convolution system," in *Proceedings of the 33rd Midwest Symposium on Circuits and Systems*, (Calgary, Alta.), August 1990.

[Panneerselvam *et al.*, 1988] G. Panneerselvam, G. Jullien, and W. Miller, "New architectures for systolic hashing," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 73–82, May 1988.

[Parks and McClellan, 1972] T. W. Parks and J. H. McClellan, "Chebyshev approximation for nonrecursive digital filters with linear phase," *IEEE Transactions on Circuit Theory*, vol. CT-19, pp. 189–194, March 1972.

[Payer, 1988] M. Payer, "Formal derivation of systolic arrays – a case study," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp 331–340, May 1988.

[Peng and Jun, 1988] S.-T. Peng and M. S. Jun, "A new VLSI 2-D systolic array for matrix multiplications and its applications," in *Proceedings of the 1988 International Conference on Parallel Processing - Vol. III - Algorithms & Applications*, (Penn State University, Pa), pp. 169–172, August 1988.

[Perry, 1991] D. L. Perry, *VHDL*. New-York NY: McGraw-Hill, 1991.

[Phillips and Robertson, 1988] W Phillips and W. Robertson, "A systolic architecture for the symmetric tridiagonal eigenvalue problem," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 145–150, May 1988.

[Poli and Bayoumi, 1988] S. P. Poli and M. A. Bayoumi, "A reconfigurable VLSI array for reliability and yield enhancement," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 631–642, May 1988.

[Psarakis and Moustakides, 1991] E. Z. Psarakis and G. V. Moustakides, "Design of two-dimensional zero-phase FIR filters via the generalized McClellan transform," *IEEE Transactions on Circuits and Systems*, vol. 38, pp. 1355–1363, November 1991.

[Ramacher and Raab, 1990] U. Ramacher and W. Raab, "Fine-grain system architecture for systolic emulation of neural algorithms," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 554–566, September 1990.

[Ramanathan and Oren, 1993] G. Ramanathan and J. Oren, "Survey of commercial parallel machines," *Computer Architecture News*, vol. 21, pp. 13–33, June 1993.

[Rivest *et al.*, 1978] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public key cryptosystems," *Communications of the ACM*, vol. 21, February 1978.

[Rose *et al.*, 1993] J. Rose, A. El Gammal, and A. Sangiovanni-Vincentelli, "Architecture of field-programmable gate arrays," *Proceedings of the IEEE*, vol. 81, pp. 1013–1029, July 1993.

[Sarkar and Majumdar, 1991] S. Sarkar and A. Majumdar, "An instruction systolic array implementation of the two-dimensional fast fourier transform," *EUROMICRO Journal*, vol. 33, pp. 101–110, November 1991.

[Sarkar *et al.*, 1991] S. Sarkar, A. Majumdar, and R. Sen, "A hardware efficient systolic solution to the two-dimensional discrete fourier transform," *EUROMICRO Journal*, vol. 33, pp. 111–117, November 1991.

[Scheiman and Cappello, 1992] C. J. Scheiman and P. R. Cappello, "A processor-time-minimal systolic array for transitive closure," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 257–269, March 1992.

[Sciuto and Lombardi, 1988] D. Sciuto and F Lombardi, "New conditions for testability in two-dimensional bilateral arrays," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 495–504, May 1988

[SGISMP, 1993] Silicon Graphics Computer Systems, Mountain View, CA, *Symmetric Multiprocessing Systems Technical Report*, 1993.

[Sha and Steiglitz, 1991] E. H.-M. Sha and K. Steiglitz, "Reconfigurability and reliability of systolic/wavefront arrays," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing ICASSP*, (Toronto, On), pp. 1001–1004, 1991.

[Snyder, 1982] L. Snyder, "Introduction to the configurable, highly parallel computer," *IEEE Computer*, vol. 15, pp. 47–56, January 1982.

[Speake and Mersereau, 1981] T. C. Speake and R. M. Mersereau, "A note on the use of windows for two-dimensional FIR filter design," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-29, pp. 127–127, February 1981.

[Squier and Steiglitz, 1990] R. Squier and K. Steiglitz, "A practical runtime test method for parallel lattice-gas automata," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 783–793, September 1990.

[Steenarrt and Zhang, 1991] W. Steenarrt and J. Y. Zhang, "Mapping recursive algorithms onto systolic architectures," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing ICASSP*, (Toronto, On), pp. 1209–1212, 1991.

