MatRISC: A RISC Multiprocessor for Matrix Applications

Andrew James Beaumont-Smith
M.Eng.Sc, B.E.(Hons.)

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

THE UNIVERSITY OF ADELAIDE

Faculty of Engineering, Computer and Mathematical Sciences
Department of Electrical and Electronic Engineering

November, 2001

Abstract

Computer workstations and real-time systems that implement signal processing, control and numerical algorithms often have a large reuse and interaction of operands due to the occurrence of matrix operators. This thesis proposes a highly integrated SOC (system on a chip) matrix-based parallel processor which can be used as a co-processor when integrated into the on-chip cache memory of a microprocessor in a workstation environment. This generalised processor architecture called MatRISC *(Matrix RISC) is a mesh-connected array of processing elements and is optimised for matrix multiplication. It is able to achieve high execution speed and processor efficiency by routing input operands from a non-blocking, high bandwidth data memory to multiple functional units through a programmable synchronous mesh network with minimal control overhead and with all processing elements operating in lock step. A static scheduling regime is used, since algorithms which can be decomposed into matrix- or vector-based kernel routines can be readily parallelised and incur no fine-grained control flow dependence on data for the large majority of operations performed. A small 16-bit RISC processor in each processing element performs housekeeping tasks for the very long instruction word (VLIW) sequencing engine and provides flexibility for multiple-instruction multiple-data (MIMD) operation which is needed when data-dependencies arise or the algorithm has a non-cyclo-static control structure. The RISC and VLIW control instructions for each processing element are generated by a compiler from an intermediate level language called Mcode, which describes the operation of the whole processor array. A matrix multiplication routine is developed which is rate-, delay- and processor-optimal when executed on the MatRISC processor. Matrix multiplication achieves a higher compute rate than vector operations using the same memory bandwidth. Significant speedup for other matrix-based kernel operations is also achieved. Algorithms which have been mapped onto the processor include the Volterra model, Fourier transform, Kalman filter and numerical decompositions. A prototype processor,* MatRISC-1 *is presented which is a CMOS implementation of a* 4×4 *array of processing elements. New architectures and CMOS implementations are presented for IEEE-754 standard compliant floating-point adders and accumulators, parallel prefix integer adders, a dynamic bus and crossbar switch for routing data and a pipelined SRAM. The MatRISC-1 processor can achieve a* $25GFLOPS$ *peak compute rate and over 90% processor efficiency with a modest clock rate of* $200MHz$ *in a* $0.35\mu m$ *CMOS technology.*

Contents

List of Figures

List of Tables

List of Abbreviations

AG	Address Generator
BLAS	Block Linear Algebra Subroutines
CBS	Crossbar Switch
CISC	Complex Instruction Set Computer
COW	Cluster of Workstations
CPI	Cycles per Instruction
CPU	Central Processing Unit
CRO	Cathode Ray Oscilloscope
CSU	Clock Synchronisation Unit
CVSL	Cascode Voltage Source Logic
DDL	Digital Delay Line
DFT	Discrete Fourier Transform
DLL	Delay Locked Loop
DM	Data Memory
DP	Data Processor
DRAM	Dynamic Random Access Memory
DSM	Distributed Shared Memory
EAC	End-around Carry
FIFO	First-in First-out
FLOPS	Floating-point Operations per Second
FP	Floating-point
FPU	Floating-point Unit
FSFG	Fully Specified Flow Graph
FFT	Fast Fourier Transform
GME	Generalised Mapping Engine
HPTC	High Performance Technical Computing
ILP	Instruction-level Parallelism

IM	Instruction Memory
IP	Instruction Processor
IPB	Iteration Period Bound
IPC	Instructions per Clock
ISA	Instruction Set Architecture
LAPACK	Linear Algebra Package
LSB	Least Significant Bit
MAC	Multiply-accumulate
MatRISC	Matrix Reduced Instruction Set Computer
MCM	Multi-chip Module
MCode	MatRISC Code
MDMA	Multi-dimensional Memory Array
MFLOPS	Mega-FLOPS
MIMD	Multiple-instruction Multiple-data
MOSFET	Metal-oxide Silicon Field Effect Transistor
MPI	Message Passing Interface
MPP	Massively Parallel Processor
MSB	Most Significant Bit
MUX	Multiplexer
NaN	Not-a-Number
PB	Period Bound
PE	Processing Element
PSSIMD	Pseudo Synchronous SIMD
PVP	Parallel Vector Processors
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTL	Register Transfer Language
RTS	Real-time System
SIMD	Single-instruction Multiple-data
SMP	Symmetric Multiprocessors
SRAM	Static Random Access Memory
S-SSIMD	Skewed-synchronous SIMD
SVD	Singular Valued Decomposition
VCDL	Voltage Controlled Delay Line
VCO	Voltage Controlled Oscillator
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word

Declaration

This thesis has been submitted to the Faculty of Engineering, Computer and Mathematical Sciences at the University of Adelaide for examination in respect of the Degree of Doctor of Philosophy.

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University or other tertiary institution and, to the best of my knowledge and belief contains no material previously published or written by another person, except where due reference is made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Andrew Beaumont-Smith

27 November 2001.

Acknowledgments

This work was done with the support of many people to whom I am indebted. Firstly I want to thank my supervisor, Cheng-Chew Lim who supported me and the MatRISC processor work and has always been willing to allow me to follow my own ideas, find extra funding for things like chip fabrication and test equipment when it was needed. Thanks also to Mike Liebelt, co-investigator on the project for always being there to bounce new ideas off, give much needed advice with computer architecture and and some excellent go-kart racing which I usually won. My friend and colleague, Warren Marwood developed the concept of an array processor in his PhD work and his continued input into the project has been very valuable. Many thanks to Neil Burgess who enlightened me to the imprecise science of computer arithmetic, his many tricks and ideas were an inspiration to help me come up with a few of my own. Neil also introduced me to the computer arithmetic conference family, who are some of the most interesting people I've met. Thanks also for bringing the IEEE Computer Conference to Adelaide and allowing me to help with the organisation.

Special thanks to the Adelaide VLSI lab crew, colleagues who worked very hard on this project and spent many late nights in the VLSI lab finishing chips and papers including Kiet To, Stephane Lefreré (our French recruit), Chris Howland, John Tsimbinos and Nick Betts. Thanks go to my other friends and colleagues from Adelaide including Sam Appleton, Ali Moini, Shannon Morton, Andrew Blanksby, Richie Beare and Said Alsarawi for many discussions, ideas and fun. I want also to acknowledge the support of my colleagues in the Alpha Development Group of Digital, then Compaq and now the Intel Massachusetts Microprocessor Design Center. Gratitude is also expressed to the Australian Research Council who provided funding to the University for the project through an ARC large grant. Thanks to Mum and Richard for prodding me when I needed to be, encouraging me to pursue postgraduate work and proof reading.

This thesis survived a house renovation, a pregnancy, a birth, some motor racing, a new job and it all could not have been done without Natalie (and later, Chloe and Emma).

ABS.

Publications

The following is a list of publications published or submitted during the Ph.D. candidature by the author.

Andrew J. Beaumont-Smith and Cheng-Chew Lim,
"Parallel Prefix Adder Design",
Proc. 15^{th} *IEEE Symposium on Computer Arithmetic*,
Vail, Colorado, USA, 11–13 June 2001, pp. 218–225.

Andrew J. Beaumont-Smith, John E. Tsimbinos, Cheng-Chew Lim and Warren Marwood,
"A VLSI Chip Implementation of an A/D Converter Error Table Compensator",
Computer Standards and Interfaces, 23, 2001, pp. 111-122.

Kiet N. To, Cheng-Chew Lim, Andrew J. Beaumont-Smith, Michael J. Liebelt and Warren Marwood,
"An Array Processor Architecture for Support Vector Learning",
Proc. 3^{rd} *International Conference on Knowledge-based Intelligent Information Engineering Systems*, Adelaide, Australia, 31^{st} August - 1^{st} September 1999, pp. 377-380.

Andrew J. Beaumont-Smith, Cheng-Chew Lim, John E. Tsimbinos and Warren Marwood,
"A VLSI chip Implementation of an A/D Converter Error Table Compensator",
Proc. IEE 3^{rd} *International Conference on Advanced A/D and D/A Conversion Techniques and their Applications (ADDA'99)*,
University of Strathclyde, Glasgow, UK, 26–28 July 1999, pp. 122-125.

Andrew J. Beaumont-Smith, John E. Tsimbinos, Cheng-Chew Lim, Warren Marwood and Neil Burgess,
"A 10GOPS Transversal Filter and Error Table Compensator",
Proc. SPIE '99, Denver, Colorado, USA, July 1999, pp.157-163.

Andrew J. Beaumont-Smith, Neil Burgess, Stephane Lefreré and Cheng-Chew Lim,
"Reduced Latency IEEE Floating-Point Standard Adder Architectures",
Proc. 14th IEEE Symposium on Computer Arithmetic,
Adelaide, Australia, April 1999, pp. 35–42.

Andrew J. Beaumont-Smith, Warren Marwood, Kiet N. To, Cheng-Chew Lim and Michael J. Liebelt,
"MatRISC-1: A RISC Multiprocessor for Matrix-Based Real Time Applications",
Proc. 5th Australasian Conference on Parallel and Real-Time Systems (PART'98),
Adelaide, September 1998, pp. 320–331.

Andrew J. Beaumont-Smith and Neil Burgess,
"A GaAs 32-bit Adder",
Proc. 13th IEEE Symposium on Computer Arithmetic,
Asilomar, California, USA, July 1997, pp. 10–17.

Andrew J. Beaumont-Smith, Michael J. Liebelt, Cheng-Chew Lim, Kiet N. To and Warren Marwood,
"A Digital Signal Multiprocessor for Matrix Applications",
Proc. 14th Australian Microelectronics Conference,
Melbourne, Australia, September 1997, pp. 245–250.

Andrew J. Beaumont-Smith, Neil Burgess, Song Cui and Michael J. Liebelt,
"GaAs Multiplier and Adder Designs for High-speed DSP Applications",
Proc. 31st Asilomar Conference on Signals, Systems and Computers,
Asilomar, California, USA, November 1997, pp. 1517–1521.

Andrew J. Beaumont-Smith, John E. Tsimbinos, Warren Marwood, Cheng-Chew Lim and Michael J. Liebelt,
"A Matrix Processor Implementation of the Volterra Model",
Proc. 2nd Australian Workshop on Signal Processing Applications,
Brisbane, Australia, December 1997, pp. 195–198.

Andrew J. Beaumont-Smith and Neil Burgess,
"Modified Kogge-Stone VLSI adder architecture for GaAs technology",
Proc. European GaAs and Related III-V Compounds Application Symposium,
Paris, France, June 1996, pp. 2A1.

Chapter 1

Introduction

Speed isn't everything, it's the only thing.
Seymour Cray.

MICROPROCESSORS execute code across application domains which pass through single stream processing, transaction processing and high performance technical computing (HPTC). The design of a microprocessor and system can target any one or all of these application domains for increased performance. The role of the microprocessor designer is to find a good solution within the design space of **technology**, **architecture** and **software**. This generally translates into maximising average performance for target benchmark programs within a power budget.

The demand for computational throughput has outpaced the currently available technology because of the continually increasing size and complexity of the problems that need to be solved by scientists and engineers. Such problems include numerical simulations with thousands of variables like those from nuclear physics and weather forecasting, and control algorithms with hundreds of variables including Kalman filtering and signal processing algorithms such as the discrete Fourier transform (DFT). Many of these algorithms are used in systems which perform real-time signal processing [HX96]. A recent HPTC application was the first linear mapping of the human genetic code [Nat01] which was conducted using over 600 Compaq Alpha microprocessors arranged in a cluster.

Computer system performance for many technical computing problems can be improved by exploiting data parallelism in algorithms which contain vectorisable code and compiling

the program for execution on parallel computer hardware. Parallel processing involves the separation of a problem into parts that can be computed in parallel to achieve a speed-up in execution time over what would be obtained if the problem were mapped onto a sequential machine. The amount of speed-up is determined by many interrelated factors encompassing the hardware architecture of the parallel processor which includes the number of data processing units, a memory system and the interconnection network that links them together. The speed-up also depends on the available instruction level parallelism (ILP) in the program, whether the algorithm is vectorisable and the compiler's ability to expose the parallelism and efficiently map the algorithm onto the parallel processor. Discussions on parallel processing can be found in [CTP96, AG94, Kri89, SFK97, PH96, IT89, HF84].

Special purpose parallel processors target maximum performance within a technology constraint for a particular class of algorithms. Systolic array processors such as SCAP [CCMC92] or DSP chips optimised for specific tasks such as the Texas Memory Systems TM-66 [MM96] are examples of this.

1 Technology

Complementary metal-oxide silicon (CMOS) has been the favoured underlying technology for building microprocessors and computer infrastructure for more than two decades [WE95, Sze88, PE88]. Rapid improvements in the speed and density of CMOS semiconductor devices has seen the clock frequency of microprocessors double every two years and they are expected to reach around 12GHz by 2012 [SIA99]. The clock frequency of a clocked microprocessor chip is the most important design parameter in the quest for higher performance.

Gordon Moore observed in 1965 that the number of transistors per square inch had doubled every year since the integrated circuit was invented and he predicted that this trend would continue [Moo65]. The pace of development did slow down, however memory density has doubled every 18 months and provides the current definition of Moore's law. This pace may slow because eventually the technology will near the atomic scaling limits and alternative fabrication technologies and techniques must be used. In 1989, the one Million transistor IC was published [KF89] and in 1999, an article describing the design and technology of a 100 Million transistor IC was published [Gep99], a milestone in the semiconductor industry. The density of transistors on a single chip has increased at the average rate of 60% per year and is expected to exceed one billion by 2012 [SIA99].

The shrinking (or scaling) of device size and an increase in the number of devices available

on a chip has led to the notion of a 'system on a chip' (SOC) where more peripherals of the microprocessor including cache memories, memory controllers and data routers are integrated on-chip [JAB+01]. The floating-point unit was once a separate chip demonstrating how far the technology has advanced. The integration of more computer system components on a single die has seen an increase in reliability and bandwidth between components, and a reduction in data transmission latency. These are key requirements for building a high performance single-chip parallel compute engine.

2 Architecture and Data Parallelism

The available parallelism for both instructions and data may be classed as either functional parallelism, where program flow and data flow are irregular (except loops) or data parallelism, where data structures allow parallel operations on elements and data flow is regular. Functional and data parallelism together with architectural techniques to support them are discussed. The focus of the work in this thesis is exploiting data parallelism to achieve high performance for a particular class of algorithms. Discussions on computer architecture can be found in [HJS99, Soh98, SFK97, PH96, PH98, HSS90, Mur90].

Architectural techniques used to increase microprocessor performance in the context of data parallelism include pipelining, replication of functional units and implementing separate instruction and data caches.

In the von Neumann model of a computer, the fetch, decoding, execution and memory transfers are carried out sequentially which means the actual data processing part of the CPU is only partially utilised. In modern computers, architecture optimisation can be achieved through pipelining [Kog81] where successive independent instructions can be executed in parallel where no branches or resource conflicts occur. For N pipelining stages, the throughput is increased by N under these assumptions. Replicating resources or functional units can also increase performance depending on the amount of concurrency available in the algorithm to utilise the extra resources. Both techniques of pipelining and replication can be used to increase the performance of processors that exploit data parallelism. The limits of pipelining are reached when dependent instructions in the program cannot be scheduled further apart than the pipeline depth, which causes an instruction issue stall. This is most likely to occur for small problem sizes in a data parallel processor. The limitations of replication are related to the additional control and routing complexity.

The latency associated with fetching instructions and operands from main memory is long

compared with fetching from local registers in a load-store architecture. The modified Harvard architecture utilises two fast cache memories between the microprocessor and main memory. One is for caching instructions and the other for data and this provides a low latency and high bandwidth supply of instructions and data to the microprocessor core. Additional cache levels may be added with increasing size and latency to further improve overall processor performance. The data memory in a data parallel processor can be highly pipelined as long latencies can be tolerated where load addresses are deterministic and can be calculated well before execution of the load operands.

Superscalar microprocessors are functional parallel machines which dynamically uncover and exploit ILP to fetch and issue up to eight instructions in parallel per clock cycle to improve the average instructions per clock (IPC). Discussions on superscalar processors can be found in [Joh91, PH96]. Superscalar machines have multiple execution units, extended data-paths and may implement out-of-order execution. Hybrid machines can be constructed to achieve higher performance or lower power for a given instruction set architecture (ISA). For example complex instruction set computers (CISC) can be microcoded [Pap96], translated into simpler reduced instruction set computer (RISC) instructions [JBD+01] or code-morphed into a very long instruction word (VLIW) -native machine [GP00]. The high control overhead needed to implement wide superscalar machines has limited the issue width to eight [Wal90].

Microprocessor architectures which exploit data parallelism to increase performance include vector and systolic machines. Vector processors can specify multiple independent operations on linear operand arrays in a single instruction [PH96]. An array of N parallel pipelined arithmetic units can produce N results per clock cycle. Vector machines can easily be scaled as much as technology allows with little increase in control overhead. Systolic processors are characterised by a rhythmic and regular movement of data between processor nodes, have simple control and are optimised for a particular class of algorithms [Kun88].

The class of ISA affects the complexity of the machine decode and control, two classes of ISA are discussed. RISC machines generally implement a subset of a CISC ISA. This means the decoding of the instruction and the control of the machine are simpler and the clock rate can be made faster. The substantial area savings made from reduced control can be used for integrating cache memory or adding more execution units for superscalar processing; the trade-off is larger executable code. VLIW processors [Fis83] reduce the control complexity of executing multiple instructions per clock cycle. In each cycle a VLIW instruction is fetched and executed. The VLIW contains multiple operations which are issued in parallel and the VLIW compiler finds dependent operations and packs them into a single VLIW instruction taking into consideration the intra-instruction and inter-instruction dependencies. The VLIW

machine compiler must have information about the details of the hardware implementation to perform the instruction scheduling. This is a disadvantage as code must be recompiled for a different VLIW machine hardware target. It also requires a large VLIW instruction cache with high fetch bandwidth to store and supply the many permutations of instructions.

Data flow machines do not have any instruction sequencing (or scheduling) but are driven forward by data dependencies. A computation node "fires" whenever data is required for computation. It is a network of data processors with local data memories and no instruction processing where results are communicated but memory is not shared.

The classic taxonomy of computer systems was produced by Flynn [Fly72] and classifies a computer by partitioning the instructions and data streams into single and multiple. Madisetti [Mad95] discusses digital-signal multiprocessors (DSMP) and control techniques for decoupled, polycyclic and superscalar microprocessors. A modern taxonomy of processor architectures is given in [SFK97]. Figure 1.1 shows a classification of parallel computer architectures by data or functional parallelism. Each sub-group is broken down according to the most distinguishing feature such as granularity, control scheduling, etc.

Large-scale computer systems can be generally divided into six categories [Fly72, HX96]:

- SIMD (Single Instruction - Multiple Data)
- PVP (Parallel Vector Processors) eg. Cray C-90, T-90
- SMP (Symmetric Multiprocessors) eg. Cray CS6400, IBM R30
- MPP (Massively Parallel Processors) eg. MasPar
- COW (Clusters of Workstations) eg. Beowulf Clusters
- DSM (Distributed Shared Memory Multiprocessors) eg. Cray T-3D

The last five are multiple-instruction multiple-data (MIMD) machines. Single-instruction, multiple-data (SIMD) machines are typically special purpose machines such as many systolic arrays [CS94, Kun88]. In a traditional SIMD machine, a controller sends all nodes the same instruction and the processing nodes operate synchronously in lock step on possibly different data sets. In the case of a DSMP system [Mad95] the program blocks are transferred to each processor node and executed thus minimising the communication overhead of broadcasting the instructions from the controller. For most general purpose applications, a MIMD machine is used where parallel processes execute possibly different instruction code on each

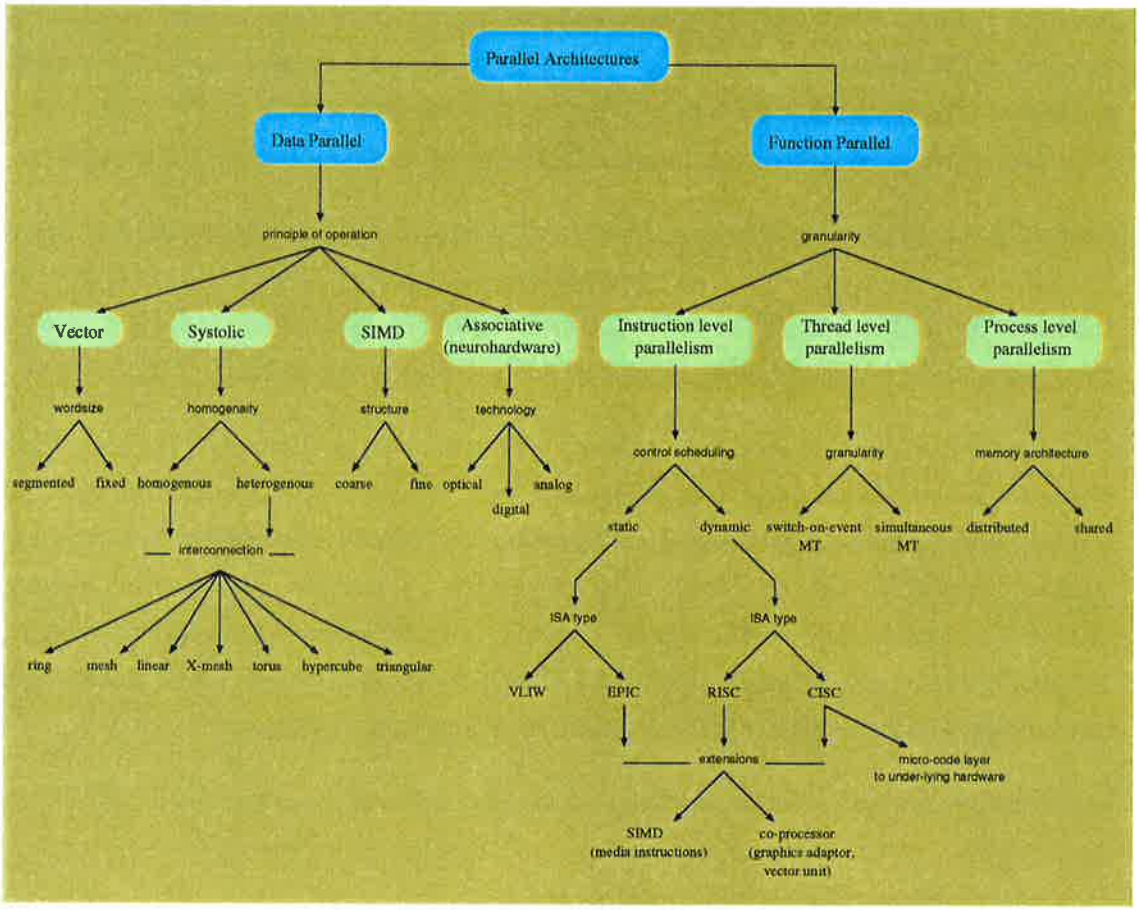


Figure 1.1 Taxonomy of parallel computer architectures.

processor autonomously. This means that processes are asynchronous as they operate independently but must be forced to wait for one another when inter-process dependencies arise. This is achieved through a set of synchronisation operations such as semaphores, barriers or communication blocks. DSM machines are based on the Stanford DASH architecture where the memory is physically distributed among different processor nodes but a cache coherency system for arbitrary block sizes (in the case of the Cray T-3D) which creates the illusion of a single address space to the users. Some DSMs and COWs can be called MPPs if they have sufficient bandwidth.

3 Software

Programming languages used to support parallel processors include OCCAM [Occ88], FORTRAN, parallel C (pC) and parallel C++. These have built-in parallel data types to expose the parallelism in a program to the compiler which can be mapped to parallel hardware. A

discussion of parallel compilers can be found in Polychronopoulos [Pol88] and a examination of programming parallel processors can be found in Babb [Bab88].

The programming models used on parallel computer systems are [Zor92]:

- the distributed computing model where the problem is divided among various nodes and there is relatively little communication. This is best used on COWs [Com].
- a message passing model where many processors in an MPP, for example, cooperate in solving a problem by sending messages of data or code to each other through a network, where routing information in a message guides it through the network. This sometimes occurs through other processors depending on the network topology.
- a data parallelism program is neither distributed nor message-passing and referred to the way SIMD machines were programmed. MIMD machines may be programmed in this style and programs may or may not be executed in lock step.

Microcode is a software hierarchy pioneered by M.V. Wilkes [Soh98] in the early 1950s. High-level instructions are executed in a number of micro-cycles and allows a high-level instruction set architecture to be executed on different underlying hardware.

The design of computer hardware is greatly influenced by the programs or target benchmarks it is required to run and emphasis can be placed on the architectural and software features to achieve a good high performance solution. Programs which perform matrix computations employ routines which have a high degree of data parallelism and these are of particular interest to the scientific and engineering community.

4 Matrix Computations

Many problems can be expressed in terms of matrices and solved using matrix operators [GL96] which are fundamental operations or kernel routines that can be performed by parallel hardware. Libraries of kernel routines which implement a standard set of matrix operators are provided with programs such as Matlab [Mat99] and the Linear Algebra Package (LAPACK) [ABB+92, ABB+] and can be tuned to minimise the execution time on a particular computer system.

A large number of engineering and scientific problems are presented in terms of matrices and example applications include real-time signal processing problems such as the fast Fourier transform (FFT) [GL96], Kalman filter [Sor85] for control and the Volterra model [Sch89] for non-linear systems with memory. Other diverse application areas include image processing and weather forecasting. Kernel routines such as the block linear algebra subroutines (BLAS) are built upon to provide a set of usable matrix functions which are called in computational routines such as Gauss-Jordan elimination, Jacobi rotations, LU and QR factorisation. The computational routines are used in driver routines and applications. General purpose matrix driver routines include solving linear equations, finding eigenvectors, matrix inversion and computing the singular valued decomposition (SVD) [GL96]. Figure 1.2 shows how applications are built on top of a software package such as LAPACK and the BLAS libraries.