[Sternheim *et al.*, 1990] E. Sternheim, R. Singh, and Y. Trivedi, *Digital Design with Verilog HDL*. Cupertino CA: Automata Publishing Company, 1990.

[Stone, 1987] H. S Stone, *High-Performance Computer Architecture*. Reading MA: Addison-Wesley Publishing Company, 1987.

[Sun, 1989a] Sun Microsystems Inc., Mountain View, CA, *User's Guide to the Sun-3/100 VMEbus*, 1989.

[Sun, 1989b] Sun Microsystems Inc., Mountain View, CA, *Writing Device Drivers for the Sun Workstation*, 1989.

[Tang *et al.*, 1991] C. Tang, K Liu, and S. Tretter, "On systolic arrays for recursive complex Householder transformations with applications to array processing," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing ICASSP*, (Toronto, On), pp. 1033–1036, 1991.

[Thomborson and Wei, 1991] C. D. Thomborson and B. W.-Y. Wei, "Systolic implementation of a move-to-front text compressor," *Computer Architecture News*, vol 19, pp. 53–60, March 1991.

[Torralba and Navarro, 1988] N. Torralba and J. J. Navarro, "A one dimensional systolic array for solving arbitrarily large least mean square problems," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 103–112, May 1988.

[Travis, 1991] D. Travis, *Effective Color Displays - Theory and Practice*. London UK: Academic Press, 1991.

[Tseng, 1988] P. Tseng, "Iterative sparse linear system solvers on Warp," in *Proceedings of the 1988 International Conference on Parallel Processing - Vol. III - Algorithms & Applications*, (Penn State University, Pa), pp. 32–38, August 1988.

[Tseng, 1990] P. Tseng, "A systolic array programming language," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 794–803, September 1990.

[Valero-García *et al.*, 1990] M. Valero-García, J. J. Navarro, J. M. LLabería, and M. Valero, "Implementation of systolic algorithms using pipelined functional units," in *Proceedings of the International Conference on Application Specific Array Processors*, (Princeton, NJ), pp. 272–283, September 1990.

[Van Dongen and Quinton, 1988] V. Van Dongen and P. Quinton, "Uniformization of linear recurrence equations: A step towards the automatic synthesis of systolic arrays," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 473–482, May 1988.

[VHDL, 1987] Institute of Electrical and Electronics Engineers, *IEEE Std 1076 - 1987, IEEE Standard VHDL Language Reference Manual*, March 1987.

[VMEbus, 1982] Motorola Semiconductor Products Inc., *VMEbus Specification Manual*, August 1982.

[Wallace, 1991] G. K. Wallace, "The JPEG still picture compression standard," *Communications of the ACM*, vol. 34, pp. 30–44, April 1991.

[Wan and Evans, 1993] C. Wan and D. Evans, "A systolic array architecture for linear and inverse matrix system," *Parallel Computing*, vol. 19, pp. 303–321, March 1993.

[Wolfe, 1989] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*. London England: Pitman Publishing, 1989.

[Wong and Delosme, 1988] Y. Wong and J.-M. Delosme, "Broadcast removal in systolic algorithms," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 403–412, May 1988.

[Wong and Delosme, 1992] Y. Wong and J.-M. Delosme, "Transformation of broadcasts into propagations in systolic algorithms," *Journal of Parallel and Distributed Computing*, vol. 14, pp. 121–145, January 1992.

[Yaacoby and Cappello, 1988] Y. Yaacoby and P. R. Cappello, "Scheduling a system of affine recurrence equations onto a systolic array," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp 373–382, May 1988.

[Youn and Singh, 1988] H. Y. Youn and A. D Singh, "A highly efficient design for reconfiguring the processor array in VLSI," in *Proceedings of the 1988 International Conference on Parallel Processing - Vol. I*, (Penn State University, Pa), pp 375–382, August 1988.

[Zhang *et al.*, 1988] C. N. Zhang, H. L. Martin, and D. Y. Y Yun, "Parallel algorithms and systolic array designs for RSA cryptosystem," in *Proceedings of the International Conference on Systolic Arrays*, (San Diego, CA), pp. 341–350, May 1988.

[Zhong and Rajopadhye, 1991] X. Zhong and S. Rajopadhye, "Synthesizing fully efficient systolic arrays," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing ICASSP*, (Toronto, On), pp. 1241–1244, 1991.