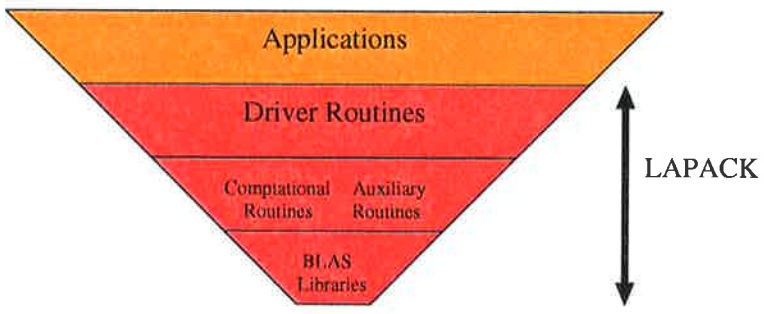


Figure 1.2 Breakdown of applications into LAPACK and BLAS.

The computational routines call library sub-routines, each of which may be executed sequentially or in parallel on a parallel processor. These sub-routines can be classified into multiple levels according the following complexity [GL96]:

- **LEVEL-1 Operators** : These have a linear complexity data and arithmetic relationship in the direction of operation. Examples include the dot product or saxpy (scalar $a \times x$ plus y), matrix transpose and matrix addition/subtraction/division.

- **LEVEL-2 Operators** : These have quadratic complexity data with an arithmetic relationship and are used for matrix-vector operations. Examples include the outer product and gaxpy (general $A \times x$ plus y, where A is a matrix and x and y are vectors).

- **LEVEL-3 Operators** : These have a quadratic data complexity and cubic arithmetic complexity which are targeted at matrix-matrix operations. Examples include matrix-matrix multiplication.

4.1 Linear Algebra Package (LAPACK)

From the introduction to LAPACK [ABB+92]:
"LAPACK is a transportable library of Fortran 77 (which has been translated to C,C++) subroutines for solving the most common problems in numerical linear algebra: systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. It has been designed to be efficient on a wide range of modern high-performance computers. LAPACK is intended to be the successor to LINPACK and EISPACK. It extends the functionality of these packages by including equilibration, iterative refinement, error bounds, and driver routines for linear systems, routines for computing and re-ordering the Schur factorization, and condition estimation routines for eigenvalue problems. LAPACK improves on the accuracy of the standard algorithms in EISPACK by including high accuracy algorithms for finding singular values and eigenvalues of bidiagonal and tridiagonal matrices respectively that arise in SVD and symmetric eigenvalue problems. The algorithms and software have been restructured to achieve high efficiency on vector processors, high-performance "superscalar". A comprehensive testing and timing suite is also provided."

LAPACK can also be used to measure and compare the performance of microprocessor systems for high performance technical computing applications. An example is the Gaussian elimination process to solve a large matrix (1000×1000) which is compiled and executed on various machines to compare processor utilisation, average compute rate (MFLOPS) and execution time.

4.2 Block Linear Algebra Subroutines (BLAS)

The block linear algebra subroutines (BLAS) [DCDH98] are designed to provide a more portable and efficient means of implementing algorithms on high performance computers, especially those with hierarchical memory and parallel processing capability. The BLAS

routines can be tailor-made for a machine to optimise the use of the available hardware. For example, the scope of the numerically intensive level-3 BLAS is:

- matrix-matrix products which include rank-k updates of a general matrix
- rank-k and rank-$2k$ updates of a symmetric matrix
- matrix-triangular matrix multiplication
- solving triangular systems of equations with multiple right hand sides.

These operations are defined for real, double precision, complex and double complex numbers and matrices of arbitrary size.

To study the speedup potential which can be obtained from the addition of hardware support for matrix computations, the LAPACK source code and BLAS libraries were modified to add timing instrumentation which would provide the timing breakdown of BLAS calls for LAPACK driver and computational routines. The modifications included counters and timing for each BLAS. LAPACK version 3.0 was modified and compiled from C source code (CLAPACK) using the GNU C compiler (GCC version 2.91.66) and run on an Intel $500MHz$ pentium processor with $256MB$ RAM and Linux version 2.2.5-22 operating system. The study was restricted to the large double precision real BLAS, driver and computational routines in the LAPACK timing suite. The timing routines included (where input matrices are $M \times k$ and $k \times N$):

BLAS routines

- DBLASA : Timing the level-2 and level-3 BLAS (DBLASA-DB2 and DBLASA-DB3 respectively) for small values of k up to 64 and matrix sizes up to 500.
- DBLASB : Timing DGEMM, DSYMM, DTRMM and DTRSM for small values of M up to 64 and matrix sizes up to 500.
- DBLASC : Timing DGEMM, DSYMM, DTRMM and DTRSM for small values of N up to 64 and matrix sizes up to 500.

Linear equation routines

- DBAND : Band matrix triangular factorisation (LU, Cholesky) and linear equation system solver (based on triangular factorisation) of general banded, positive definite

banded and triangular banded matrices for band-widths up to 200 and matrix size of 1000.

- DTIME2 : Rectangular matrix QR, LQ, QL, RQ and QP decompositions and reduction to bidiagonal form for matrix sizes up to 400.

Eigensystem routines

- DGEPTIM : Generalised Non-symmetric Eigenvalue Problem which includes 18 routines to compute the generalised Schur form, generalised upper Hessenberg form, left and right generalised eigenvectors for matrix sizes up to 200.
- DNEPTIM : Non-symmetric Eigenvalue Problem which includes twelve routines to compute the Schur form, left and right eigenvectors and eigenvalues from matrices in Hessenberg form for matrix sizes up to 300.
- DSEPTIM : Symmetric Eigenvalue Problem which includes 23 routines for various tridiagonal matrices for matrix sizes up to 400.
- DSVDTIM : Singular Value Decomposition which includes 18 routines to compute singular values from dense and bidiagonal matrices for matrix sizes up to 200.

Other timing routines in the LAPACK suite include single precision real and complex numbers, double precision complex numbers and small problem size timing routines. The execution time of small problem sizes was swamped by system run time and not considered useful for obtaining benchmark results. The single precision floating-point routines had almost the same run time as the double precision routines. Single precision is generally of less interest than double precision for numerical analysis since most computer hardware is optimised for double precision arithmetic.

Figure 1.3 shows the execution time for the large double precision routines in the LAPACK timing suite. Figure 1.4 shows the percentage breakdown for each BLAS type for each of the timed routines and the total over all routines on the right side of the graph. The percentage of miscellaneous time is shown as MISC and represents system time and the time to generate input matrices which is particularly significant for DGEPTIM, DNEPTIM, DSEPTIM and DSVDTIM. This accounts for approximately 20% of the total run time. The time spent executing BLAS calls constitutes 70% of the total run time and the most notable BLAS calls are discussed below. In the following discussion A, B and C are matrices, x and y are vectors, α and β are scalars.

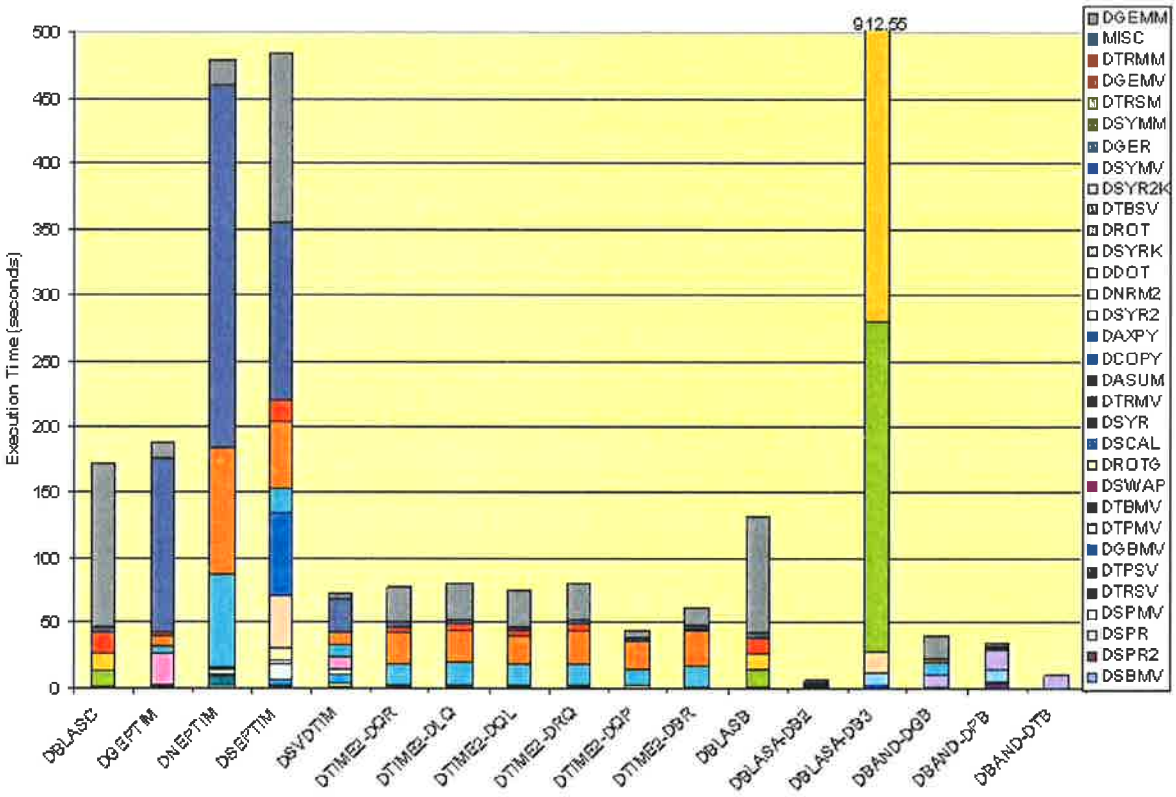


Figure 1.3 Execution time for LAPACK timing routines.

DGEMM - general matrix-matrix multiply :

$$C = \alpha \times A_{m \times k} \times B_{k \times n} + \beta \times C_{m \times n}$$

where A and/or B may be transposed matrices. This accounts for 70% of the execution time in DBLASB and DBLASC and 20% overall.

DTRMM - triangular matrix-matrix multiply :

$$B = \alpha \times A \times B_{m \times n}$$

where A may be a transposed unit, non-unit, an upper or lower triangular matrix. A and B may be swapped.

DGEMV - general matrix-vector multiply :

$$y = \alpha \times A_{m \times n} \times x + \beta \times y$$

where A may be transposed.

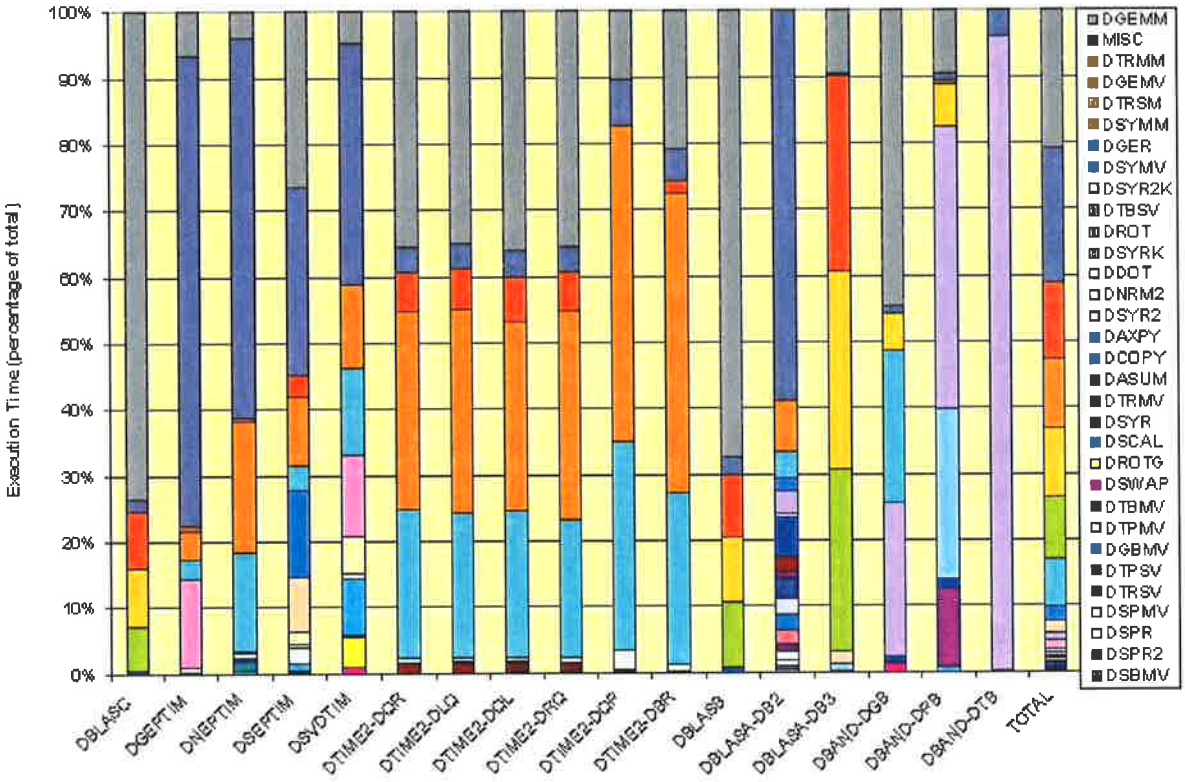


Figure 1.4 Percentage of total execution time for each BLAS.

DTRSM - solving triangular matrix with multiple right hand sides of the form :

$$A \times X_{m \times n} = \alpha \times B_{m \times n}$$

where X and A may be swapped, A is a unit, non-unit, an upper or lower triangular matrix. A may be transposed.

DSYMM - symmetric matrix-matrix multiply (indefinite matrices) :

$$C = \alpha \times A \times B_{m \times n} + \beta \times C_{m \times n}$$

where A and B may be swapped and A is a symmetric matrix.

DGER - rank 1 operation :

$$A = \alpha \times x_m \times y'_n + A_{m \times n}$$

The speed-up potential depends on whether the BLAS, computational or driver routines are targeted. Hardware acceleration of the BLAS will yield an upper bound (assuming the

Table 1.1 Execution time (seconds) and MFLOPS rates for DGEMV and DGEMM BLAS run on various computer systems. [ABB+]

	DGEMV				DGEMM			
	Values of matrix size, $n = m = k$							
	100		1000		100		1000	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
DEC Alpha Miata	.0151	66	27.778	36	.0018	543	1.712	584
AlphaServer DS-20	.0027	376	8.929	112	.0019	522	2.000	500
IBM Power 3	.0032	304	2.857	350	.0018	567	1.385	722
IBM PowerPC	.0435	23	40.000	25	.0063	160	4.717	212
Intel Pentium II	.0075	134	16.969	59	.0031	320	3.003	333
Intel Pentium III	.0071	141	14.925	67	.0030	333	2.500	400
SGI O2K (1p)	.0046	216	4.762	210	.0018	563	1.801	555
SGI O2K (4p)	5.000	0.2	2.375	421	.0250	40	0.517	1936
Sun Ultra 2 (1p)	.0081	124	17.544	57	.0033	302	3.484	287
Sun Enterprise 450(1p)	.0037	267	11.628	86	.0021	474	1.898	527

BLAS routines are now infinitely fast) for the speed-up of approximately 20 for DBLASC but only 1.4 for DGEPTIM where execution time is dominated by non-BLAS procedure calls. A greater speed-up can be achieved if the computational and driver routines are re-written to make best use of the parallel hardware.

Table 1.1 shows the execution time and MFLOPS for the DGEMV and DGEMM BLAS run on ten computer systems [ABB+]. The results for the SGI Origin 2000 machine show that moving from a single to a four processor configuration approximately halves the execution time for DGEMV (vector) and reduces the execution time for DGEMM (matrix) by approximately four. DGEMM scales almost linearly with the number of processors. The execution time is in the order of seconds for all machines tested.

5 Parallel Processors

A parallel processor is made from three basic hardware components: processing nodes, memories and an interconnection structure which are discussed.

5.1 Processing nodes

According to Skillicorn's taxonomy of computer architecture [Ski88] which is based on multi-level classification of a computers architecture and the data flow between its component parts, 28 classes of architectural configurations are possible. The first level architecture classification scheme specifies the number of instruction processors (IP), instruction memories (IM), data processors (DP), data memories (DM), the number of external interface units and the types of interconnecting switches. A second level classification specifies the state diagram for each functional unit which indicates the level of pipelining. The first level of classification can be used to classify processors into four types [Mad95] which are listed in order of relaxation of the scheduling:

- register to register
- horizontal or VLIW
- polycyclic
- decoupled.

Figure 1.5 shows a simple level-1 representation of each superscalar architecture. The register to register architecture in Figure 1.5(a) has no direct connection between the DM and the DPs and accesses are to the DP register bank. Both DPs can be kept busy if the source and result buses as well as the source and destination operands are not needed by both DPs in the same cycle. Memory contention can also affect performance. The horizontal architecture shown in Figure 1.5(b) has separate registers for each DP, each with its own result bus. The crossbar interconnection eliminates bus conflicts and reduces control hardware. The efficiency with which the DPs are used depends on the number of simultaneous reads and writes to register files, access to buses, data dependencies and memory delays. Software pipelining is the scheduling technique used for VLIW processors where data fetches are overlapped with execution to improve performance. The polycyclic architecture is shown in Figure 1.5(c) and delay elements are used as buffers between the DM and DPs. The delay elements can be read

in any order and any number of times but must be written in order of arrival. Destructive reads can also be executed and the delays can improve throughput because shorter clock periods can be used. The software scheduling is static in these first three architectures. The decoupled architecture is shown in Figure 1.5(d) and the restriction on synchronising the IP and DP is relaxed by placing delays between them. This allows the IP to move ahead of the DP and process instructions out of sequence if necessary. The delays must match the results with the labels for storage. This technique is beneficial for software pipelining and the scheduler can be replaced with a dynamic run-time control in software.

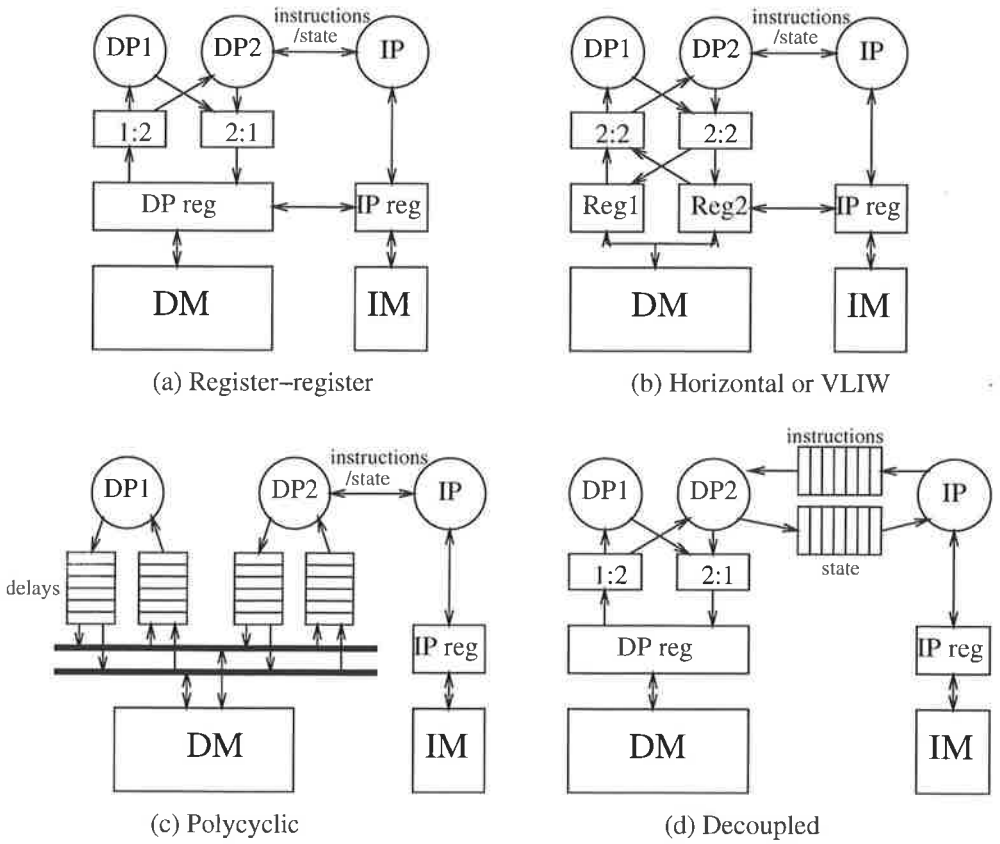


Figure 1.5 Four principal types of processor architectures.

5.2 Memories

The memory system for a parallel processor can be shared, distributed or both. The memory should be fast and wide enough to supply enough operands to the processors to keep them busy, large enough to hold enough data for the size of target problems and ideally be contention free. On-chip memories, caches and local memories are usually constructed from SRAMs while a shared memory can be constructed from either SRAMs or DRAMs.

Several memory systems have been proposed for matrix processors [Mar94, Bet00, To97] and an implementation has been carried out for SCAP [CCMC92]. Marwood [Mar94] describes several address generation schemes based on number theory which can be implemented using modulo counters. These schemes (such as the alternate integer representation) map vectors of matrix operands from a linear memory space to the edge of the processor array. For fast access to large matrix structures from conventional memory components, caches must be employed to speed up the memory-processor bandwidth to prevent the array from being under-utilised. Significant time penalties are paid if the cache miss rates become too high [CCMC92]. The memory-memory architecture [To96] is a memory system which contains interleaved banks of Rambus RDRAMs [KOS+93, RAM93a, RAM93b] connected to an address generator system through a bank of address queues. A major problem is the complexity and hardware cost of the system to obtain low latency deterministic memory behaviour because of the high probabilities of contention due to low memory partitioning. Additional hardware must be used to detect or predict and solve memory contention.

Betts [Bet00] provides an elegant solution to this problem with an inverted address generation technique where operands are supplied from a high bandwidth Rambus channel and re-ordered into two SRAM banks on two edges of the array by an intermediate memory controller. The large amount of operand re-use contributes to the high efficiency of this system although on start-up the array is under-utilised depending on the number of Rambus channels available. It is not clear how well this architecture can be scaled given that the memory controller is not distributed and all operands must pass through it.

5.3 Interconnection networks

The interconnection network links the processors and memories together. It provides a means of passing data between processors and memories (and other processors) as well as control information and messages. Discussions on processor interconnection topologies can be found in [Sie85, Qui87] and topologies which are applicable to matrix algorithms are reviewed below.

5.3.1 Loosely coupled memory: hypercube

An n-dimensional hypercube structure is a binary n-cube network which contains $2n$ nodes and an example of a four-hypercube is shown in Figure 1.9 (black dots are processor nodes). Each processor has an n-bit address and is directly connected to n adjoining processors, therefore it must have n ports. The addresses of adjacent nodes differ by one bit. A ten-dimensional hypercube can also be implemented by assigning two processors per node, as in the case of a TMS 320C40 [Jai94] for example. A MPI must be implemented since there are no shared variables or memories to communicate. The key advantages of a hypercube network are:

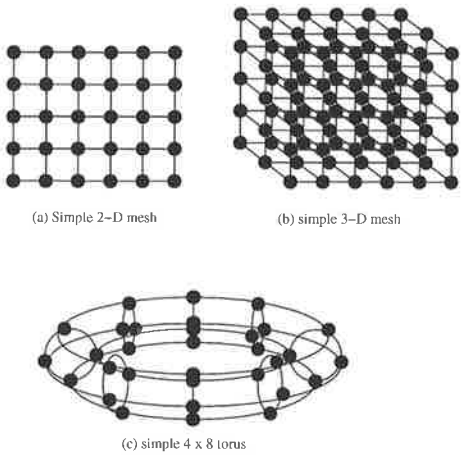


Figure 1.6 Three simple mesh networks.

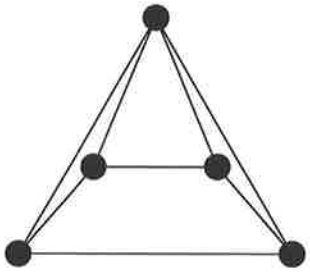


Figure 1.7 Pyramid network building block.

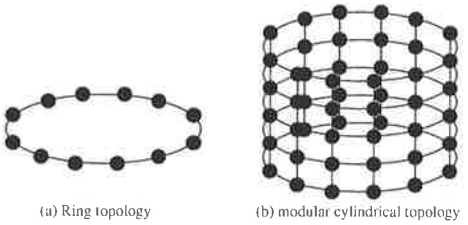


Figure 1.8 Ring based networks.

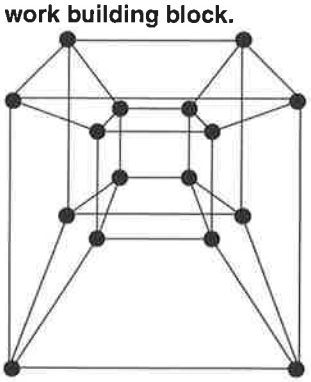


Figure 1.9 Four-hypercube interconnect topology.

- easily mapped and efficiently implemented
- shortest path separates the two farthest nodes
- alternate paths can be used to minimise network congestion
- flexibility in expanding and contracting the network without significant software or hardware modification or cost
- fault tolerant if nodes and links break.

The major drawback of the hypercube structure is that the number of ports scales linearly with the dimension of the network which reduces the I/O port width to each neighbour given a fixed number of I/O pins per processor. This makes practical network scaling difficult. One further disadvantage is that the number of processors must double each time the system is scaled up.

5.3.2 Simple mesh

Mesh interconnect topologies are the most popular for linking processing nodes together. They are a 2-D planar mesh, 3-D mesh and a torus which is a wrapped 2-D mesh shown in Figures 1.6(a)–(c). Mesh structures have been used for interconnecting processing nodes in systolic arrays and parallel processing systems [Zor92, JAB+01, CCMC92]. The interconnection is localised and the structures are readily implemented on planar substrates. The system is easily expandable to incorporate more processing nodes. In a 2-D mesh, two ports per node are needed for corner nodes, three for edge nodes and four ports are needed for the remainder. In a 3-D mesh, three ports per processor are needed for corner nodes, four ports for edge nodes, five for face nodes and six for internal nodes. The mesh diameter is of order $log(n)$ where n is the number of nodes. Mesh structures are a preferred structure for matrix multiplication algorithms due to the simple communication paths and algorithm mapping [Zor92].

5.3.3 Pyramid architecture

A pyramid structure uses the basic building block shown in Figure 1.7 to recursively build a higher level pyramid network. They are widely used in image processing where a large set of data is decomposed. The node at the top is called the parent and all the ones on the bottom are the children, each parent has four children which are connected in a mesh. The base level of the pyramid has $2n \times 2n$ nodes and each level starting at the top ($n = 0$) has $2n \times 2n$ nodes. The number of links per processor varies depending on where it is in the pyramid. For the case of $n = 2$, nodes with four, five and seven links are needed.

5.3.4 Ring architecture

A ring architecture is a wrapped array of processors where the first and last edges are joined. The advantages over a simple mesh are that the maximum inter-processor communication paths are halved, there is an alternative link to send data which is useful for fault tolerance because messages are re-routed in the opposite direction from the broken link. A simple ring shown in Figure 1.8a only needs two links per processor (more if redundant links are made) and the cylindrical architecture requires four and three links per processor for centre and end nodes respectively. The cylindrical architecture may be wrapped to form a torus.

5.3.5 Mesh of trees

The leaves of a binary tree are an n^2 mesh for a mesh of trees. There are $3n^2 - 2n$ processors and each requires three links (two for leaf and root nodes). As the network size increases the number of communication ports increases linearly (as with hypercubes) which means this network is not easily scalable.

5.3.6 Fat tree

The processors in this architecture are like leaves on a tree and the links are the branches which have more bandwidth the farther away from the processors they are. The Thinking Machines CM-5 [Hil85] uses a fat tree topology but has thick branches closest to the processors on the basis that the local communication will be the greatest. There are two networks on this system: one to transfer data, and the other for broadcasting, synchronising and coordinating control, and sending instructions to the nodes. The Fat Tree has only been used for MPP implementations which are for general purpose use [Zor92].

5.3.7 Reconfigurable links

All topologies reviewed (except the torus and the Fat Tree) require a variable number of communication ports per processing node. If a fixed number of pins per processor is available to transfer data and all links are of a fixed width, there will be spare buses available on some nodes. These can either be used for external access by a host processor or two processors per node could be used at the nodes requiring a larger number of links. Another solution is to have reconfigurable links which can be switched either before processing begins (deterministically) or dynamically. The parameters which describe the switch are the degree (2×2, 3×3 etc.), number of settings, width and crossover capability (number of different paths that can be simultaneously connected). Switch architectures can be classified into one of four types:

- one to one
- n to n - i^{th} unit of one switch set can connect to the i^{th} unit of the other switch set
- one to n
- n by n where each unit of one set can connect to any other unit of the other set and vice-versa.

The dynamic reconfigurability of the switch can be controlled by packet switching (destination in the data header) or circuit switching (pre-established). Packet switching can add a large circuit penalty if a special router is needed, however switches are considered to be cost-effective in implementing MPP [Jai94].

6 Review of Parallel Processors

For current generation processors, the compute rate available on-chip (largely due to pipelining) is not matched by the I/O rate although on-chip cache memories alleviate this problem [McC95]. However, for algorithms which have a low re-use of input operands or poor cache performance due to the algorithm or problem size, bottlenecking in the system performance occurs because it is memory bound. This memory to compute rate imbalance has led to the development of optimised processors for some specific tasks. One such chip is the Texas Memory Systems TM-66 [MM96] which has a $1.25GFLOPS$ peak compute rate and a total external bandwidth of $960MB/s$ and is optimised for real and complex FFTs, convolutions, correlations and matrix multiplies. It is primarily designed to perform FFT butterfly operations but is sub-optimal for matrix multiplication due to a fixed data-path. Other machines such as HiPAR [WOK+97] are a single chip solution to image processing problems as they employ a 16-bit data-path array, caches and a large matrix memory on-chip connected by buses.

The data processing part of the node can be classified as follows:

- general purpose commercial microprocessor (eg. Intel i860 in the Paragon)
- high performance/specialised commercial microprocessor (eg. InMOS T9000)
- DSP (eg. Texas Instruments TMS320C40)
- custom (eg. nCube 2S, Cray Y-MP, DSTO SCAP).

DSP chips are a popular processor choice for implementing real-time systems because they are inexpensive and small, utilising commercial microprocessor cores with additional pipelined execution units to increase the computational performance [GMS+94]. Examples include multiprocessor DSP systems [Dev96, Jai94] and Aladdin [Ter95] which was built in 1994 by Texas Instruments and integrates 50 RISC processors and memory in 4.5×3 inches (liquid cooled) with a performance of 1.6GFLOPS.

In this section, several key parallel processor systems and chips used to implement these systems are reviewed.

6.1 Clusters of Workstations

Workstation based multiprocessor systems offer hundreds of MFLOPS of processing power each but the CPUs must manage the OS, graphics displays and possibly other users programs which makes them relatively low in efficiency. The links between workstations such as Ethernet are relatively slow (100Mb/s) compared to the speed of data transfer in a highly integrated system (order of Gb/s). This coupled with a software Message Passing Interface (MPI) overhead are the largest drawbacks of these systems. They are most suited to problems that can be partitioned to require little message passing.

6.2 InMOS T9000 Transputer

The T9000 Transputer [Tra88, Kno91] was the last transputer chip built by Inmos Ltd. and is a complete computer in a single VLSI chip. It supersedes the earlier T805 transputer. The T9000 integrates over 2 million transistors and contain three major subsystems: a pipelined 32-bit superscalar processor with a 64-bit, 25MFLOPS FPU, a communications processor with four 70MB/s communications links and a 16KB fully associative cache. The sub-systems are connected together using a crossbar switch. The T9000 was designed to form part of a multiprocessor system and the instruction set architecture is tuned to the execution of concurrent programs. T9000 chips may be connected via a network of packet routers which allows an almost unlimited number of virtual channels to be established between transputers. There are two levels of caching, the first level is on the chip since one-third of all accesses made by the processor are to the workspace of the currently executing process. The second level write-back cache sits between the major functional blocks and the external memory system and allocates on a write miss to make all accesses to main memory of complete cache lines. The maximum data rate from external memory is 200MB/s. Each bank of the cache can access one word every cycle, so the on-chip memory bandwidth is 800MB/s. A workspace cache which caches the first 32 locations of the workspace is used for fast access to variables (as fast access as local registers in a register machine). The internal memory is divided into four 4KB banks which can also be configured as pure RAM to improve the predictability of execution time. The crossbar has nine ports as follows: 1 for PMI, 4 for the processor, 3 for the virtual communications processor (VCP) and 1 shared between the scheduler and the control unit. The instruction set is arranged as a 4-bit function and a 4-bit operand and was chosen to achieve compact program representation. The semantic content of many transputer instructions is low, making pipelining difficult. Several dependent instructions are grouped together and executed on a pipeline and up to eight instructions can be grouped in this way. Four instructions can be fetched per clock cycle so the peak sustainable execution rate is 200MIPS with a 50MHz clock rate. The VCP accepts high-level communications commands from the processor and translates them into sequences of packet exchanges on the serial links. To ensure consistent behaviour, interlocks are required to synchronise data with the CPU.

6.3 Texas Memory Systems Viper-5 (TM-66)

The Viper-5 vector processing system [TMS96, MM96] was manufactured by Texas Memory Systems in 1996. The Viper-5 has from one to four 'S2' vector processing nodes each with a 1.25GFLOPS compute rate. The vector processing nodes contain two custom designed TM-66 (swiFFT) chips, a scalar processor which operates as a supervisor, 2GB shared system memory and six I/O ports each with 200MB/s bandwidth. The TM-66 chip architecture is shown in Figure 1.10 and is optimised for real and complex FFTs (radix- 2,3,4,5,6,8,16), convolutions, correlations and matrix multiplies. The TM-66 is controlled from a VLIW field in the vector processor and contains twenty 32-bit IEEE floating-point units and six 32-bit ports at 40MHz for a total external bandwidth of 960MB/s. A program development software

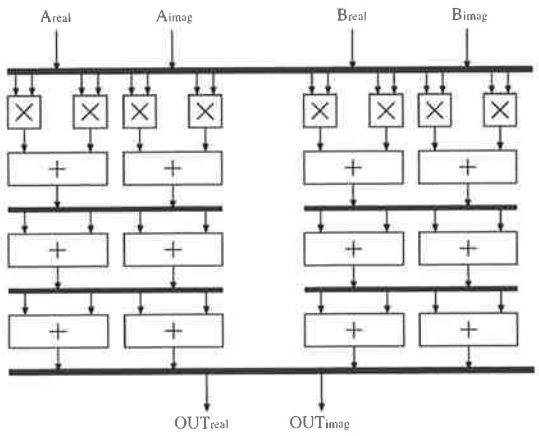


Figure 1.10 TM-66 chip architecture.

includes compilers, linker and assemblers for C and Fortran and an optimised math library. FFT capabilities are from 64 to 64 million points, Table 1.2 shows the measured performance of the vector processing node and the TM-66 chip. The peak performance is 680 MFLOPS.

6.4 Scalable Array Processor

Warren Marwood [Mar94] summarises the scalable array processor (SCAP) system which was developed by the Defence Science and Technology Organisation, Adelaide, South Australia. SCAP [CCMC92] is a 32-bit floating-point pipelined systolic array matrix processor that operates as an SBus device in a SUN SPARCstation. The architecture of SCAP is a 2-D mesh-connected array (or inner step processor) which is supported on the boundary by address generators, caches and a host interface. Each processing element chip contains 20

Table 1.2 Viper-5 (vector processor) performance benchmarks.

Application	Size	Execution time (ms)
FFT (complex)	$1K$	0.04
FFT (real)	$512K$	19.7
matrix multiply (real)	512×512	210
matrix multiply (complex)	512×512	839
dot product (complex)	$64K$	0.92

Table 1.3 SCAP system performance.

Application	Execution time (ms)	Performance (MFLOPS)
FT (1–D, 3540 point using 2D factorisation)	20	130
FT (2–D of 380x380 image)	1385	66
FIR (4000 tap, $1K$ points)	35	210
Polynomial evaluation (10^{th} order, 60×60 complex matrix)	136	114
QR factorisation of 59×60 matrix	561	87
SGEMM (600×600 BLAS)	5139	84

single precision floating-point processors arranged in a 4×5 array which are capable of multiply-accumulate, multiply, add and subtract. The clock rate is 20MHz and the peak processing rate of the chip is 20MFLOPS. Arbitrary size arrays can be made (a 5×4 array of chips achieves 400MFLOPS) and the PE chips are relatively low power at 0.5W/chip. The system implemented integrated a 4×5 array of PE chips on a MCM-C for a total of 400 PEs. There is also a custom data controller chip that supports matrix addressing. The measured performance of the implemented SCAP system is shown in Table 1.3 [CCMC92].

The performance of the SCAP system is half to three quarters of the performance of a CRAY-2 for the BLAS SGEMM routine. Several performance problems were identified:

- the interface through the device drivers is slow, system calls can take longer than the time to execute the matrix operations
- the penalty of a cache miss is very high

- overall system performance would be improved if high performance scalar/vector processing were provided on the cached side of the bus.

This means that there should be tighter coupling between scalar, vector and array processors in such a multiprocessor environment.

6.5 Systolic processors

Systolic processors were first introduced by Kung and Leiserson [KL78]. An extensive treatment of systolic processor techniques is given by Kung [Kun88] and Moore [MMU86]. Recent implementations have been reviewed by Marwood [Mar94]. Dedicated matrix processors include systolic machines derived from the concept of the Whitehouse *et al.* engagement processor [WS81]. A drawback with these machines is that they are not flexible or reconfigurable enough to be useful for a wide range of algorithms (and in some cases, problem size). As a result, configurable systolic machines were designed with switchable interconnections including the configurable, highly parallel computer (CHiP) [Sny82] and the programmable systolic chip (PSC) [FKM+83]. The Warp processor [Tse90, BCC+88] is a linear systolic array of processing nodes with two-way links where each node is a $10MFLOPS$ data-path and a microcoded controller with a horizontal instruction set. A high-level language compiler generated code for each node. More recently, the scalable array processor (SCAP) [CCMC92, Mar94] which was discussed in the previous section, broadened the range of algorithms by using novel address generation techniques and a fast memory subsystem to overcome the memory bandwidth bottleneck. In spite of the favourable compute-to-bandwidth ratio of such a processor, the memory bandwidth required to sustain the computation rates that can be achieved with a moderately sized array of processors poses a number of challenges to current technologies.

6.6 Summary

The compute rate, I/O bandwidth and execution times of selected algorithms for some of these machines is shown in Table 1.4.

Table 1.4 Comparison of Fourier transform and matrix multiplication performance on various processor systems. († - run using MATLAB)

Manufacturer	TI	Alacron	Sun	DEC	DSTO	TMS
Machine	TMS320	FT-SHARC	Ultra 170	Alpha	SCAP	Viper S2
Type	DSP	DSP	workstn.	workstn.	coproc.	RTS
Processing node	320C6701	ADSP-2106x	Ultra SPARC-1	21064	custom	TM-66
Number of processors	1	8	1	1	20	2
Clock rate (MHz)	166	40	167	300	20	40
Performance (MFLOPS)	1000	960	300	-	400	1250
I/O Bandwidth (MB/s)	400	640	-	-	-	-
FFT(1K comp)	108μs	72μs	4s†	0.48ms		80μs
FT (3.5K real)					20ms	
Mat.mult.(512x512)			3.5s	5s		210ms
Mat.mult.(600x600)			15s†		5.1s	

7 Thesis Outline

A wide range of problems can be decomposed into matrix-based kernel routines which exhibit a high degree of data parallelism. This means that high compute rates and processor efficiency can be achieved for signal processing and other algorithms which have high parallelism and incur no fine-grain data dependence on control flow. In particular, matrix multiplication dominates the execution time for many algorithms (over 90% in many cases) because it is an order (N^3) problem with a high compute-to-bandwidth ratio. Significant speed-up can be achieved by mapping this problem to dedicated parallel matrix processor hardware. The term "MatRISC" (Matrix RISC) was first coined in [Mar94] to describe a mesh-connected array of processing elements (PEs) cooperating to perform a limited range of matrix operations optimally, as measured against some cost function.

This thesis presents a generalised architecture for a MatRISC array processor and a CMOS implementation of a specific machine which extends the notion of a vector processor to a matrix processor.

Chapter 1 discusses the design space of a microprocessor and the three elements: technology, architecture and software. Applications that require matrix computations are discussed and decomposed into driver, computational and library routines. The block linear algebra subroutines are studied to determine their usage by computational and driver matrix algorithms. A review of parallel processors is given and a summary of key machines.

Chapter 2 presents the generalised architecture of the MatRISC processor array, discusses the technology constraints, explores the key kernel algorithms and gives simulation results of an abstract machine representation for various hardware configurations. Matrix multiplication and element-wise kernel algorithms are developed for the MatRISC processor and the performance of these algorithms is presented. A high-level design study of the generalised MatRISC architecture is given showing peak and average performance for a range of matrix sizes.

Chapter 3 discusses the microarchitecture and CMOS implementation of MatRISC-1, a 4×4 array of processing elements. Each component of the processing element is discussed in detail and a VHDL model of the processor array is developed. The results of fabricating parts of the processing element in a CMOS process are presented.

Chapter 4 presents the software hierarchy of the MatRISC processor. A set of software tools was developed to support compilation, execution and debugging of code on the processor array. A Matlab-style language called Mcode is presented for use with the MatRISC processor.

Chapter 5 contains a detailed discussion of integer parallel prefix adders and IEEE floating-point adder architectures. An algorithm to synthesise a wide range of parallel prefix adder architectures is developed which covers the complete range of design trade-offs. The delay versus area of 16, 32 and 64-bit adders is graphed for a $0.25\mu m$ technology. New end-around carry parallel prefix adders are also synthesised and presented. Three IEEE double precision floating-point adders are presented which use a flagged prefix adder which combines the addition and rounding of the significand. A three cycle latency adder was designed and implemented in $0.5\mu m$ CMOS and meets the IEEE-754 standard. A two cycle dual-path floating-point adder is presented which splits the computation into a near and far path based on the exponent difference and also uses a flagged prefix adder. A three cycle floating-point adder which accumulates operands in a two cycle loop is also presented.

Chapter 6 presents the mapping and performance of application algorithms mapped to the MatRISC architecture. The Volterra model is a non-linear system with memory. The second order Volterra model has been expressed in terms of two dimensional matrices and it is shown that higher order models are also expressed in terms of the second order model. Other algorithms include the Fourier transform, Kalman filter, LU decomposition and Gauss-Jordan elimination.

Chapter 7 summarises the work and suggests future directions for research.

Additional material on the MatRISC processor project can be found at the following web-sites:

http://www.eleceng.adelaide.edu.au/chiptec/matrisc
http://www.eleceng.adelaide.edu.au/Personal/abeaumon/matrisc

8 Contributions

The novel contributions of this thesis are in the following areas:

Architecture

- Generalised parallel array processor architecture for executing matrix-based computations and systolic algorithms as a kernel algorithm with high performance.
- Architectures for IEEE-754-compliant floating-point adders and accumulators using a flagged prefix adder which merges the significand addition and rounding.
- An algorithm for the synthesis of parallel prefix adders with and without the end-around carry.
- Mapping of the design space for parallel prefix adders for speed, area and valency (cell-radix) trade-offs under a technology constraint.
- A 16-bit RISC controller for MatRISC processors with guaranteed single cycle instruction execution to maintain program synchronisation between processing elements.
- A generalised mapping engine which, in conjunction with the Mcode compiler can sequence control for data parallel algorithms.

Algorithms

- Matrix multiplication algorithm implementation on a MatRISC processor.
- Expressing a second order Volterra model in terms of matrices and extending this to higher order Volterra models and implementation on a MatRISC processor.

Implementation

- Dynamic bus and crossbar switch for routing data on the processor array in conjunction with a pipelined non-blocking memory.
- A digitally controlled clock synchronisation circuit.

Software

- A language to express matrix operations and data structures.
- A compiler to map kernel algorithms to the processor array which builds a minimum set of very long instruction words and generates instructions for the RISC processor in each processing element.

Chapter 2

MatRISC Processor Architecture

MATRIX Reduced Instruction Set Computer (MatRISC) is an architectural machine concept for exploiting the parallelism within matrix-based data structures and algorithms to speed up the execution of real-time, signal processing and high performance technical computing (HPTC) programs. This chapter details the architectural concept of a MatRISC processor, discusses the technology constraints, explores the key kernel algorithms and provides simulation results of an abstract machine representation for various configurations. The work done in Chapters 1-5 has been reported in [BSMT+98, BSLL+97].

1 Abstract Machine Architecture

The proposed MatRISC processor is an integrated memory/processor array interconnected by a mesh of high speed switches. Abstract processor models for the data-paths only are shown in Figures 2.1 and 2.2 for the two-dimensional and three-dimensional cases. The

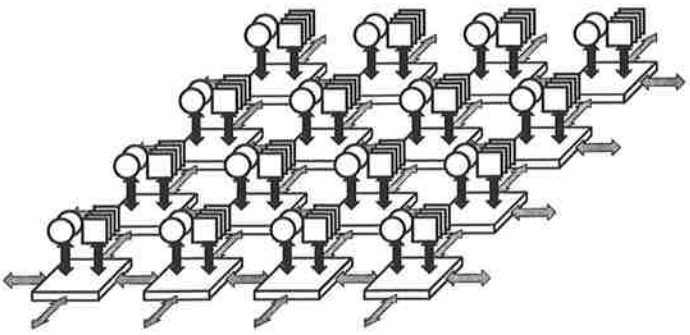


Figure 2.1 Abstract 2-D MatRISC processor model.

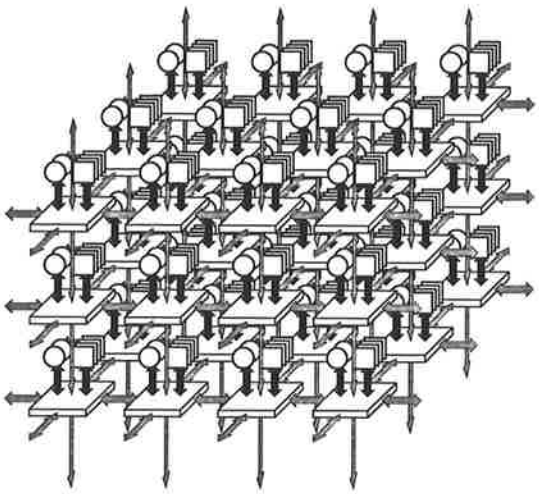


Figure 2.2 Abstract 3-D MatRISC processor model.

switches are represented as blocks and arrows indicate a bus. Each data memory node is represented by a circle. The data memory and switches together form a multi-dimensional memory array (MDMA). This provides a high bandwidth supply of input data to the data processors (DP) which are represented as squares. The MDMA is designed to match the memory bandwidth with the sustainable peak computation rate and overcome memory bank contention issues for the set of kernel matrix-based operations.

1.1 Control instruction

At the level of the abstract machine, control instructions for the processor array include the data memory (as read/write, number of ports and input addresses), data processors (as type, number of functional units and operation) and switches (as configuration and direction). As discussed in Chapter 1, *data-independent* kernel matrix operations have control instruction sequences which can be expressed in terms of a closed form iterative expression. These expressions when mapped into hardware produce a deterministic flow of control instructions which have no data dependencies. *Data-dependent* operations have control instruction sequences which contain data dependencies (eg. unbounded iterative convergent algorithms) which can disrupt the flow of control due to feedback paths. Data-independent control instruction sequences will also exist which do not map into the available hardware due to their complexity.

Most algorithms contain a mixture of data-independent and data-dependent control instruction sequences. Data-dependent control instruction sequences are always under program control whereas data-independent control instruction sequences may be under program control or issued by a *generalised mapping engine* (GME). The GME is a complex programmable

finite state machine with no inputs that depend on data operands or results.

The two control mechanisms for the MatRISC processor are:

- **microprocessor control:** A machine that executes stored instructions and has an interface to the GME and MatRISC control instruction
- **generalised mapping engine control:** The initialisation of this controller is provided by the microprocessor and it drives the MatRISC control instruction.

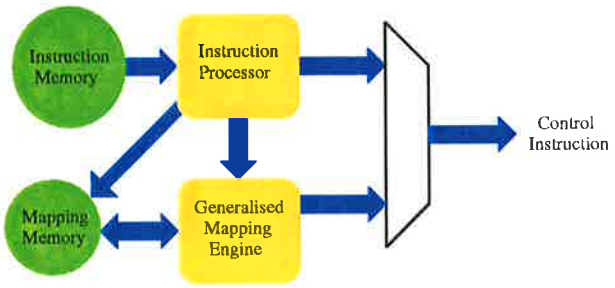


Figure 2.3 MatRISC control scheme.

The program and GME control can be partitioned at any level and boundary in the array, however the control and data must be synchronised across the partitions to maintain the integrity of the machine. The partitioning is arbitrary - a partition can be defined by a *processing element* (PE) in the array which is a grouping of a memory node, data processors and a switch and local control over these elements is within the partition. Figure 2.3 shows the major control blocks for a PE. The distributed control allows the machine to be more easily modularised, scaled and implemented. When the processor is in the GME mode of operation, it is rate- and processor-optimal for a particular set of processor design parameters and a cyclo-static schedule for functional units.

2 MatRISC Structural Decomposition

A two-dimensional MatRISC processor can be constructed as a scalable mesh-connected array of identical processing elements (PEs) as shown in Figure 2.4. Each PE contains memory, network switch, data processors and control as shown in Figure 2.5, a schematic of the high-level architecture.

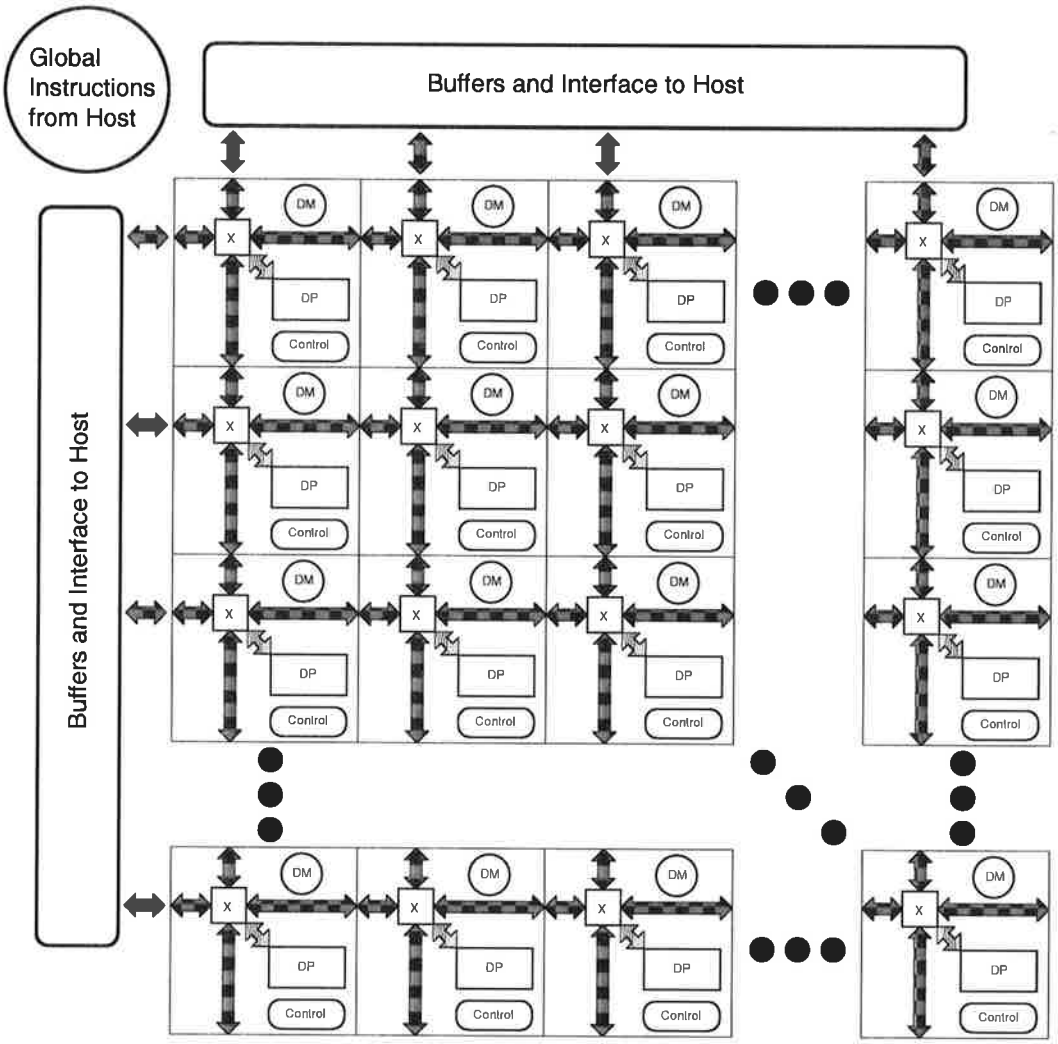


Figure 2.4 MatRISC processor array.

2.1 Processor array

The proposed implementation is a highly integrated fine-grained parallel processor. Each PE is directly connected to its four neighbours (denoted as north, south, east and west) by four I/O buses through which operands can be routed by a network switch at each node. The mesh bus structure ensures that operands can be routed through any distance on the array as a single hop (the diameter of the mesh is one) since all transactions for matrix kernel operations can pass directly between PE nodes with no extra hops between intermediate nodes. Traffic patterns in the MDMA which produce bus hazards can be overcome by time sharing of the buses. Mesh structures have been used widely for interconnecting processing nodes in systolic arrays and parallel processing systems and are a preferred structure for matrix mul-

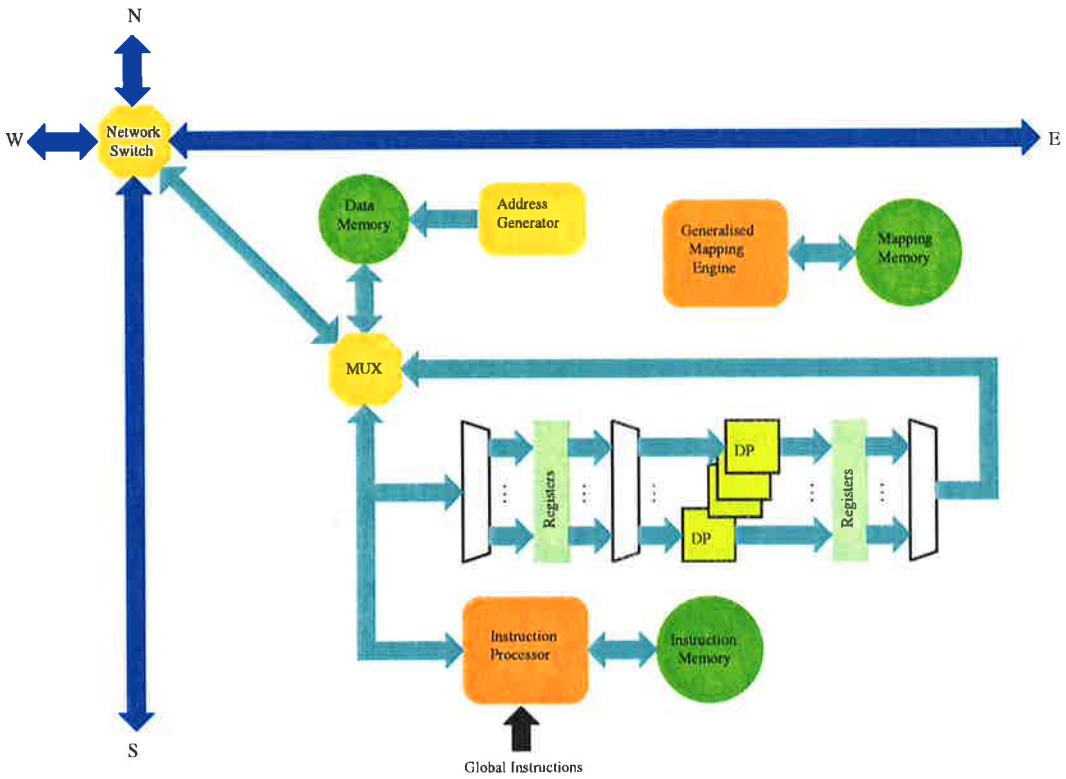


Figure 2.5 A processing element.

tiplication algorithms due to the simple communication paths and algorithm mapping. The interconnections are localised, the structures are readily implemented on planar substrates and are easily expandable to incorporate more processing nodes. A host processor accesses the array through interfaces on two edges of the array and a global instruction is sent to the array to start and stop execution of the local program and for access to program and data memories in each PE.

2.2 Processing element architecture

The PE features (Figure 2.5) a multiplexed data-path with separate multi-port instruction memories (IM) and data memories (DM), a RISC-based instruction processor (IP) and generalised mapping engine (GME) for control, an address generator (AG), multiple data processors (DPs) and a network switch with four I/O buses. The PE is assumed to have a synchronous clock and is pipelined for performance where the minimum time unit is a *processor cycle*. A brief description of each component is given so a simple model of the machine can be developed. More detailed descriptions are given in Chapter 3.

Data processors and memory

The DPs are 64-bit IEEE standard floating point multipliers and adders which are compatible with a host processor and preserve arithmetic precision. The DM is a two-port read, one-port write synchronous memory with a cycle time equal to that of the processor. The DM addresses are produced by an AG based on the alternate integer representation [Mar94] to access multi-dimensional data structures from a linear memory space through number theory mappings. These include multi-dimensional, sub- and transposed-matrices, common factor, skewed, circulant and constant mappings.

Instruction processor and memory

The IP is a small RISC microprocessor which can issue control instructions to the data-path, program the GME and perform housekeeping tasks.

Generalised mapping engine

The GME is a programmable hardware mapping engine which can issue control instructions to the data-path at the rate of one per processor cycle.

Network switch

The network switch configures the four I/O buses and can connect them in any grouping and direction or as a point-to-point connection (systolic operation) within each cycle.

3 System Partitioning and Implementation

A MatRISC processor may be partitioned depending on the total transistor count, I/O pin count, memory type, packaging options and the system it is to be integrated with. It may be used in any of the following configurations:

- integrated on-chip as the highest level cache memory with a host processor core
- real-time system - a stand alone system providing computation support for a task
- coprocessor to a host microprocessor with a shared memory. Access is through a high speed system or memory bus
- multiple MatRISC systems with hosts on a high speed network to provide massive parallel processing support.

Three configurations are discussed below.

3.1 Host processor integrated with a cache-MatRISC processor

For good performance with low power for moderate size problems, the processor array could be integrated onto a single chip with a large SRAM memory and a host processor core. The largest on-chip (second or third level) data cache can be partitioned into a MDMA with a PE placed at each partition and interconnected by the crossbar switch mesh. The partitioned cache still appears as a banked cache memory to the host processor core, however it is now capable of storing data structures which are accessed by the address generators. This integrated MatRISC processor has the benefit of being 'inside' the host processor's memory and has high bandwidth direct memory access through an on-chip memory controller. The host processor core accesses the MDMA/cache through requests to a memory controller which may interrupt the operation of the MatRISC processor array. The MDMA data buses are shared and also transport data to and from the host processor core. This approach is akin to 'spreading the processors among the memory' and is possible for algorithms with low data-dependence on control flow and where the control can be distributed. Figure 2.6 shows the high-level architecture of an integrated host/processor array chip. It is estimated that a single PE core, excluding the data memory occupies around $2mm^2$ of silicon area in a $0.1\mu m$ CMOS process. A 4×4 MatRISC processor array would occupy approximately 8% of the die area in a $400mm^2$ reticle. A 16MB second-level cache could be split into 1MB banks (per PE).

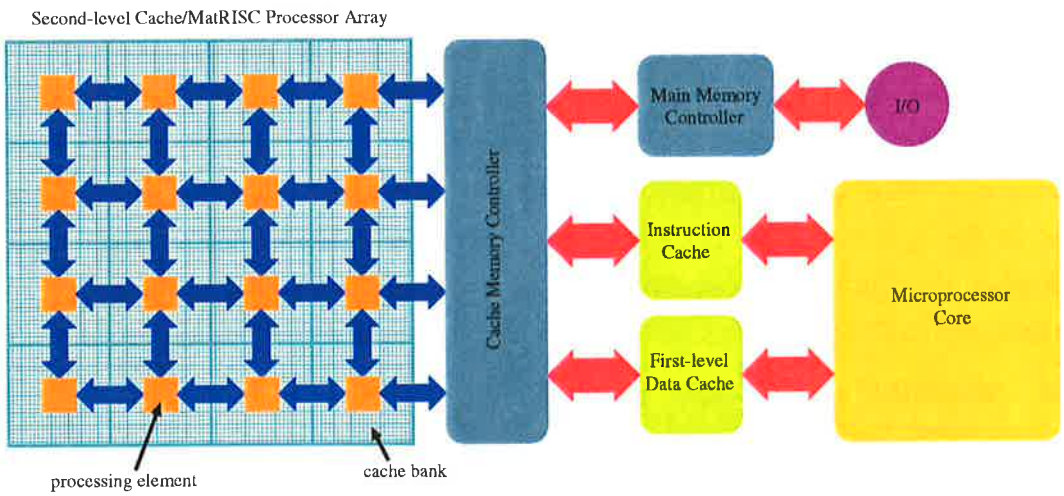


Figure 2.6 Integrated host microprocessor core/processor array solution for a HPTC processor module.

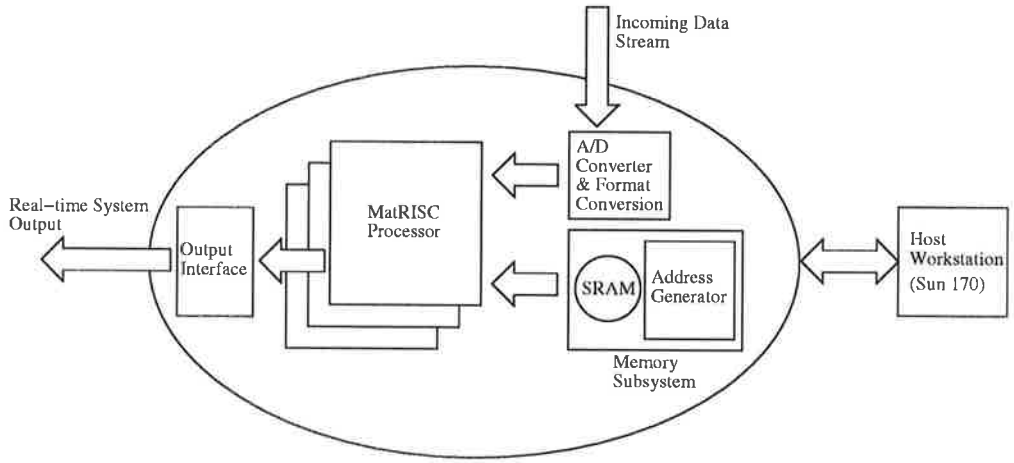


Figure 2.7 Schematic of a real-time system integrating at least one MatRISC processor.

3.2 Real-time system

Real-time systems implement an algorithm to map an input function to a desired output function, subject to deadline constraints imposed by the application. These systems are used extensively in real world applications such as control of vehicles, aircraft, spacecraft, robotics applications and image processing for radar. In many high performance systems, the complexity of the control and signal processing algorithms coupled with the stringent deadline constraint requirements on output rate has rendered current systems inadequate.

A real-time system based on the MatRISC processor array is shown in Figure 2.7, constructed from one or more MatRISC processor arrays and devices such as a high speed A/D converter and/or a memory subsystem through high bandwidth links. A host processor is responsible for non-real-time tasks such as management of the system, non-real-time plant monitoring and application development. Due to data storage limitations within the MDMA, some applications require a fast external memory subsystem which incorporates an SRAM memory and a programmable address generator. This subsystem is used for storage of coefficient matrices or other data structures. The memory subsystem can be interfaced in a similar way as the A/D converter. A medium sized extended Kalman filter for a guidance system involving 20×20 matrix computations, and a Volterra-based nonlinear compensation, are investigated in Chapter 6.

3.3 MatRISC coprocessor module

The MatRISC processor array could be integrated into an D-type (alumina substrate) multi-chip module (MCM) with the PEs and possibly memories fabricated as discrete chips. The PEs can be bump-bonded to the substrate or directly wire-bonded chip-to-chip since the processor bus connections are simple and regular. An MCM to support 16 processors and data

memory would have an area of around $100cm^2$, a breakdown of the area for each component in the PE is given in Chapter 3. This array processor organisation is the implementation focus for the prototype described in this thesis.

4 Architectural Features that Enhance Performance

Deeper pipelines and faster processor clock

The pipeline depth is not constrained by the existence of feedback paths in the data-path and the need to resolve data dependencies on control flow quickly. Deeper pipelines lead to faster clocking and higher throughput [PH96, DAC+92]. Performance may degrade for tasks involving small matrices whose size is around the same magnitude as the pipeline depth as data is not issued until the previous result has emerged from the pipeline.

Deeply pipelined and faster data memory

Traditionally, SRAM and DRAM memories are low latency, however the synchronous DM can be deeply pipelined which may allow its speed to be increased. A pipelined memory hierarchy can also be built to allow a larger DM to be employed. The controllers need to start fetching operands earlier to account for the additional skew which adds small extra startup and finishing delays to pipelined operations.

Low and constant network latency

A low and constant latency through the MDMA ensures pairs of operands arrive at the data processors at the same time. This reduces control complexity due to re-timing and reduces register storage overhead.

5 Architectural Performance Constraints

Limitations to performance and problem size are due to processor clock rate (also memory cycle time and scaling) and size of data memories.

Processor scaling

The MatRISC processor is intended to be integrated on a single chip or as a multi-chip module and can theoretically be scaled in performance by changing the array size. The limit of processor scaling is determined by the technology, packaging constraints and chip size. It is not scalable to arbitrarily large arrays because the speed of signal propagation along a bus will limit the cycle time. The worst case delay of a data signal is to pass along the euclidean

distance of the array, and this fixes the minimum clock period of the processor.

Asynchronous communications

Asynchronous techniques when used in a highly integrated processor module will demand more complex programming and hardware, however asynchronous techniques *must* be used if the array size exceeds the synchronous limit (the signal propagation delay across the array is longer than the processor cycle time). Larger hierarchies of MatRISC processors can be constructed where groups of locally-synchronous PEs communicate asynchronously with neighbours. This is illustrated in Figure 2.8.

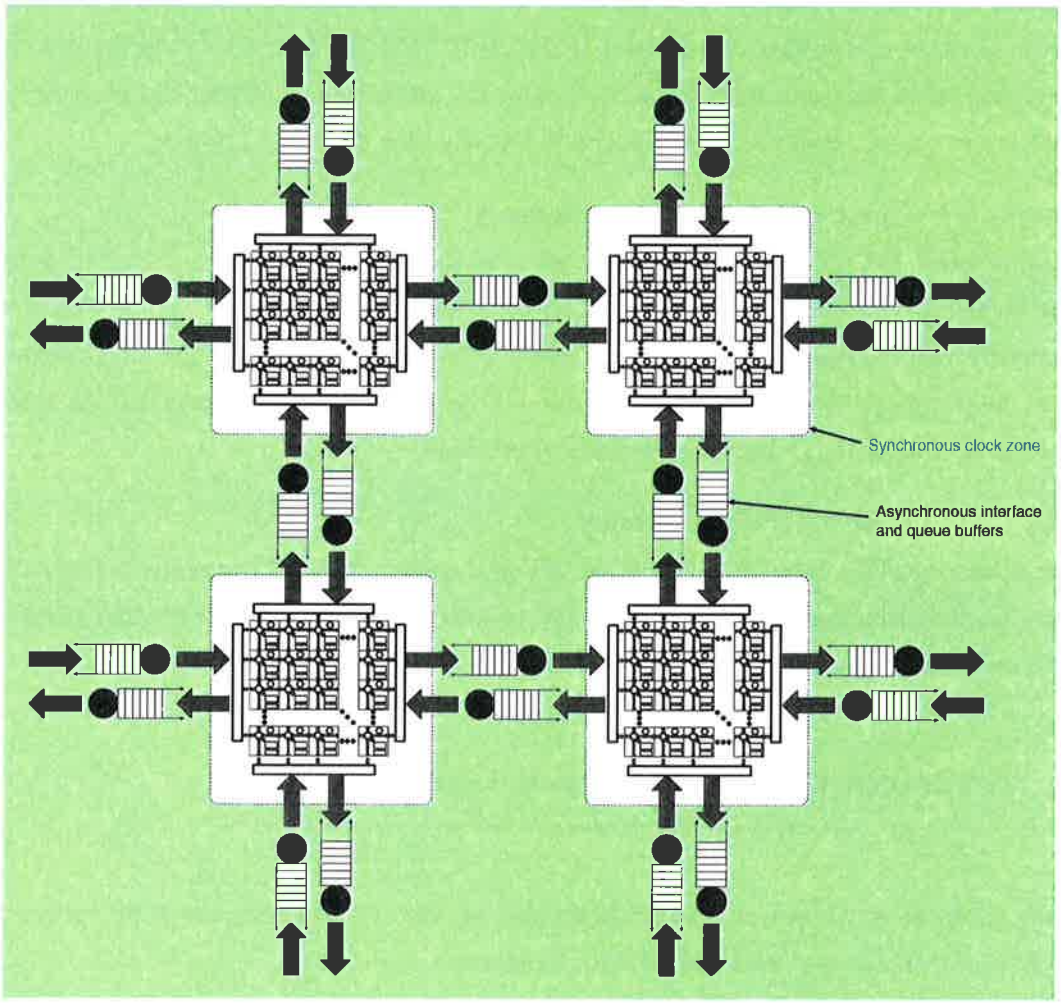


Figure 2.8 System of asynchronously connected MatRISC processors.

An asynchronous approach [App97] is desirable from the point of view of scaling and robustness from hazards. Buffers would also be needed to soak up any excess data arriving at a

PE. Good load balance can be achieved between processors because of the regular nature of matrix-based algorithms. The long data streams with processor-optimal computation rates means there is no slack time to process excess data and the buffer overflow is inevitable for large matrix sizes unless a control mechanism is used to stall faster processors. The data synchronisation can be achieved either by handshaking or encoding the data using a Manchester encoding scheme, for example. All of these schemes require extra hardware at the transmitter and receiver while utilising less of the available bandwidth. As the clock cycle times decrease, less time will be available to transmit the data through the network unless the propagation delay scales with the cycle time. This will be less likely as the parasitic capacitance does not scale well with VLSI process scaling in deep sub-micron technologies [WE95]. Other emerging technologies such as optical waveguides integrated on CMOS substrates provide a faster and higher bandwidth transmission medium. This technology is best suited to linking synchronous processors together with an asynchronous bus fabric. It is likely the area of the "synchronous zone" in which a MatRISC processor can be built will reduce in size with VLSI process scaling. Another solution is to use short hops in the network (mesh diameter two or three) but this makes control more complex and increases the execution time.

6 Scheduling, MIMD and S-SSIMD Operation

A compiler statically schedules the pipelines in each PE to avoid contention in buses, data memories and data processors for kernel operations. Code for kernel operations is highly optimised and built into the compiler as a template. Different templates are needed for different memory mappings and types of data structures. Data dependencies in applications and driver algorithms are handled using message-passing protocols which occur relatively infrequently compared with data-independent operations. Although the MDMA is able to operate as a multiple-instruction multiple-data (MIMD) processor, it achieves high performance for matrix kernel operations by operating in skewed-synchronous single-instruction multiple-data (S-SSIMD) mode.

A fully specified flow graph (FSFG) defines the data flow between indivisible nodal operations and data dependencies [Mad95]. For cyclic graphs, bounded parameters of the FSFG can be found:

- *iteration period bound* (IPB): the minimum achievable latency between iterations of a FSFG (called 'rate optimal')
- *processor bound* (PB): minimum number of PEs required to implement the FSFG at a

given IPB (called 'processor optimal')

- *periodic delay bound* (PDB): maximum periodic throughput delay under the fastest possible rate (called 'delay optimal').

There are two scheduling schemes – static and non-static. The machine class and scheduling technique is given below:

- **Static Scheduling**
  1. critical path method: acyclic graphs
  2. systolic: cuts and re-timing, regular
  3. SIMD: broadcast instructions
  4. MPP: low complexity.
- **Dynamic Scheduling**
  1. SSIMD: skewed instruction scheme, one PE performs a tile iteration
  2. PSSIMD: group of PEs perform a tile iteration
  3. cyclo-static: extra degree of freedom, tiles can be shifted in the processor space.

7 Where Does Data Come From?

The origin of the data for processing is a pivotal issue in simulating and benchmarking a processor. Operands stored in the data memory may have come from another memory system, a peripheral such as a hard disk or network interface or the result of a previous computation. It is beyond the scope of this thesis to characterise performance based on data originating from all possible sources, although a real-time system interface is discussed in Chapter 6, Section 1. It is assumed that operands originate in the data memories of the MatRISC processor as the result of a previous computation. This is a reasonable assumption if the data set for a problem fits inside the MDMA and the low level BLAS are called sequentially from a driver or computational routine.

7.1 Load versus compute time

In some cases the time taken to load operands into the memory of the MatRISC processor may be longer than the time to compute the result. If the only calculation to be performed is a single matrix multiplication on a large set of data which is stored outside the array,

the computation can proceed as the data is loaded. For the algorithms under investigation, most have multiple kernel operations chained together and external memory use restricts bandwidth resulting in lower performance.

8 Algorithm Mapping and Decomposition

The applications considered are primarily signal processing and numerical analysis problems which are rich in matrix- and/or vector-based computations. Algorithms can be divided into three general categories: dense matrix/vector, sparse matrix/vector and systolic arrays. Dense matrix/vector problems are algorithms that are the most easily parallelised with low control overhead. Regular systolic arrays can be implemented in a straightforward manner but algorithms with irregular data storage and access such as sparse matrix/vector problems may require a higher control and communication overhead, resulting in lower efficiency and are not considered further. Block level algorithms can be used to batch-process data that will not fit into the MatRISC processor data memory. Applications to be executed on the processor array can be decomposed as shown in Figure 2.9.

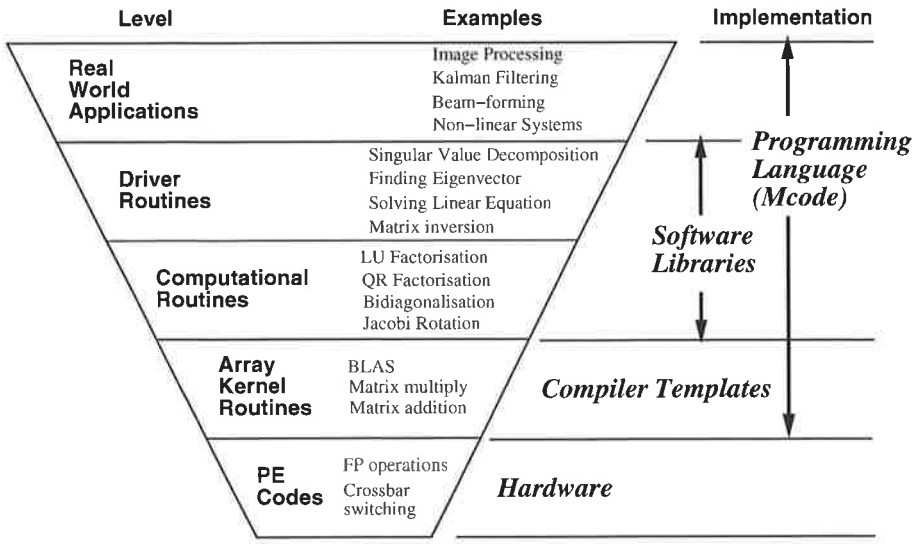


Figure 2.9 The decomposition of algorithms based on their level of complexity and the method of implementation in the MatRISC system.

The analysis presented in Chapter 1 determined the occurrence of each operation for a set of benchmark algorithms and the percentage it contributes to the execution time. This is now used to determine which operations should be mapped into hardware. The most commonly

used operators are implemented in hardware and these include floating-point addition (68%), floating-point multiplication (27%) and integer and logical operations. It is assumed less-used math functions are supported in software as a maths library [AS72] written in retargetable assembly. These include reciprocals and division, square root, complex number operations, power functions and transcendental functions. Other math functions can be implemented via a general purpose table lookup and/or algorithmically generated.

8.1 Kernel routines: matrix operators

A set of matrix–based algorithms can be decomposed into data-independent operations, in particular, the BLAS on matrices of medium size ($N <\approx 1000$). As discussed in Chapter 1, these operations can be classified into several levels according to their complexity:

- **LEVEL 1:** Routines of this level have a linear complexity in both their data and arithmetic relationship in the direction of operation, eg. dot product, matrix addition.
- **LEVEL 2:** These have a quadratic complexity in both data and arithmetic relationship which are used for matrix-vector operations, eg. outer product.
- **LEVEL 3:** These have quadratic data complexity and cubic arithmetic complexity which are targeted at matrix-matrix operations, eg. matrix multiplication.

To minimise the execution time for an algorithm, the execution time for dependent matrix multiplication needs to be minimised. This leaves levels 1 and 2 as memory bound operations.

8.2 Definitions

The following parameters are used to define the abstract model of the machine to be studied:

- N_1, N_2, N_3 - matrix sizes for non-square matrices
- p - physical array size (order)
- V - virtual factor
- s - DM size in operands (or M_b if specified) in each PE
- S - register size ($S = sp$) for both x and y registers
- B_l - load-store bandwidth in operands per second
- B_{dc} - data controller bandwidth for reads in operands per second
- B_c - compute bandwidth of one array side in operands per second

- w - bus width (bits) for each crossbar bus in each PE
- w_o - number of bits in an operand
- T_a - cycle time for the array
- T_{op} - operand cycle time of the array
- nDP - number of data processors in each PE

9 Matrix Multiplication

Matrix multiplication is the fundamental operation performed by the MatRISC processor and is an order N^3 operation where N is the order of the matrices to be multiplied. Matrix multiplication has the longest execution time compared to addition, subtraction and scalar multiplication, which are $O(N^2)$ operations. It is also the most common. Therefore, it is critically important to the overall speed of the processor that matrix multiplication is highly optimised by minimising its execution time. Discussions on matrix multiplication techniques for parallel computers can be found in [Mod88, Qui87, ABB+, GL96, GSvdG95, LWMM96, vdGW97, CDW94]. The ratio of compute rate to I/O bandwidth should be maximised for a particular processor since the bus width and speed is limited, whereas the speed of internal data processors can be made much faster. There are two methods for calculating the matrix product: by inner product or outer product computation. These are the end cases and other methods are hybrid combinations generated by exploiting the commutativity property in the algorithm.

Table 2.1 Properties of inner and outer product computations.

Method	Computations per $N \times N$ matrix	Number of input operands	Compute to I/O ratio	Number of output operands
inner product	$N(N-1)$	$2N$	$(N-1)/2$	1
outer product	N^3	$2N$	$N^2/2$	N^2

9.1 Matrix multiplication methods

The general form of the matrix multiplication algorithm is presented for a general single processor system. The forms of the algorithm are discussed for parallel machines and an

implementation for the MatRISC processor is reported. To perform matrix multiplication of two matrices X and Y of size $N_1 \times N_2$ and $N_2 \times N_3$ respectively on a scalar processor, three nested loops are required to compute the $N_1 \times N_3$ result, Z. The ordering of the loops can be arbitrarily chosen but the choice affects the speed of the computation, the data memory access patterns, the hardware utilisation and in what ordering the result is produced. One of the simplest methods is to order the loops in an inner-product ordering written below in pseudo-code:

```
for k in 1 to N3                                          -- count across Y
  for j in 1 to N1                                  -- for each Y, count down X
    for i in 1 to N2                  -- perform an inner product at each j,k
      read X(j,i)
      read Y(i,k)
      Z(j,k) = Z(j,k) + X(j,i) * Y(i,k)                -- multiply accumulate
    end loop
    write Z(j,k)
  end loop
end loop
```

The result, Z is computed one complete element at a time in column order. The outer two loops are swapped to compute the result in row order. This has a low hardware usage because only one temporary variable is required for Z and no other intermediate results need to be stored in registers. Two new operands are read from the data memory every cycle. The memory address patterns could be optimised in a conventional microprocessor to maximise data cache hits.

Modern computers are able to compute a multiply-accumulate much faster than it is possible to access main memory without some sort of memory cache scheme. If a machine has adequate cache capability for a desired range of problem sizes, the loops could be re-ordered to compute the first element of the inner product for a set of elements in Z. The pseudo-code below computes the first element of the inner product for each column vector of Z in sequence, then adds the second inner product element to the first and so on.

```
for k in 1 to N3                                          -- count across Y
  for j in 1 to N2                                  -- for each Y, count across X
    read Y(j,k)
    for i in 1 to N1                        -- do one element of the inner product
                                                  -- for each result element
      read X(i,j)
      read Z(i,k)
      Z(i,k) = Z(i,k) + X(i,j) * Y(j,k)                  -- multiply accumulate
      write Z(i,k)
    end loop
    write Z(i,k)
  end loop
end loop
```

Although this method has more reads and writes, operand Y is read only every N_1 cycles. $N_1 \times X$ reads are carried out every $N_1 N_2$ cycles. The other reads and writes can be directed to the cache which has faster access time than main memory. This means there are $N_1 + N_2$ main memory reads every $N_1 N_2$ cycles which is much less than $N_1 N_2$ reads for the inner-product method. This reveals two inter-related problems:

1. The matrix product algorithm implemented on the processor must incorporate the maximum re-use of operands while not increasing the local data cache requirements to unmanageable levels.

2. To maximise processor efficiency, all data processors and main memories must be kept busy as close to 100% of the time as possible over the whole range of matrix sizes (i.e. the processor must be I/O-compute balanced).

This same approach can be applied to the $p \times p$ processor array, the difference is that p^2 operations are running in parallel which means a significant speed advantage over a scalar processor. The MDMA architecture provides a natural and therefore efficient solution for the storage and operations on matrices. In the MDMA itself, matrices can be stored and accessed in many different ways. Several approaches to applying the MDMA architecture to the matrix multiplication problem will now be presented.

9.2 Inner products of partial inner products technique

This approach is to compute p elements of p result elements (result vector) in one cycle as shown in Figure 2.10. The outer loop across the X matrix and down the Y matrix partitions is an inner product using the processing array. The inner loop is for p cycles and computes p partial inner products of the result. This requires $p^2 + p$ operands to be accessed each cycle which are in-place multiplied together and p rows must be accumulated at the conclusion of the computation. The Y (or X) matrix must be stored in transposed form and a bus connects the rows together to accumulate the results while a vertical bus must be used to broadcast the Y operands. This has high bandwidth requirements since all memories must operate each cycle and only Y matrix data is shared among PEs. This method produces each result element very quickly but will not be considered further due to its high bandwidth to compute ratio.

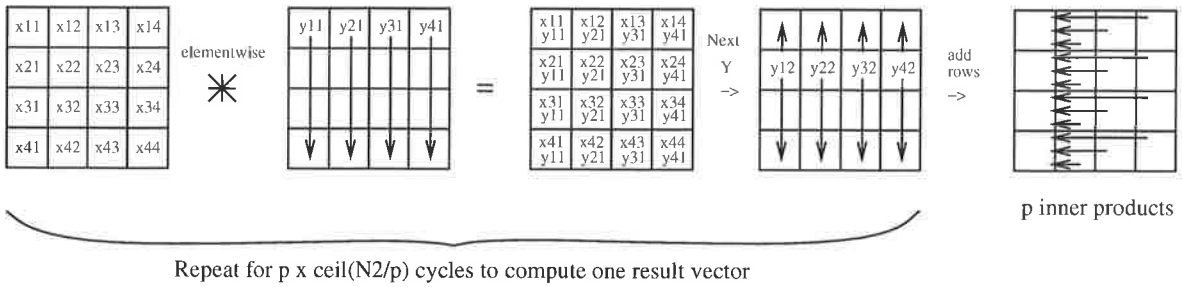


Figure 2.10 Processor array of order p^2 to compute p inner products for $p = 4$.

Other techniques similar to the above can be used but lead to declines in performance due to their higher operand bandwidth to compute ratios.

9.3 Direct mapping and 'inner products of outer products' technique

The inner step systolic array of Kung [Kun88] receives a series of time-skewed vector pairs on orthogonal boundaries and at its peak computes p^2 in-place inner products for each element of the result matrix. The key to the high compute rate to bandwidth ratio is the outer product which minimises the operand bandwidth to compute ratio in the array as shown in Figure 2.11. If buses are used to broadcast the operands to the PEs, the start-up and finishing times have been eliminated and the computation of a result partition is the in-place accumulation of a series of outer products. This uses less bandwidth than the previous approach and requires less time to compute the results because the inner product elements do not need to be moved and accumulated at the end of each resulting inner-product matrix. Figure 2.12 shows how the matrices are stored in the processor array and the memory sequence and broadcast

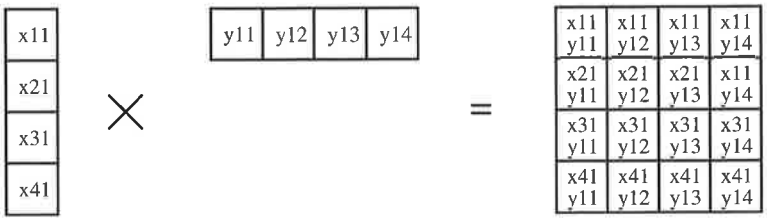


Figure 2.11 Outer product on a $p = 4$ processor array.

requirements for producing the first result matrix partition. This storage approach is called a direct MDMA mapping. There are 81 operand cycles needed to complete the matrix product.

Virtual factor

It is well known that minimising the execution time for a matrix multiplication routine on a microprocessor with hierarchical memory is carried out by varying the blocking factor which maximises cache hits and increases the re-use of operands [ABB+92]. The block size is the size of the result partition which is to be calculated. In the MatRISC processor, a fixed physical array of processing elements can be made to appear as a larger machine by making the block size an integral number of machine sizes p. This means the input operands can be reused or more data processors can be used which increases the performance of the processor. The multiplier for the physical array size is the **virtual factor**, V.

A virtual factor is applied to both the memory storage and the compute rate, so that matrices are stored using a block MDMA direct mapping as shown in Figure 2.13.

For a virtual factor of $V = 4$, the size 9×9 matrices do not completely fill one order $Vp = 16$ partition. For each memory cycle shown, there are V computations in each PE which means the V previous values need to be kept in a register file or FIFO and V^2 results need to be stored for every V^2 processor cycles. A scheme for scheduling of the multiplications within each PE is shown in Figure 2.14 for the case of $V = 4$.

A $\star$ indicates multiplication operations to be performed with 4 multiply-accumulates (MACs) per processor cycle. One extra cycle is needed at the end of each computation to complete. The example in Figure 2.13 highlights the poor compute rate for small matrices (order around a few Vp partitions) due to the under-utilisation of the DP and DM. This example takes the same time as an order 16 matrix multiplication.

Memory accesses are constrained in groups of V^2 per PE which makes pre-fetching on the diagonal PEs (which must supply twice as much data) more complicated since 16 extra

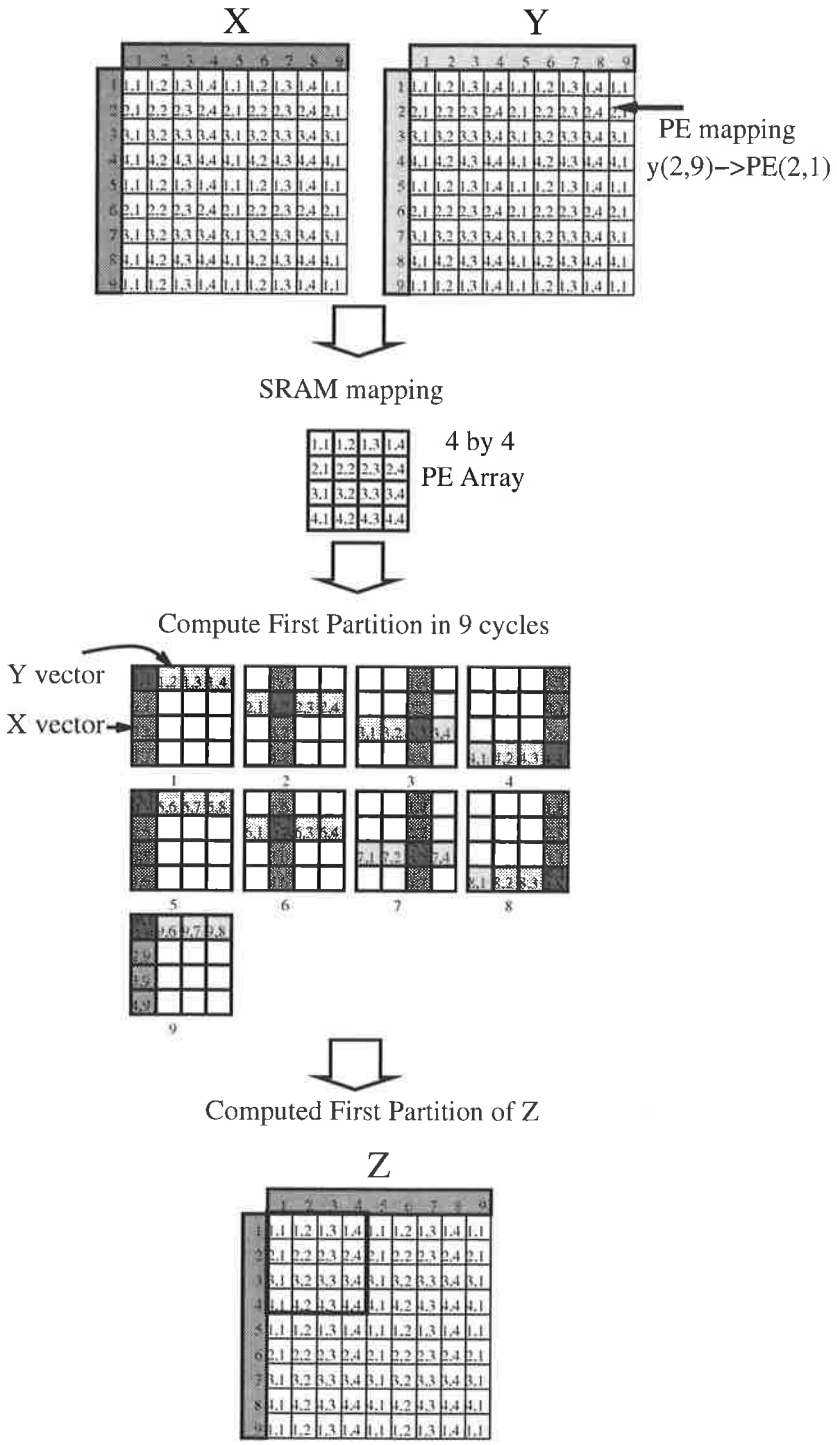


Figure 2.12 MDMA direct mapping of X and Y matrices and computation of the first result partition using a $p = 4$ processing array.

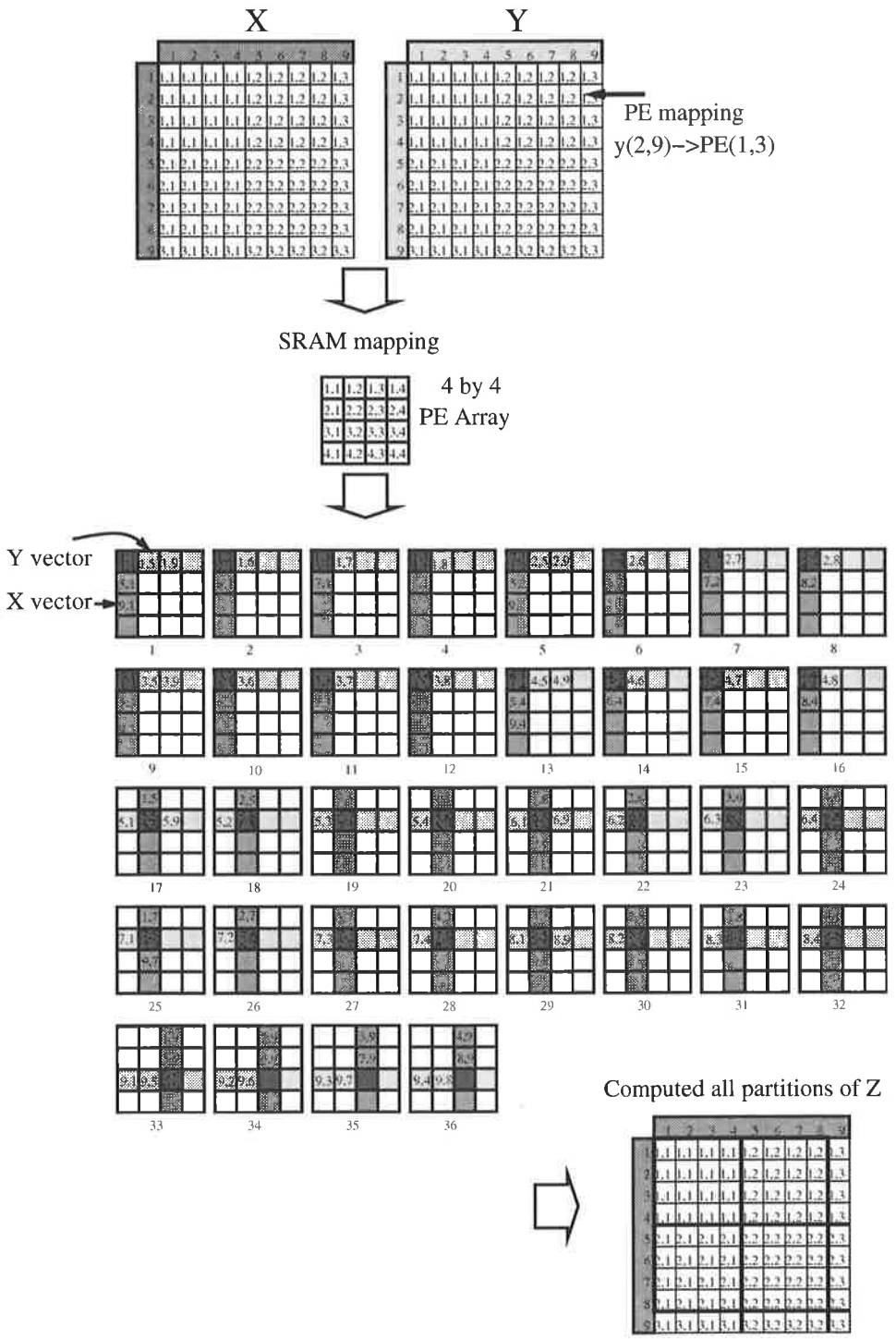
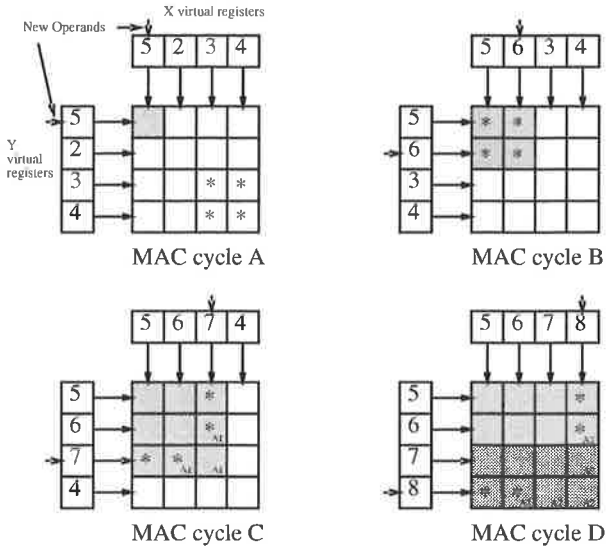


Figure 2.13 Block MDMA direct mapping with virtual factor computation of the result matrix Z on a $p = 4$, $V = 4$ processing array.



Start : cycle A.
Finish : after cycle A (1 extra).

Cycle A : choose 4 of left over A1 and A2. (clean up phase).
Cycle C : choose 2 of A1.
Cycle D : choose 2 of A2 and left over A1.

Figure 2.14 Scheduling data processors with $V = 4$ for cycles five to eight. The symbol $\star$ indicates possible multiplication of data.

operand registers will be needed. These registers are replicated in all PEs which increase the hardware cost. Another problem is the last calculation of an operand partition (N^2) requires operand access from $PE(1,1)$ since this should be pre-fetching V^2 operands for the start of the next partition so an extra V^2 cycle would be needed in this case. The writing of results to data memory occurs at the end of each result partition computation. Either V^2 extra operand cycles can be added or a second set of result registers are used to store the previous result partition until free memory cycles enable it to be written. This scheme will now be analysed further.

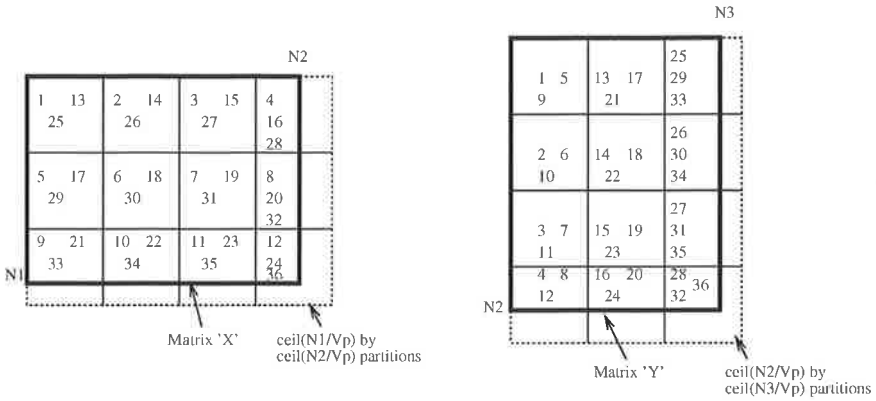
9.4 Matrix multiplication analysis

It is assumed that at every processor cycle, T_a, two complete vectors are fetched from the DMs and broadcast along orthogonal buses to all PEs and placed into the appropriate DPs. These operations can be pipelined but in this analysis startup and finishing times of the pipeline are ignored since they are overlapped during the computation and only add a few cycles to the end. It is also assumed that diagonal operands for one matrix are pre-fetched during the cycle before they are needed and that results can be written back when the memory is idle. For the multiplication of two matrices, $X(1 : N_1, 1 : N_2)$ and $Y(1 : N_2, 1 : N_3)$ to form a result matrix, $Z(1 : N_1, 1 : N_3)$, the pseudo-code executed in each IP is:

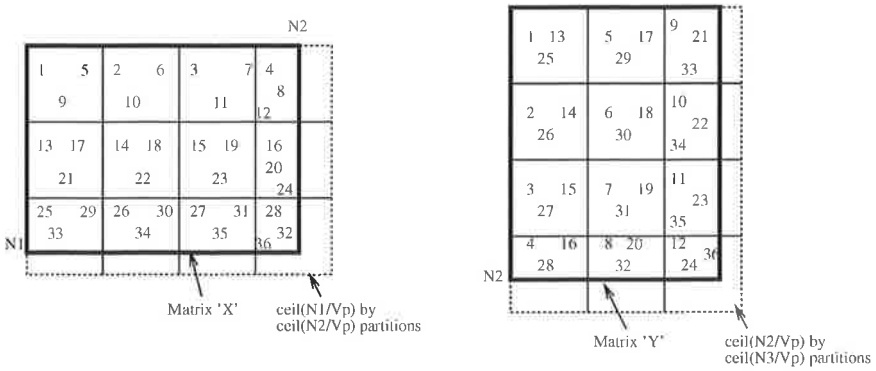
```
ceil_N1 = CEIL(N1/(V*p))          -- X matrix; the number of partitions down.
N2' = N2/(V*p)                     -- the number of inner product partitions.
ceil_N3 = CEIL(N3/(V*p))         -- Y matrix; the number of partitions across.
for k in 1 to ceil_N3                                  -- do partitions across Y
  for j in 1 to ceil_N1                                 -- do partitions down X.
    for i in 1 to N2'                        -- do partitions across X, down Y
      for t in 1 to V*p                  -- do outer products inside partition
        for h in 1 to V                          -- do virtual factor partition
          Z'(j,k,h) = Z'(j,k,h) + X'(i-1+t,j,h) * Y'(k,i-1+t,h)
        end loop
      end loop
    end loop
    write Z'
  end loop
end loop
```

where Z' is a $p \times Vp$ matrix. The complete matrix partition, $Z(j, k)$ is formed from four Z' matrices with the rows interleaved. $X'(i-1+t, j, h)$ is a length p vector, four of which form a vertical vector in partition j of matrix X. $Y'(k, i-1+t, h)$ is a length p vector, four of which form a horizontal vector in partition k of matrix Y. It may be useful to note that the loops for N_2 and N_3 can be swapped depending on whether the result matrix is formed in column or row order and the loops for i, t and h can be arbitrarily swapped and run in any order (eg. reversed, interleaved). The two outer loops (j and k) which direct the order of the formation of the inner products must perform on an integral number of partitions. The inner loop used to calculate the outer products (i) may have the ceiling function applied to it since

the formation of the outer products can be blocked when $i+t > N_2$. All outer products after this point become zero. Figure 2.15 shows two schemes for calculating the result matrix in either column or row order. The numbers in each partition indicate which cycle that partition is computed in the processor array. Note that the N_2 direction end partitions do not need to be fully stored in the array, since the computation of the outer products can be stopped at N_2. Figure 2.16 shows the formation of the result matrix when a virtual factor is applied to the computation.



(a) Calculate result in column order



(b) Calculate result in row order

Figure 2.15 Two ways to sequence matrix partitions.

The operand cycle time of the processor array is the time to transfer a single operand across the bus. This is given by:

$$T_{op} = \frac{T_a w_o}{w} \tag{2.1}$$

which is $10ns$ in our example.

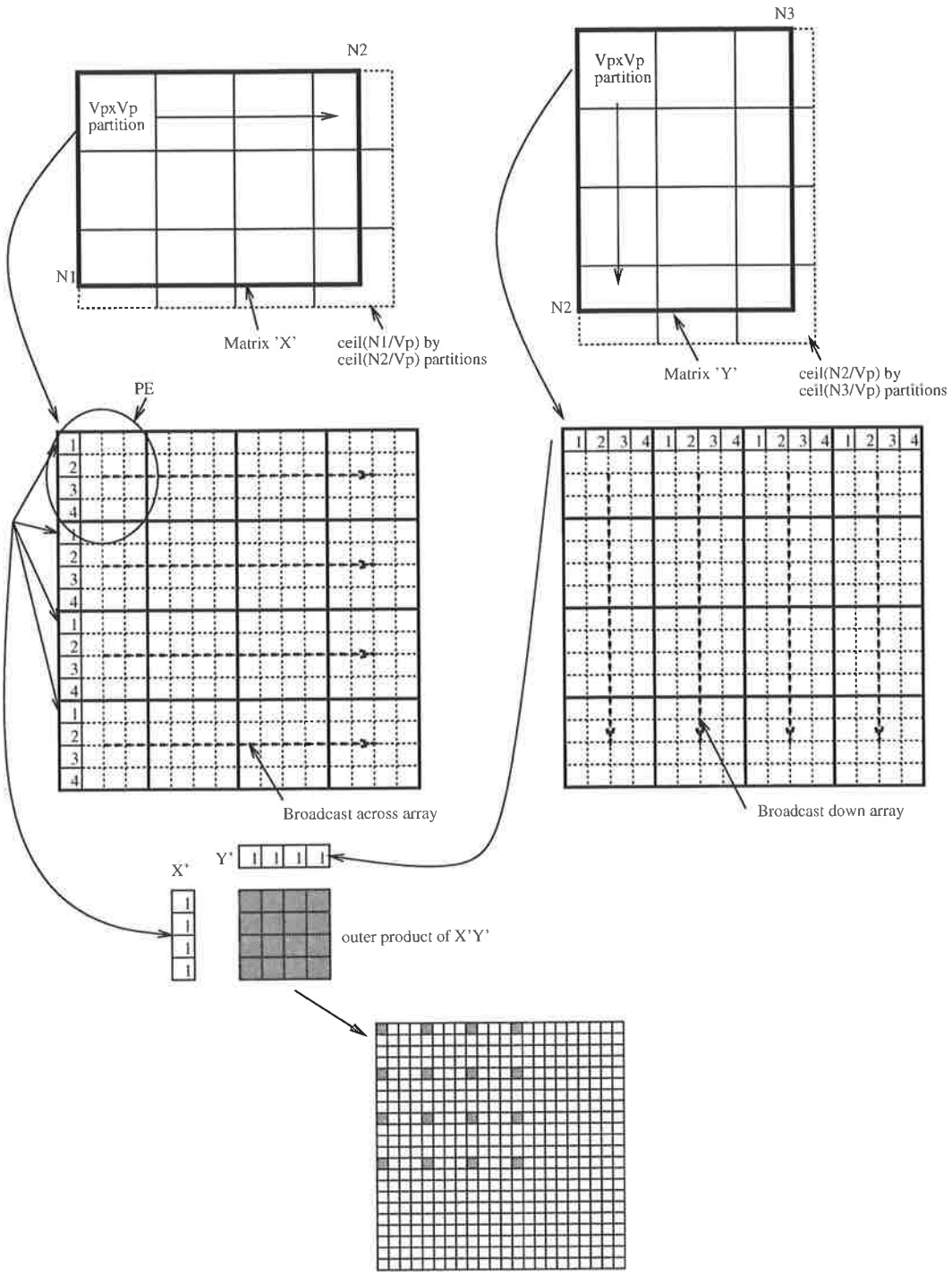


Figure 2.16 **Constructing outer products using virtual factors ($V = 4, p = 4$)**

The virtual factor and the number of DPs can be varied to change the I/O to compute balance of the PE. The virtual factor has the effect of localising $V \times V$ neighbouring operands in one PE. This means each time a memory in a PE is activated to supply operands onto a bus, it may do so for V operand cycles or the operands may be supplied out of order to facilitate memory interleaving (this is effectively changing the sequence of the t loop in the pseudo code). This means that for each operand cycle, V multiply-accumulates need to be carried out. If we assume the cycle time of the MAC is the same as the processor array cycle time, then the number of MAC DPs required (nDP) is given by:

$$nDP = \frac{VT_a}{T_{op}} \tag{2.2}$$

In the example so far, two MAC DPs are required for a virtual factor of $V = 4$. To use two pipelined MACs, a final add of operand sets in the pipelines may be needed to form the inner products. This will make the computation time longer and appears as a fixed overhead for each inner product that is calculated. This is mandatory for the case where $V = 1$ and there is more than a single pipelining stage in the MAC.

If it is assumed that the PE is bandwidth limited in terms of I/O and DP compute rate, i.e. equation 2.1 does not hold and there is an excess or balance of compute resources, then the total time to compute the matrix product $X \times Y$ is the product of the four loop lengths in the pseudo-code (t,i,j,k):

$$T_{XYBL} = \lceil \frac{N_1}{VP} \rceil N_2 \lceil \frac{N_3}{Vp} \rceil V T_{op} \tag{2.3}$$

This assumes that for each operand that arrives in a PE, there is enough compute resources remaining to calculate the virtual factor outer products. If there are less DPs than given by equation 2.2, then the system is compute bound and not memory bandwidth limited. The total time to compute the matrix product becomes:

$$T_{XY} = \lceil \frac{N_1}{VP} \rceil N_2 \lceil \frac{N_3}{Vp} \rceil \frac{V^2 T_a}{nDP} \tag{2.4}$$

This assumes that a whole partition must be computed at the end of the computation even though it is not full (matrix padded out with zeros), hence the ceiling functions in the equation.

9.4.1 Compute Rate

The compute rate for one matrix multiply is relatively constant for the array except when the edge partitions are computed. The average compute rate for a single matrix multiply can be calculated using equations 2.3 and 2.4. This is always less than the peak rate which is only

achieved when the matrix size is an integral number of Vp. The ratio of the average rate to the peak rate is the efficiency of the processing array. The number of arithmetic operations (two for a single multiply-accumulate) is given by:

$$ops = 2N_1N_2N_3 \tag{2.5}$$

The compute rate, R (FLOPS) for the bandwidth limited case is:

$$R_{BL} = \frac{ops}{T_{XY}} = \frac{2N_1N_3}{VT_{op}\lceil\frac{N_1}{Vp}\rceil\lceil\frac{N_3}{Vp}\rceil} \tag{2.6}$$

The compute rate for this case is inversely proportional to the array cycle time, T_{op}. For the case where the array is not bandwidth limited (equation 2.4), the compute rate becomes:

$$R_{BL} = \frac{ops}{T_{XY}} = \frac{2N_1N_3nDP}{VT_a\lceil\frac{N_1}{Vp}\rceil\lceil\frac{N_3}{Vp}\rceil} \tag{2.7}$$

This means that the theoretical compute rate in this case is directly proportional to the number of DPs and the number of PEs and inversely proportional to the array cycle time. In both cases, the compute rate drops rapidly when the matrix size fails to fit into integral partitions of Vp. This leads to the 'saw-tooth' effect seen in the next section which is a characteristic of these array processors.

9.5 Simulation

By applying equation 2.7 directly, the compute rate of equal-sized matrices is computed as a function of matrix order using Matlab [Mat99]. This is shown in Figure 2.17 for $V = 4$, $p = 4$, $nDP = 2$ and $T_{op} = 10ns$. The average compute rate for matrices of order 1-1000 is 12 $GFLOPS$ which is very close to the peak of 12.8 $GFLOPS$. The characteristic saw-tooth pattern occurs because as the matrix order increases it must periodically compute extra partitions which are not filled so the array is under-utilised for that partition.

Figure 2.18 shows the compute rate for four values of virtual factor, $V = 1, 2, 8$ and 16 for a fixed bandwidth ($T_{op} = 10ns$) and array size, $p = 4$. The number of DPs required is $nDP = 1, 2, 4$ and 8, respectively.

Figure 2.19 shows the peak and average matrix multiply compute rate as a function of virtual factor over all matrix orders from 1-1000 on a bandwidth limited, order $p = 4$ processor array. The number of DPs per PE required to prevent the array from being compute limited is also shown. The points marked with an asterisk indicate that the extra DP is not fully utilised and the points where $V = 2, 4, 8, 10, 14$ and 16 are balanced in terms of I/O and compute rate; these should be the first choice for a virtual factor. A side effect of increasing the virtual

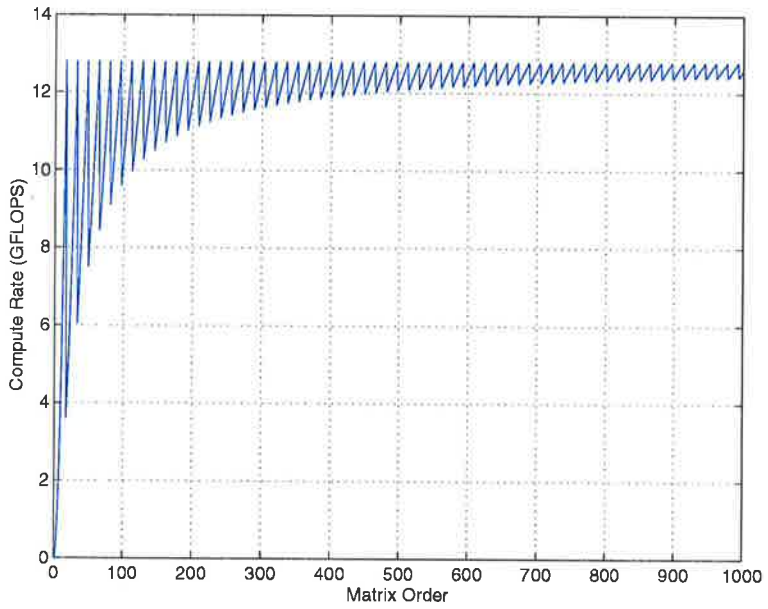


Figure 2.17 Compute rate vs matrix order for a processor array ($V = 4, p = 4, T_{op} = 10ns$).

factor is that the average compute rate falls away slightly. This is due to a larger proportion of matrices that are less than Vp in size indicating that the array is under-utilised for a slightly larger proportion of the time. Although the range of matrices has been studied up to order 1000, the minimum DM requirements per processor is given by the maximum number of operands to store three matrices (X,Y and Z) divided by the number of DMs:

$$s = V^2\lceil\frac{N_1}{Vp}\rceil\lceil\frac{N_2}{Vp}\rceil + V^2\lceil\frac{N_3}{Vp}\rceil\lceil\frac{N_2}{Vp}\rceil + V^2\lceil\frac{N_1}{Vp}\rceil\lceil\frac{N_3}{Vp}\rceil \tag{2.8}$$

The DMs on the east and south edges of the array store less operands than the other DMs. The ceiling function has to be applied to the N_2 parameter since this calculates the maximum storage required for the array. Table 2.2 shows the maximum matrix size that can be stored for $s = 1, 2, 4$ and $8Mb$ DMs (assuming $N_1 = N_2 = N_3$ for an array of order $p = 4$ and 8).

Table 2.2 Maximum matrix size for various DM and processor array sizes.

Array size	$s = 1Mb$	$s = 2Mb$	$s = 4Mb$	$s = 8Mb$
$p = 4$	288	408	577	816
$p = 8$	577	816	1154	1633

The simulation is restricted to an $s = 4Mb$ DM with an order $p = 4$ processor array and

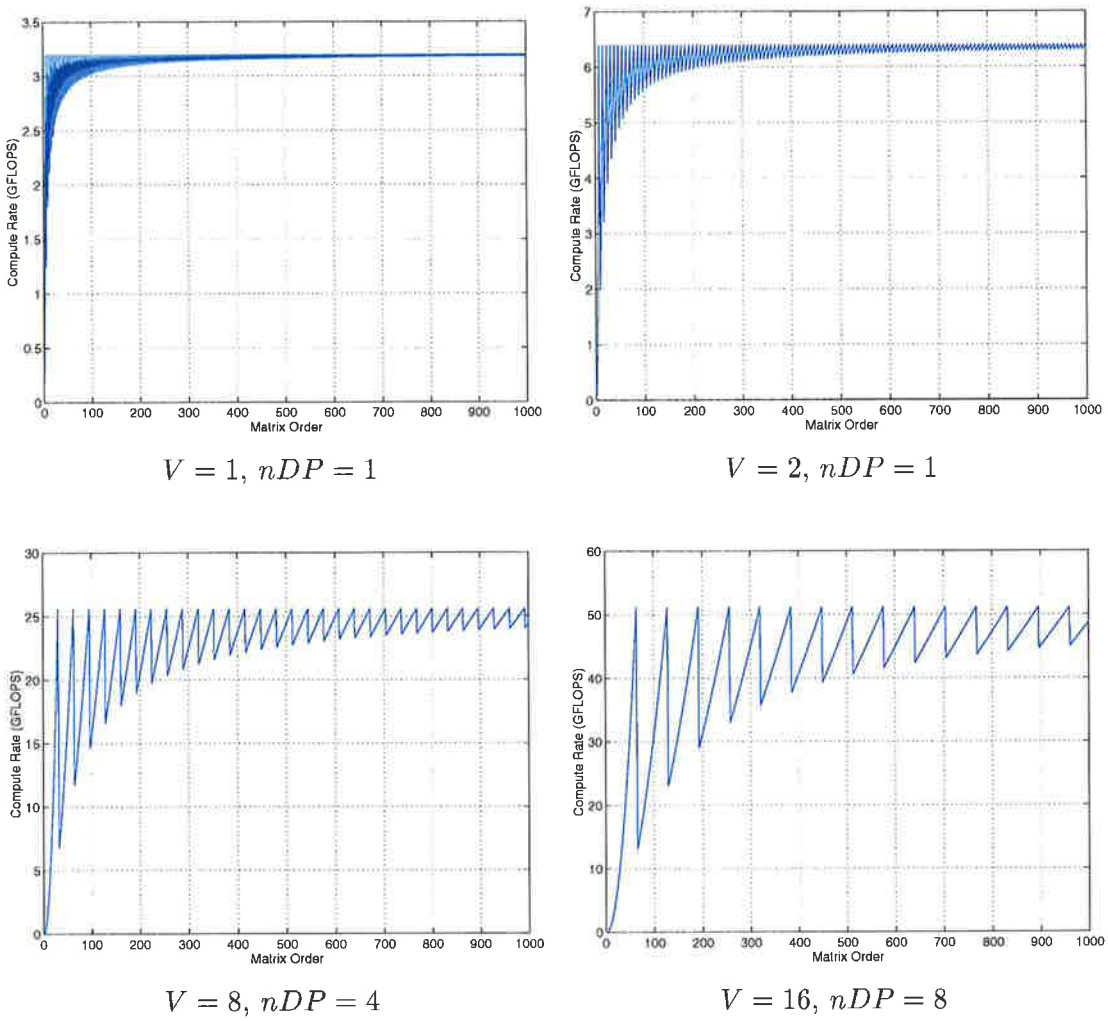


Figure 2.18 Matrix multiplication compute rates for various values of virtual factor, V and number of data processors, nDP.

the largest matrix that can be processed is approximately of order $N_1 = N_2 = N_3 = 400$. Figure 2.20 is a re-simulation of the average and peak matrix multiply compute rate versus virtual factor with compute rate averaged over matrices of order 1-400. The average compute rate has dropped by as much as 7 $GFLOPS$ (at $V = 16$) compared to when the DMs were larger in Figure 2.19.

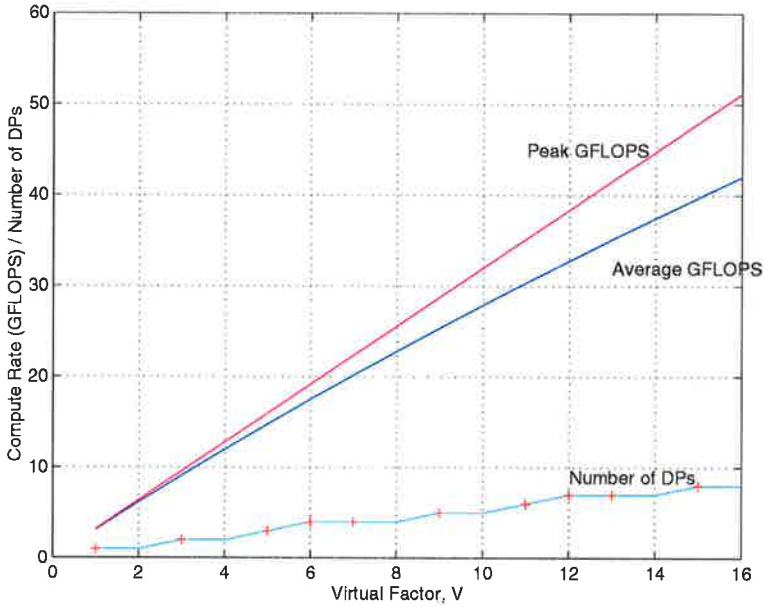


Figure 2.19 Average and peak matrix multiply compute rate for matrices from orders 1-1000 and number of data processors vs virtual factor.

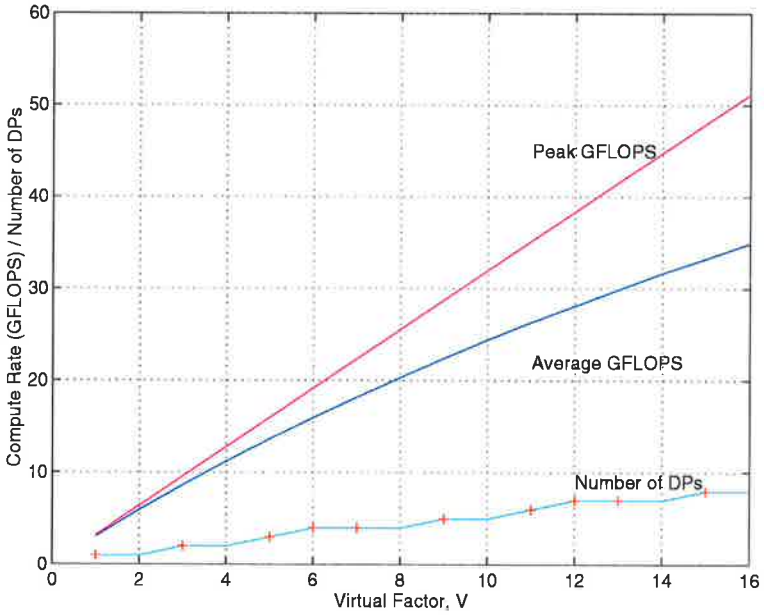


Figure 2.20 Average and peak matrix multiply compute rates for matrices from orders 1-400 and number of data processors vs virtual factor.

9.5.1 A direct mapped storage scheme with a virtual factor computation

The results of the previous section show that there is clearly a need to improve the average case performance, particularly for smaller order matrices. We can improve the average case by adopting a small virtual factor which ideally should be $V = 1$ so that the matrices are directly mapped into the MDMA. This however, severely reduces the peak performance of the processor array. The solution is to decouple the MDMA virtual factor from the compute balance virtual factor by recognising that the vectors used to form the outer products do not need to have contiguous elements. An example is shown in Figure 2.21 where the formation of a complete outer product over the $Vp \times Vp$ array occurs every three processor cycles. Note that the computation takes 27 operand cycles compared to 36 for the block direct mapped MDMA case.

The smallest MDMA virtual factor ($V = 1$) is used which means matrices are directly mapped and this has several significant effects:

- On average, operands are more evenly distributed across the MDMA which means on average more processors are able to be utilised, improving the average case performance
- Interleaving of memories is improved (only V accesses each time compared to V^2 for the block direct mapped MDMA) which means less temporary registers are needed for diagonal pre-fetches
- Simplifies addressing of operands from outside of the array
- Simplifies addressing of operands within the PE
- less wasted memory space due to zeros padding out operand memory locations in the edge partitions.

However, the reduced number of operand cycles can only be used when computing the last bottom-right corner partition of Z (bottom row of partitions of X and right column partitions of Y). This technique will now be more fully detailed and analysed.

9.6 Direct mapping and 'inner products of outer products' with a virtual factor

As with the block direct mapping case, it is assumed that at every processor cycle, two complete vectors are fetched from the DMs and broadcast along orthogonal buses to all PEs and placed into the appropriate DPs.

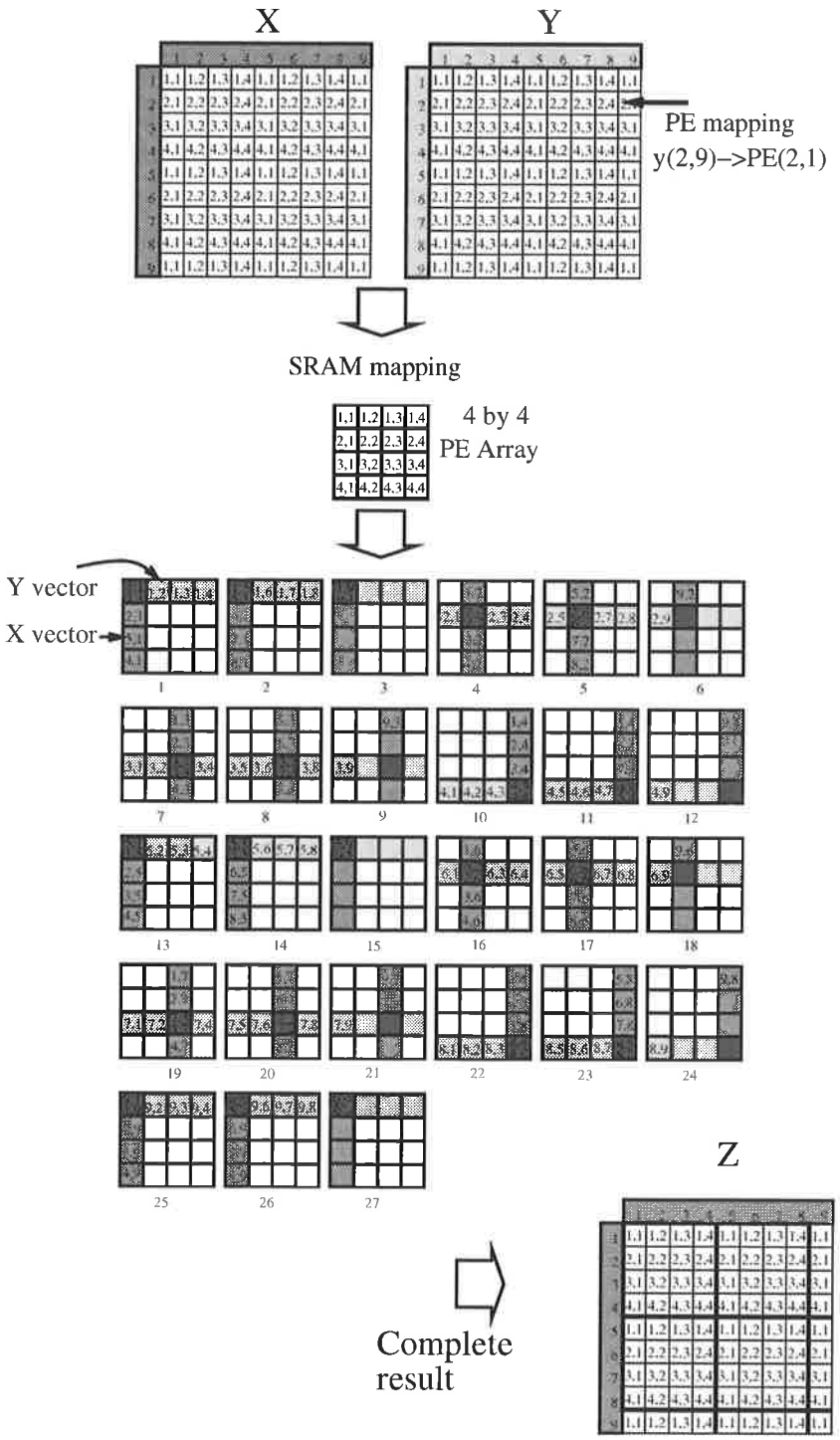


Figure 2.21 **MDMA direct mapping with virtual factor computation of the result Z on a processing array ($p = 4, V = 4$).**

The pseudo-code executed in each PE to compute $Z(1:N_1, 1:N_3) = X(1:N_1, 1:N_2) \times Y(1:N_2, 1:N_3)$ is:

```
ceil_N1 = CEIL(N1/(V*p))          -- X matrix; the number of partitions down.
ceil_N3 = CEIL(N3/(V*p))         -- Y matrix; the number of partitions across.
rem_N1 = CEIL(MOD(N1,(V*p))/p) -- number of virtual partitions left over (N1)
rem_N3 = CEIL(MOD(N3,(V*p))/p) -- number of virtual partitions left over (N3)
for k in 1 to ceil_N3                                  -- do partitions across Y.
  for j in 1 to ceil_N1                                  -- do partitions down X.
    for i in 1 to N2                                -- do vectors across X, down Y.
      if k = ceil_N3 and j = ceil_N1                -- computing last partition of Z
        for h in 1 to maxrem_N1,rem_N3              -- do partitions down X, across Y
          Z'(j,k,h) = Z'(j,k,h) + X'(i-1+t,j,h) * Y'(k,i-1+t,h)
        end loop
      else
        for h in 1 to V     -- do virtual factor partitions down X, across Y
          Z'(j,k,h) = Z'(j,k,h) + X'(i-1+t,j,h) * Y'(k,i-1+t,h)
        end loop
      endif
    end loop
    write Z'
  end loop
end loop
```

A simulation of a processor array for the case of $p = 4$ and $V = 4$ for matrices of orders 1-1000 showed an average improvement in compute rates of 2.5%. There is a noticeable improvement in the compute rate for small order matrices which doubled in some cases using the direct mapped MDMA scheme.

9.7 Summary of matrix multiplication algorithms

Several storage and execution techniques have been studied for performing matrix multiplication on a partition:

- Inner products of partial inner products
- Inner products of outer products
- Inner products of outer products using a virtual factor

- Block direct DM mapping of operands.

The virtual factor has the effect of increasing the compute rate of the array by using more (or faster) DPs and more on-chip registers which allows more re-use of operands. This can be implemented in two possible ways:

- squash a local sub-matrix of operands into a single DM. This has been shown to have poor performance for small matrices due to array under-utilisation and causes problems for matrix addition (see next section)

- stretch the virtual size of the array (Vp) over the direct mapped matrices which makes the data interleaved by a factor of p for V steps.

The second implementation method can proceed in several ways, either a single outer product at a time can be produced or a number of outer products can be produced in parallel. Both methods require the same amount of result register space but have different control mechanisms and input register space requirements. The trade-off is with memory access patterns in the MDMA. The computation of a complete $Vp \times Vp$ partition of the result matrix should be computed using the block inner product method (either column or row major) to minimise the write and read-back of partial results. Any other technique means the amount of data that needs to be cached is increased. If the matrix size falls outside of the capacity of the DM, a block matrix algorithm can be used.

10 Matrix Addition

Matrix addition is a bandwidth limited element-wise operation in the MatRISC processor. Every PE operates independently of all others as data is not shared because the matrices are mapped over one another in the same DM. This operation is the same for other element-wise operations (multiply, divide, square-root). The pseudo-code algorithm for matrix addition is:

```
ceil_N1 = CEIL(N1/(V*p))                    -- the number of partitions down.
ceil_N2 = CEIL(N2/(V*p))                    -- the number of partitions across.
for k in 1 to ceil_N2                        -- do partitions across.
  for j in 1 to ceil_N1                      -- do partitions down.
    for t in 1 to V                          -- do virtual factors across.
      for h in 1 to V                        -- do virtual factors down.
        Z'(j,k,t,h) = X'(j,k,t,h) + Y'(j,k,t,h)
        write Z'
      end loop
    end loop
  end loop
end loop
```

The operation to compute $Z'(j,k,t,h)$ can occur in any ordering of j,k,t and h since no data is re-used and the operations are fully commutative. Each addition requires the fetch of two operands and the write back of the result. This means that only one DP is needed and the array is bandwidth limited under the following condition:

$$T_{op} \geq \frac{T_a nDP}{3} \tag{2.9}$$

Equation 2.9 is always true if the PE is to efficiently perform matrix multiplication. The execution time to compute the addition $Z = X + Y$ is the product of the length of the four loops:

$$T_{X+YBL} = 3\lceil\frac{N_1}{Vp}\rceil\lceil\frac{N_2}{Vp}\rceil V^2 T_{op} \tag{2.10}$$

The time to perform the matrix addition is directly proportional to the operand cycle time but is affected by the virtual factor. If the matrix does not fit into an integral number of Vp partitions, the performance is reduced because the zeros around the edges of the matrices are included in the computation of the southern and eastern edge partitions. The number of floating-point additions is given by:

$$ops = N_1 N_2 \tag{2.11}$$

The compute rate, R can then be calculated:

$$R = \frac{ops}{T_{X+YBL}} = \frac{N_1 N_2}{3\lceil\frac{N_1}{Vp}\rceil\lceil\frac{N_2}{Vp}\rceil V^2 T_{op}} \tag{2.12}$$

For matrix orders that lie on integral numbers of Vp partitions, the compute rate is maximised and is directly proportional to the square of the order of the array and inversely proportional to the operand cycle time.

10.1 Simulation of the matrix addition algorithm

A Matlab simulation of a processor array of order $p = 4$, $V = 4$ with at least one DP for matrix orders from 1-1000 is shown in Figure 2.22. The characteristic saw-tooth patterns occur because whole partitions on the west and south boundaries of the array are not filled with operands but they still take the same amount of time to compute as a full partition. Figure 2.23 shows the peak and average matrix addition compute rates for an order $p = 4$

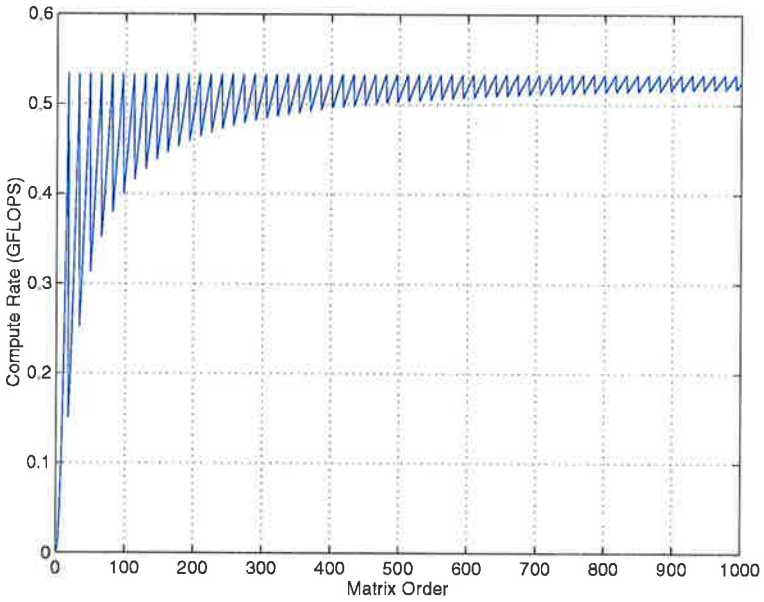


Figure 2.22 Compute rate for matrix addition on a processor array ($p = 4$, $V = 4$).

processor array as a function of virtual factor. The average compute rate drops as the virtual factor is increased because using larger virtual sub-matrices tends to force data to accumulate in the north-west corner of the array for matrix orders that are not on integral partitions of Vp. This makes the average case look progressively worse. The average case matrix addition performance is maximised if a virtual factor of $V = 1$ is used and this is clearly evident in Figure 2.24. This is possible in an array with a higher virtual factor since the way the matrices are loaded into the DMs would change. This helps to distribute more evenly the operands throughout the array instead of bunching them into a corner as shown in Figure 2.25 for the simple case of $N_1 = N_2 = 3$. However, a $V = 1$ storage method is not optimal for the compute rate for matrix multiplication which requires the highest virtual factor with an appropriate number of DPs to maximise performance. The stored matrix data cannot be easily re-ordered by changing the virtual factor so this method can only be used if no matrix multiplications are to be performed. The trade off between the virtual factor and the speed

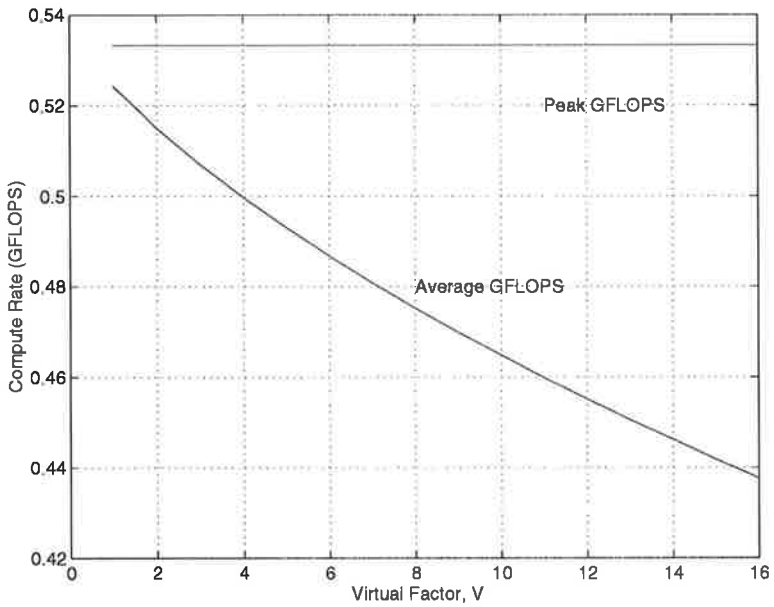


Figure 2.23 Peak and average matrix addition compute rates vs virtual factor for a processor array ($p = 4$) over a range of matrices of order 1-1000.

at which the algorithm computes depends on what proportions of different operations are carried out.

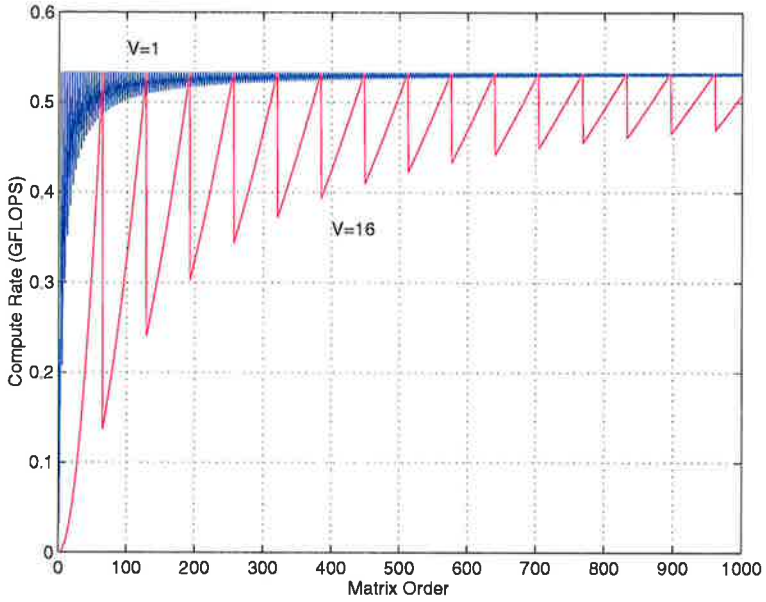


Figure 2.24 Matrix addition compute rate for a processor array ($p = 4$) with virtual factors of $V = 1$ and 16.

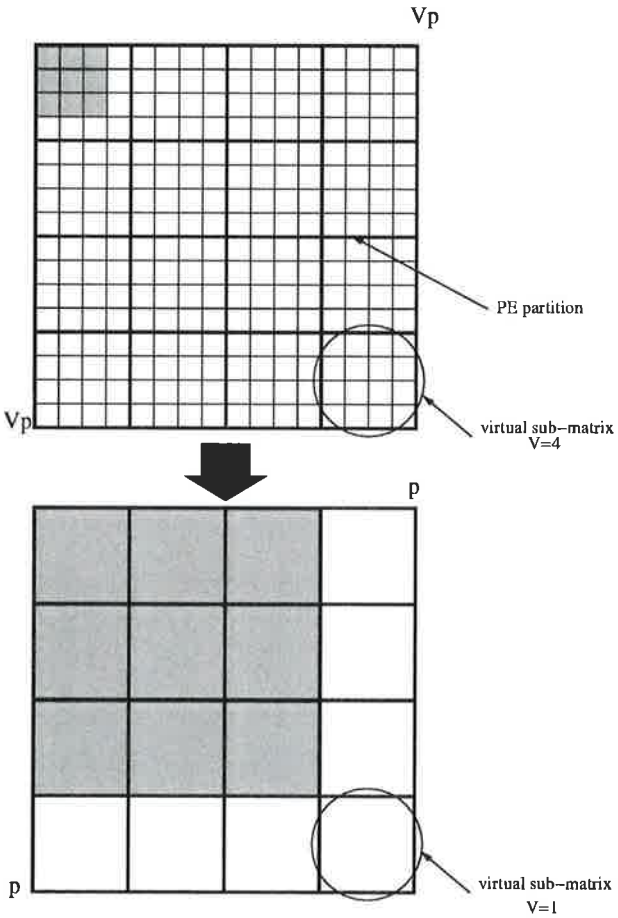


Figure 2.25 Transforming a matrix of order three on a processor array with $V = 4$ into an array with $V = 1$.

11 Strassen Matrix Multiplication

Strassen's technique for block matrix multiplication [GoVa89] was investigated for use on the processor array. It trades-off one block matrix multiply for 11 block additions which may be faster if the block matrix is large enough that it takes longer than 11 block additions to perform one matrix multiply. An iterative block-based technique was studied for a number of processor configurations where the fastest technique (outer product, block or Strassen's method) was found at each matrix size and used to compute the partitions of the next sized matrix (two times the order). It was found that Strassen's algorithm only became faster when the order of the processing array exceeded $p = 32$ (or 1024 nodes). This is due to the large speed-up of matrix multiplication using the outer product method, whereas matrix addition does not achieve the same level of speed-up. A plot of the difference in execution time between Strassen's method and the outer product method is shown in Figure 2.26. Strassen's method is faster where the plot is negative. The error propagation in Strassen's method is worse due to the increased number of arithmetic operations.

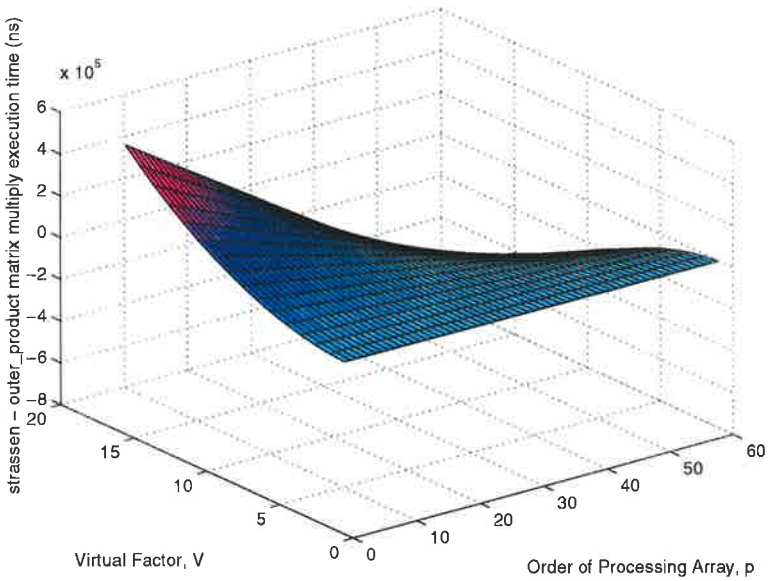


Figure 2.26 Difference in execution time between Strassen's method and the outer product matrix multiplication method for various processor array configurations.

12 Summary and Discussion

In this chapter, the matrix multiplication and addition kernel routines were derived and simulated. To perform matrix multiplication, PEs were connected together by horizontal and vertical buses to perform outer products, which is the most efficient matrix multiplication algorithm on this type of array because it maximises the compute rate to memory bandwidth ratio. The algorithm used was the "inner products of outer products" or strip mining. However, this has been modified by artificially enlarging the width of the "strip" by a virtual factor, V. This means more of the operands can be cached in each PE, leading to a faster execution time. Memory accesses can be interleaved along rows and down columns to avoid hazards. The memories were dual port (read mode only) to avoid operand pre-fetching and increased address generator complexity.

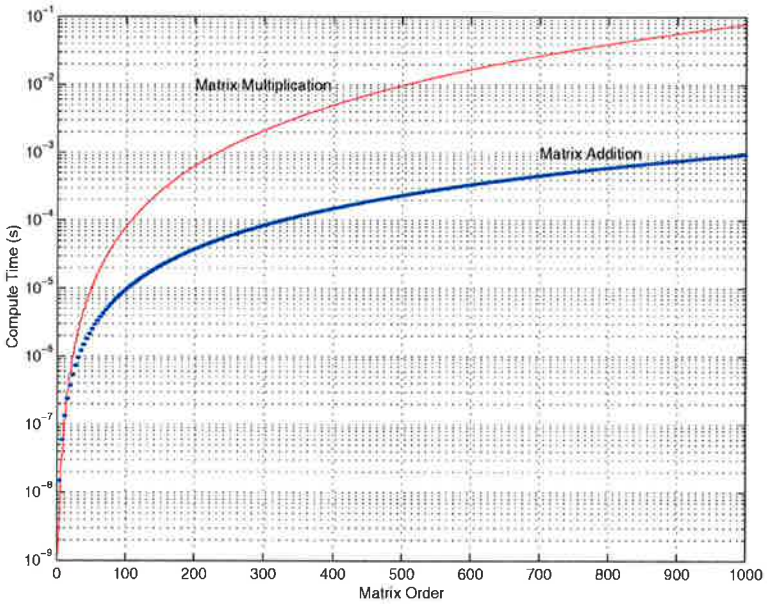


Figure 2.27 Execution time vs matrix size for matrix multiplication and addition.

The compute rate and execution time of matrix multiplication and addition when executed on a MatRISC processor (with parameters: $p = 4$, $V = 4$, $nDP = 4$ and $T_a = 10ns$) for equal size matrices is shown in Figures 2.27 and 2.28, respectively. The data rate, compute rate and memory design for the MatRISC processor have been optimised for a matrix multiplication

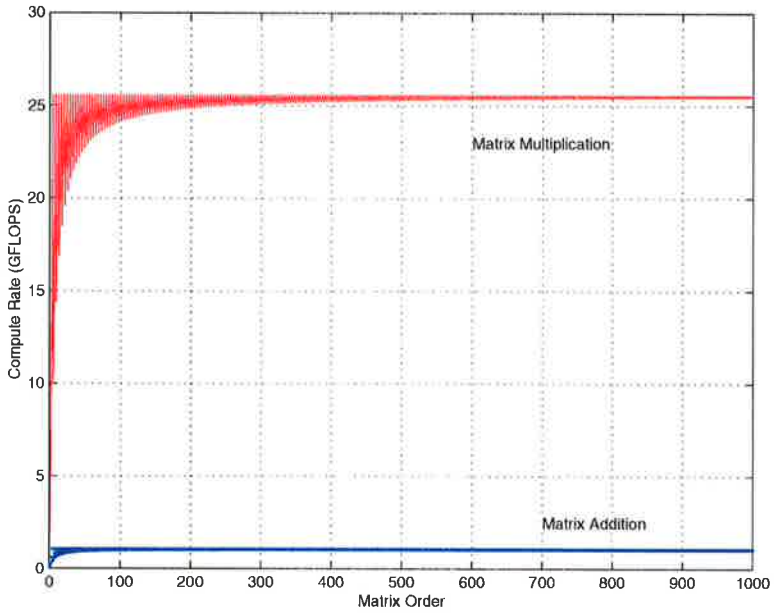


Figure 2.28 Compute rate vs matrix size for matrix multiplication and addition.

algorithm and the average multiplication compute rate for matrices from order 1-1000 is 25.16 $GFLOPS$ which is very close to the peak of 25.6 $GFLOPS$. The characteristic saw-tooth pattern occurs because as the matrix order increases it periodically must compute extra partitions which are not filled and the array is under-utilised for that partition. To achieve higher compute rates and faster execution times, the number of DPs and local register storage is increased. The execution time to compute a 500×500 matrix multiplication is 10ms for this example MatRISC processor. Matrix addition is a bandwidth limited operation on this MatRISC processor since there are three local memory accesses for each computation. The average addition compute rate for matrices from order 1-1000 is 1.048 $GFLOPS$ which is only 4% efficient but a much higher compute rate than a scalar microprocessor.

Chapter 3

MatRISC-1 Microarchitecture and Implementation

MICROARCHITECTURAL and implementation details of the MatRISC processor are presented in this chapter. The high-level architecture of a MatRISC processor was investigated in the previous chapter for a range of parameters including array size, virtual factor and number of data processors. The case of a 4×4 two-dimensional array and processing elements containing four multiply-accumulate data processors and having a virtual factor of four is designed and studied here. This particular implementation is called MatRISC-1. Each of the PE components are discussed in detail, a VHDL model of the PE and array is developed and the fabrication of parts of the PE in a CMOS process is presented.

A full-custom VLSI design methodology together with university [Adv95b, Adv95a] and commercial [Met96] CAD tools and a carefully crafted set of shell scripts was used to design and layout the circuits. Fabrication of the parts was done with Hewlett-Packard ($0.5\mu m$ and $0.35\mu m$ CMOS) and TSMC (Taiwan semiconductor manufacturing corp. - $0.25\mu m$ CMOS) and arranged through MOSIS, USA (*www.mosis.org*). References on the design of VLSI circuits can be found in [Bak90, GD85, WE95, Sho88].

1 MatRISC-1 Implementation

MatRISC-1 is a prototype implementation of the MatRISC processor architecture. The PEs are identical and integrated as a single CMOS chip with either on-chip or external DM in the form of a high speed SRAM. The array processor module is a two-dimensional $p = 4$

array of PEs integrated in a multi-chip module (MCM) shown in Figure 3.1 with various external memory chip options. The PEs may be bump-bonded to the substrate or directly

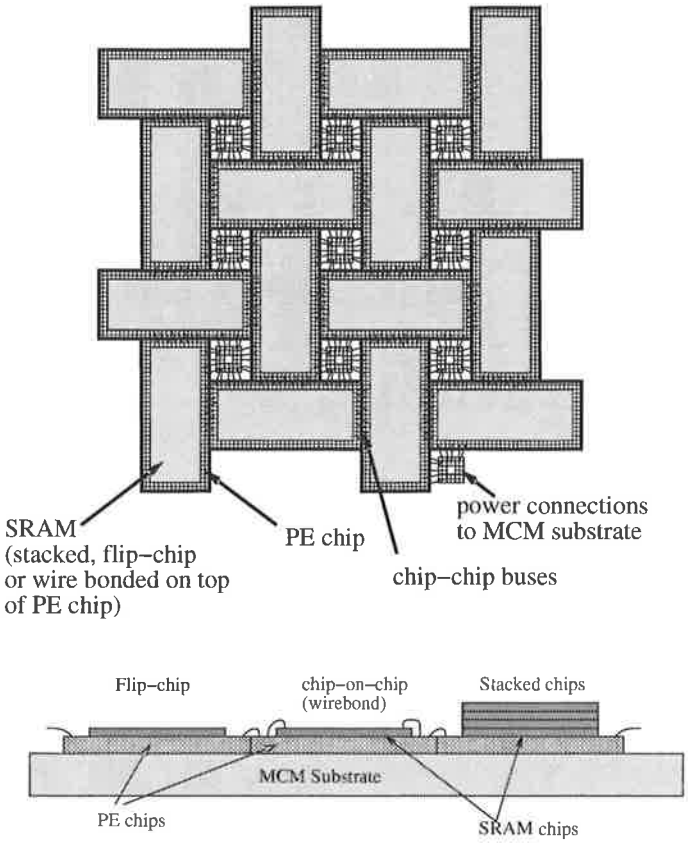


Figure 3.1 MCM solutions for a processor module.

wire-bonded chip-to-chip since the processor bus connections are simple and regular. The prototype MatRISC-1 processor MCM contains a 4×4 array of PEs and occupies an area of $100cm^2$. Each PE contains four floating-point (FP) adders and four FP multipliers. Each PE is assigned a unique identification number which indicates its position in the array and is used for sequencing memory accesses and driving the data-path and buses. The architecture of the MatRISC-1 PE is shown in Figure 3.2.

The area of each PE is approximately one million transistors according to a count of the transistor populations of the parts in Table 3.1, assuming implementation in a $0.25\mu m$ CMOS technology and excludes the data memory which is a separate SRAM chip. The silicon area occupied per PE is estimated to be $10mm^2$. Three test chips have been successfully fabricated and tested in $0.5\mu m$ and $0.35\mu m$ CMOS processes. These chips contained test structures for crossbar switching elements and buses, a dual-port SRAM, a $1GHz$ voltage controlled

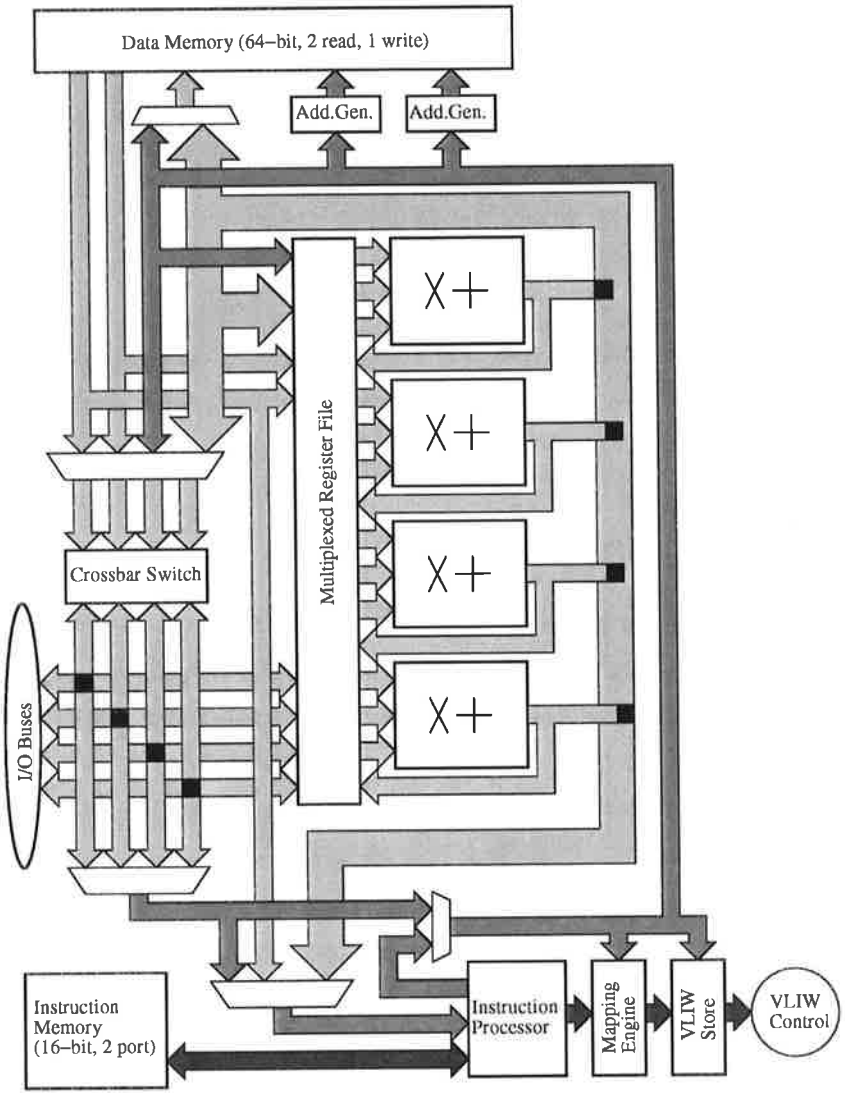


Figure 3.2 MatRISC processing element architecture.

Table 3.1 Component area estimates for a processing element.

Component	Area (mm^2)	Transistor count
FP adder	0.48	37k
FP multiplier	0.95	120k
Instruction Memory (4kWords)	2.25	400k
Instruction Processor	0.15	7k
Mapping Engine	0.25	12k
Crossbar Switch	0.64	15k
Address Generator	0.06	3.5k
Data-path Switches	0.05	5k
Data-path Registers per FPU	0.20	25k

oscillator, clock synchronisation circuits using delay lines, a $2.1ns$ 56-bit dynamic parallel prefix adder and a multiplier reduction tree with booth encoding implemented in dynamic cascode voltage source logic (CVSL). Each component of the PE is now discussed in detail.

2 Crossbar Switch and Buses

The crossbar switch shown in Figure 3.4 is responsible for routing 64-bit operands around the PE through each of the four external compass buses (north, south, east and west). It is capable of changing the configuration at every processor clock cycle and is implemented using dynamic domino-logic.

2.1 Dynamic processing element bus

The PE bus connections run across the top of the chip in the top level metal and are connected to neighbouring chips through low parasitic pads. A direct chip-to-chip wire-bonding method has been chosen to achieve the smallest parasitic load on the buses. Several designs for wire-bonding have been trialled for various pitches and lengths of wire-bond to determine the pad pitch of the 64-bit buses around the periphery of the PE chip. A single bit of one bus circuit is shown in Figure 3.3. The bus circuits are pre-charged to Vdd in the first half of each clock cycle using pull-up transistors on the periphery of the chip. Each bus has a fast evaluation transistor which can pull the bit-line low if the voltage on the input drops a p-transistor threshold below Vdd. Buses are connected together via programmable links which are n-type pass transistors with a low series resistance. A group of connected buses are also cross coupled to latch the data in the evaluation cycle, further enhancing the speed.

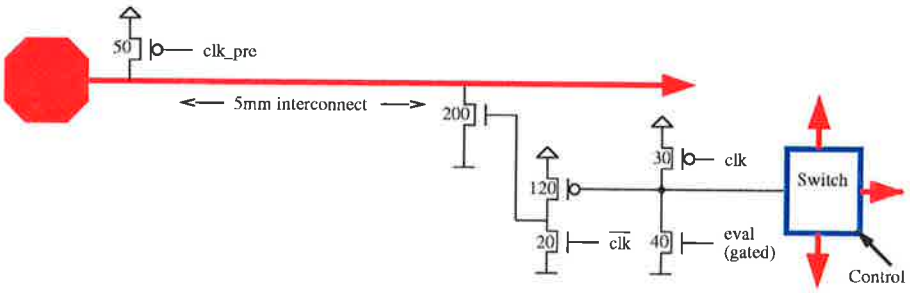


Figure 3.3 A single-bit bus circuit showing pre-charge and evaluation logic.

2.2 Crossbar switch

Three crossbar circuits which pass signals bi-directionally between interconnected buses are shown in Figure 3.4. Crossbar switch circuits and buses have been fabricated in a $0.5\mu m$

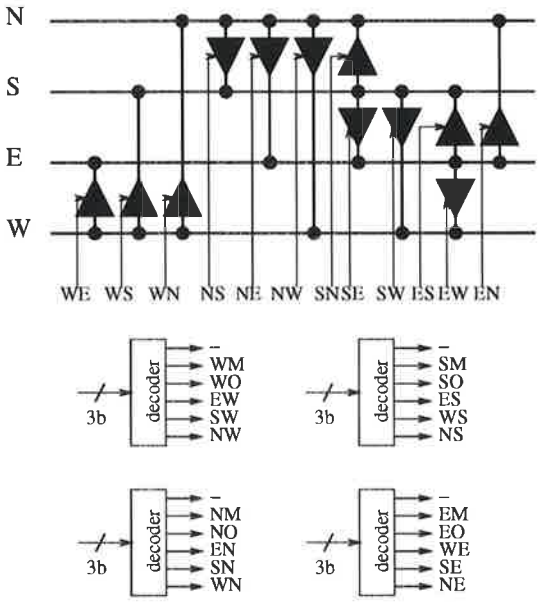


Figure 3.4 Crossbar switch and dynamic bus circuits.

CMOS process to test the dynamic bus strategy for connecting the PEs together. Five test chips have been wire-bonded into a multi-chip module shown in Figure 3.5. This was built in collaboration with the Defence Science & Technology Organisation, Australia. The delay

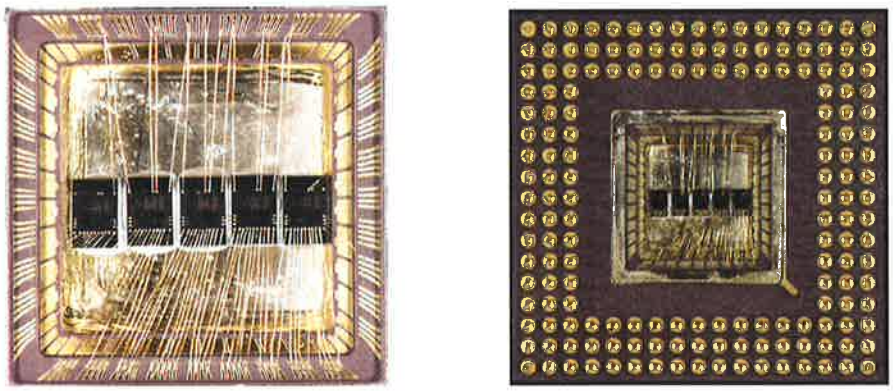


Figure 3.5 Five chips packaged for testing the MatRISC dynamic bus and crossbar switch implemented in $0.5\mu m$ CMOS.

per chip has been measured at $320ps$ which includes two half buses plus the crossbar switch delay, as shown in Figure 3.6. The euclidean distance for a signal to travel in a 4×4 processor module is eight chip delays which corresponds to a delay of $2.6ns$. A conservative cycle time for the module of $5ns$ is achievable in this technology. It is estimated that 256 buses per chip will consume less than $1W$ when operating at $200MHz$ with all buses evaluating.

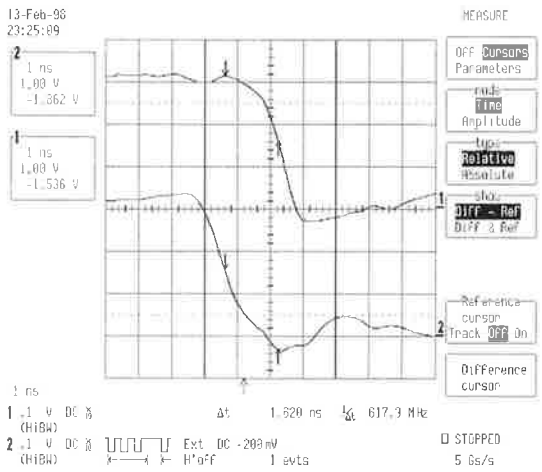


Figure 3.6 Evaluation delay of the dynamic buses through five processing element chips.

3 Address Generator

An address generator (AG) is needed to sequence addresses to the data memory for accessing stored data structures for each memory address port. The data structures are matrices and vectors which have address sequences that would normally be produced by the ALU in a microprocessor under program control. A wide set of mappings can be automatically generated in dedicated hardware based on the alternate integer representation [Knu98,Mar94]:

$$Address = B + (N_1 D_1 + N_2 D_2 + N_3 D_3 + N_4 D_4)_Q \tag{3.1}$$

where B is the base or a constant offset into memory, Q is the modulus, $D_1 - D_4$ are the delta or stride constants and $N_1 - N_4$ are counters. A machine which implements this equation can generate number theory mappings of multi-dimensional data structures which reside in a linear memory space. A four-dimensional AG can access up to four dimensions of a multi-dimensional data structure. If a two-dimensional matrix, A:

$$A = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \end{pmatrix}$$

is stored in row order in a linearly addressed memory:

The AG can be programmed to access A by producing the following address sequences:

- row order as a simple counter [$B = 0$,$Q = 32$,$D_1 = 1$,$D_2 - D_4 = 0$,$S = 0$]:
  0 1 2 3 4 5 6 7 8 9 10 ... 31

Address	Stored data
0	0
1	1
2	2
⋮	⋮
8	8
9	9
⋮	⋮
31	31

- column order $[B = 0, Q = 32, D_1 = 8, D_2 = 1, D_3 - D_4 = 0, S = 0]$:
  0 8 16 24 1 9 17 25 2 ... 31

- sub-matrix in row order $[B = 9, Q = 8, D_1 = 1, D_2 = 5, D_3 - D_4 = 0, S = 0]$:
  10 11 12 13 18 19 20 21

- diagonal vector $[B = 0, Q = 32, D_1 = 9, D_2 - D_4 = 0, S = 0]$:
  0 9 18 27

- constant $[B = 20, Q = 1, D_1 - D_4 = 0, S = 0]$:
  20 20 20 20 20 20 20 20 ...

Other mappings include prime- and common-factor and circulant matrices. An AG design based on the alternate integer representation is shown in Figure 3.7. The AG is accessed through the load bus by programming the mapping registers. The modulus function is speculatively executed to meet the constraints of single cycle execution. Each of eight register sets are multiplexed into the AG and allow up to eight data structures to be referenced and a new address can be generated and a new data structure referenced in the same cycle. This allows the AG to rotate addresses between several data structures, if, for example, there were alternating reads and writes to two different data structures in memory.

4 Memory

There are two memories in the MatRISC processor with differing requirements. The design of the SRAM was conducted by Kiet To [To99]. The SRAM cell design is a six transistor cross coupled inverter pair with current sense amplifiers employed at the end of each bit-line.

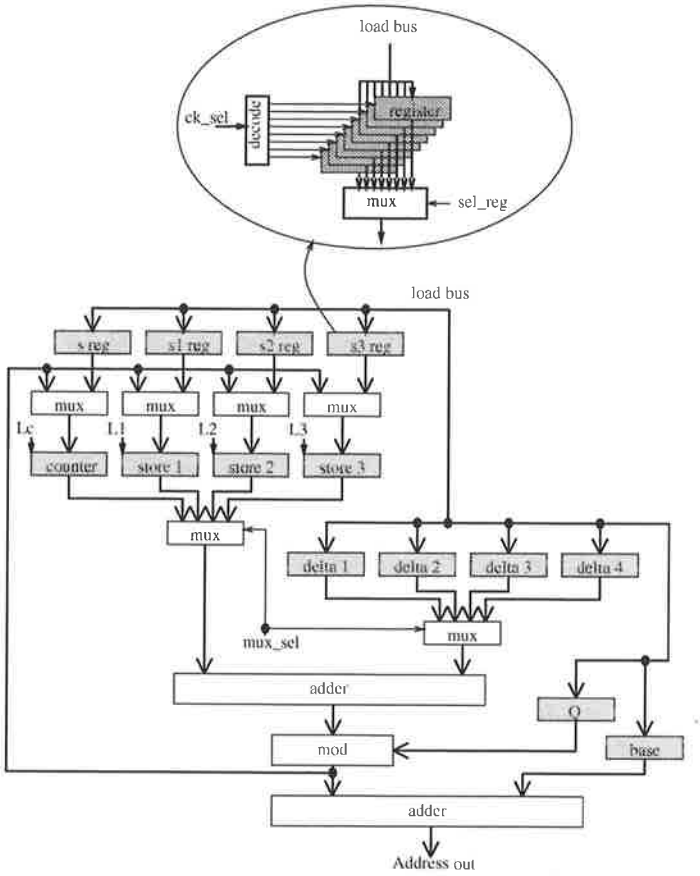


Figure 3.7 Four dimensional address generator with speculative execution.

4.1 Instruction memory

The instruction memory is a single cycle latency dual-port read, single-port write 16-bit memory to support the instruction processor. One cycle latency is needed to fulfil the requirements of speculative execution in the IP for branching or operand pre-fetching. The size of the memory can be relatively small - around 4Kwords of instructions is sufficient for a demonstration MatRISC system. The IM is implemented on-chip to tightly couple it to the IP. Two trial fabrication runs of the IM have been carried out and the measured read access time was $3.5ns$. Part of the IM can be made into a ROM to store kernel routines.

4.2 Data memory

The data memory supports two 64-bit operand reads and a single write per cycle. The address generators keep the DM supplied with deterministic sequences of addresses. A long latency from address to data can be tolerated and so memory bandwidth can be maintained for a potentially large memory bank. Figure 3.8 shows the size of the data memory in Mbits versus matrix size for various processor sizes assuming that three matrices are stored in the

array (two source and one result matrix). Current single chip high speed SRAMs have a

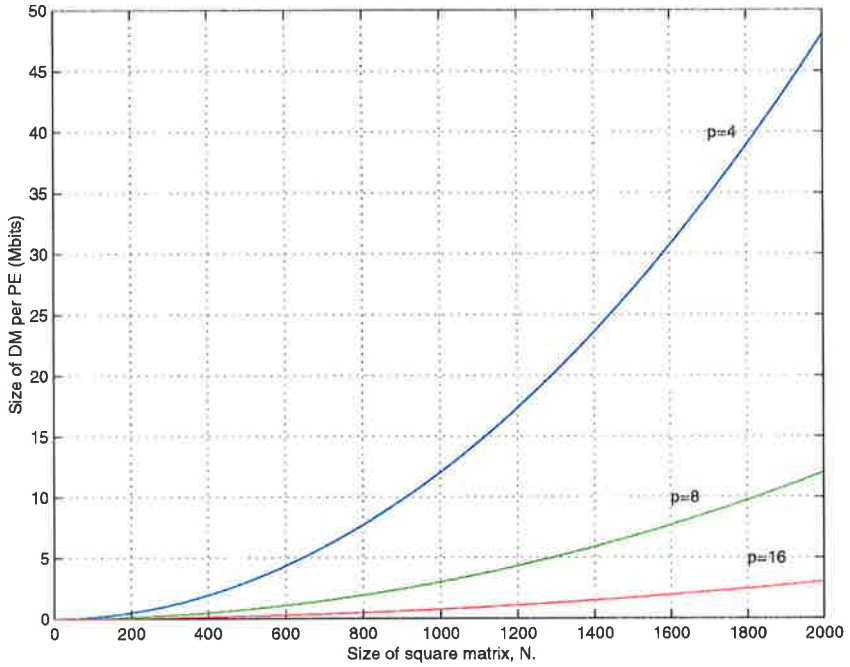


Figure 3.8 Memory size per processing element vs matrix size of three stored matrices for MatRISC arrays of size $p = 4, 8, 16$.

cycle time of under $5ns$ in densities of more than $4Mbits$ in single port configurations. A 4×4 processor array could store three square matrices of order 500 with an external SRAM. A smaller SRAM could be made on-chip but would restrict the problem size.

5 Data Processors

The data processors perform FP addition, multiplication, accumulation and multiply-accumulate. The architecture of a DP is shown in Figure 3.9. The possible configurations of the DP are shown in Figure 3.10 and include multiply, add (different sources), accumulate and multiply-accumulate. The FP adder is a 3-cycle latency pipelined design which is discussed in detail in Chapter 5. The FP multiplier was designed by Chris Howland and uses Booth recoding [Boo51] and is implemented using dynamic CVSL logic in a 54-bit Wallace tree [Wal64].

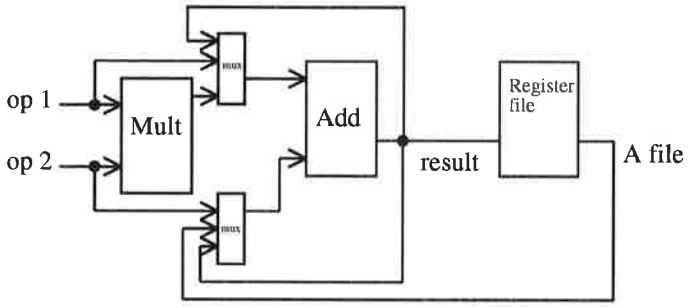


Figure 3.9 Data processor architecture.

6 Instruction Processor Core

The instruction processor (IP) is a 16-bit RISC machine core with a small instruction set architecture designed to operate at high speed and efficiently execute the following tasks in the PE:

- maintaining the generalised mapping engine and stores
- maintaining the address generators
- scheduling VLIWs from the nano-store
- performing data dependent operations (eg. comparisons)
- performing non-atomic operations (eg. division, square root)
- maintaining MIMD synchronisation with other processor nodes through message passing
- interface with host processor.

The features of the IP CORE are:

- 16-bit load-store architecture
- RISC instruction set
- 4KWord instruction memory with dual ports
- single cycle operation for every instruction

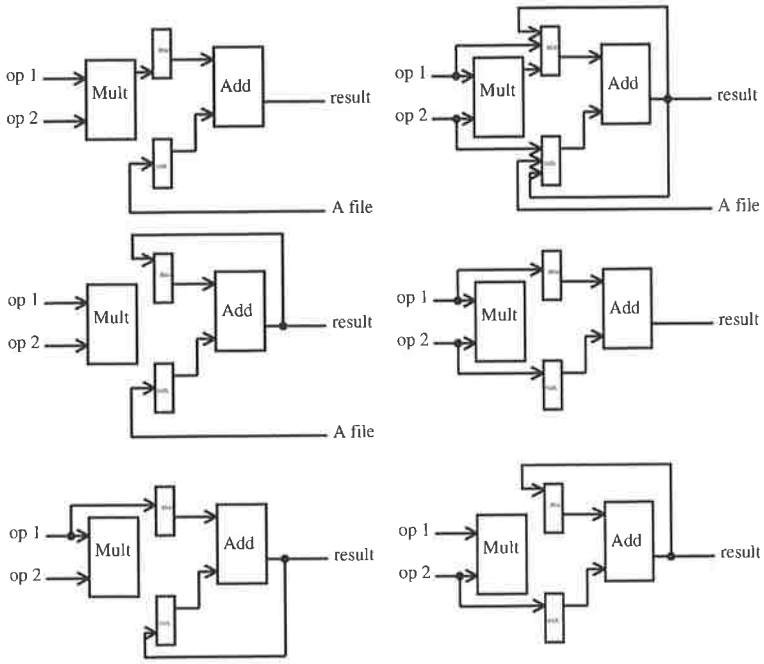


Figure 3.10 Data processor operations.

- no cache
- fast instruction decoding
- pre-fetching of branch instruction targets and operand data
- 16 × 16-bit registers.

A schematic of the IP core is shown in Figure 3.11.

6.1 Programmers model

The programmers model of the IP is shown in Figure 3.12. The data bus is 16-bits wide and the address bus range is 24-bits. The data types supported in the instruction set are bits, words (16-bit), long words (32-bit) and quad-words (64-bit). The programmers model includes 16 word size registers (R0 to RF), a 24-bit program counter, a condition code register, stack pointer and stack. The set of instructions were determined by surveying the operations that are needed in the PE. These included the following critical operations:

- logical: AND, XOR
- arithmetic: ADD, SUB, INC, DEC
- shift: SAR, ROL.

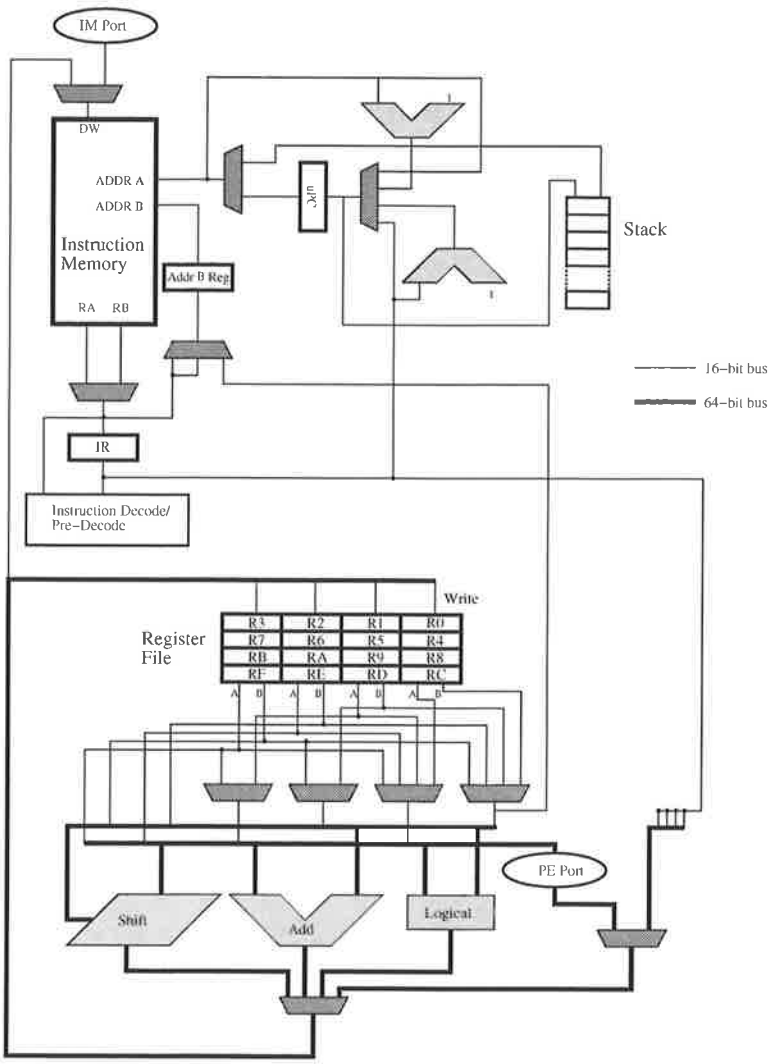


Figure 3.11 Schematic of the instruction processor core.

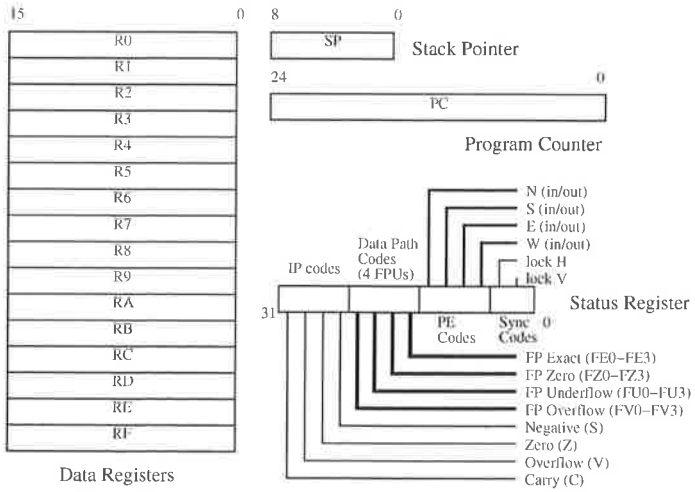


Figure 3.12 Instruction processor programmers' model.

Long word and Quad word arithmetic and logical operations are supported in hardware. Non-critical operations which are rarely used or could not fit in the small instruction set can be constructed from implemented operations for logical operations:
logical or (OR), logical negate (NOT), logical end-around rotate right (ROR), logical shift right (SHR), logical shift left (SHL),
or for the following arithmetic operations:
add with carry (ADC), negate (NEG), subtract with borrow (SBB), compare (CMP), unsigned multiplication (MUL), unsigned division (DIV), long (32-bit) word addition and subtraction (ADDD, ADCD, SUBD, SBBD).

The supported addressing modes are listed in Table 3.2. All modes operate on word data varying address word length depending on the operation.

Table 3.2 Data addressing modes.

Mode	Generation
Absolute Data Addressing	EA=zero memory page address
Immediate Data Addressing	EA=0x0000 + 8-bit data
Register Indirect Addressing	EA=(Rn)

6.2 Instruction set architecture

There are four types of instructions in the IP: I, R, J and D-type. I-type instructions are register and memory operations. These include loads and stores between registers and memory. R-type instructions are register-register operations including logical, arithmetic and moves. J-type instructions are for instruction sequencing including jumps, conditional branches and return. D-type instructions are data-path instructions and control the operation and sequencing of the nano-store. The formats of the four instructions are shown in Figure 3.13.

The instruction mnemonics and format is shown in Table 3.3 with the decoded opcode and a description.

- Rd = destination register (R0 - RF)
- Rs = source register (R0 - RF)
- Addr12 = 12-bit address value

Table 3.3 Instruction processor core sequencer instructions.

Instruction	Type	Opcode	Description
LDR Rd Rs	I	0001	Register load indirect address, EA = (Rs)
STR Rs Rd	I	0011	Register store indirect address, EA = (Rs)
LDZ Rd Addr8	I	0000	Absolute register load from zero memory page, Rd = (Addr8)
LDV Rd Data8	I	0010	Register load immediate (zero extend to 16-bits), Rd = 0x00Data8
AND Rd Rs	R	0100 ALUop=00	Logical bit-wise AND: Rd = Rs AND Rd
XOR Rd Rs	R	0100 ALUop=01	Logical bit-wise XOR: Rd = Rs XOR Rd
SAR Rd Rs	R	0100 ALUop=10	Shift arithmetic right: C=Rd(0), Rd = Rs>>1
ROL Rd Rs	R	0100 ALUop=11	End-around rotate left: Rd = Rs<<1
DEC Rd Rs	R	0101 ALUop=00	Arithmetic decrement: Rd = Rs - 1
INC Rd Rs	R	0101 ALUop=01	Arithmetic increment: Rd = Rs + 1
ADD Rd Rs	R	0101 ALUop=10	Arithmetic addition: Rd = Rs + Rd
SUB Rd Rs	R	0101 ALUop=11	Arithmetic subtraction: Rd = Rs - Rd
MOV Rd Rs	R	0110	Register-register move: Rd = Rs
RET	J	0111(0)	Return from subroutine: PC=(TOS);SP–
BEZ Addr12	J	1000	Branch if equal to zero: PC=Addr12 if Z=1
BNZ Addr12	J	1001	Branch if not equal to zero: PC=Addr12 if Z=0
BGT Addr12	J	1010	Branch if greater than: PC=Addr12 if Z=0 and S=0
BLT Addr12	J	1011	Branch if less than: PC=Addr12 if S=1
BOV Addr12	J	1100	Branch on overflow: PC=Addr12 if V=1
BCS Addr12	J	1101	Branch if carry set: PC=Addr12 of C=1
JMP Addr12	J	1110	Unconditional jump: PC=Addr12
JAL Addr12	J	1111	Jump to subroutine and link: TOS = PC;SP++;PC=Addr12
NII Nano11	D	0111(1)	Issue instruction from nano-store todata-path

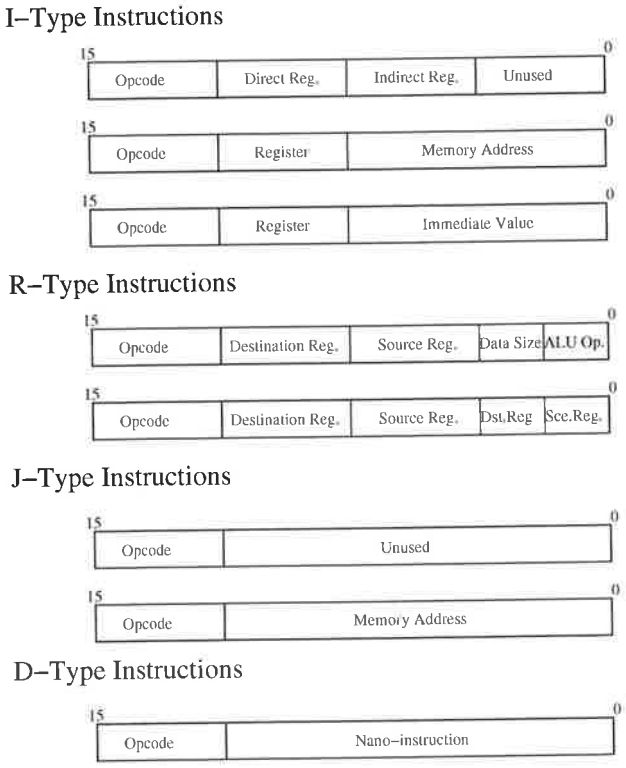


Figure 3.13 Instruction processor instruction formats.

- Addr8 = 8-bit address value
- EA = effective address
- PC = program counter
- TOS = top of stack
- SP = stack pointer
- Data8 = 8-bit data value
- Nano11 = 11-bit instruction sent to nano-store.

The data-size field in R-type instructions determines the size of the data to be used for logic and arithmetic operations. Adjacent registers for long word operations are paired together with the lowest word value as the referring register, a 32-bit register and L0 = R1:R0 forms a long word. There are eight long word registers, L0 - L7. The four quad word (64-bit) registers can be used to perform operations on FP values. These are Q0=L1:L0=R3:R2:R1:R0 and similarly for Q1,Q2 and Q3. The encodings for this are shown in Table 3.4. The byte arithmetic operations inhibit the carry propagation half way through the adder.

Table 3.4 Data size encodings for R-type instructions.

Data	Size	Code
Word	16-bit	00
Long word	32-bit	01
Quad word or float	64-bit	11
Byte	8-bit	10

6.3 Software tools

A set of software tools has been developed to assemble, execute and debug assembly code on a model of the processor core [Hod98a, Hod98b]. A model of the processor core was written in VHDL [Ash96, LSU89] which is merged with the VHDL model for the complete processor array. The model was compiled and simulated using CAD tools from Model Technologies (supplied by Mentor Graphics Corp.). The IP assembler has been written in 'C' [KR88] and interprets assembly language mnemonics converting them into a hex format which can be loaded into the IM. The assembler also extends the instruction set of the system, by interpreting some instructions which are not directly implemented in the IP, into an extended series of instructions. The debugger examines the output trace file of the VHDL simulator and allows a user to trace the execution of the source program after its execution has been completed. This enables interactive examination of register contents, execution path and other information. It also allows break points and single step execution of the trace. The debugger was also written in 'C'. Software support for the MatRISC processor is discussed in Chapter 4.

7 Generalised Mapping Engine

The generalised mapping engine (GME) operates independently of the IP and issues a horizontal VLIW (or nano-instruction) to the PE control path at the rate of one instruction per processor cycle. The generated code is static; each control word fully describes the processor operation for that cycle including register mapping, AG operation, network switch operation, DP instructions and so forth. All scheduling, including taking into account pipeline delays, are carried out by the compiler. In a cyclo-static schedule a relatively small number of horizontal instructions can be executed in a nested loop which unroll as the GME is run.

The GME stores complex sequences of VLIWs in a compact way by using two levels of

instruction referencing. The compiler (described in Chapter 4) reverse maps a sequence of VLIWs stored in memory to a code sequence or 'clique' which is a group of instructions which contain no branches out. Normally, cliques do not have an entry point anywhere in the middle either, but this is not the case in the GME. The instruction clique types are defined in Figure 3.14 and an example is given. The clique instruction specification is given as:

$$< length > \quad < iterations > \quad < startaddress > \quad < linkloop > \quad < endloop >$$

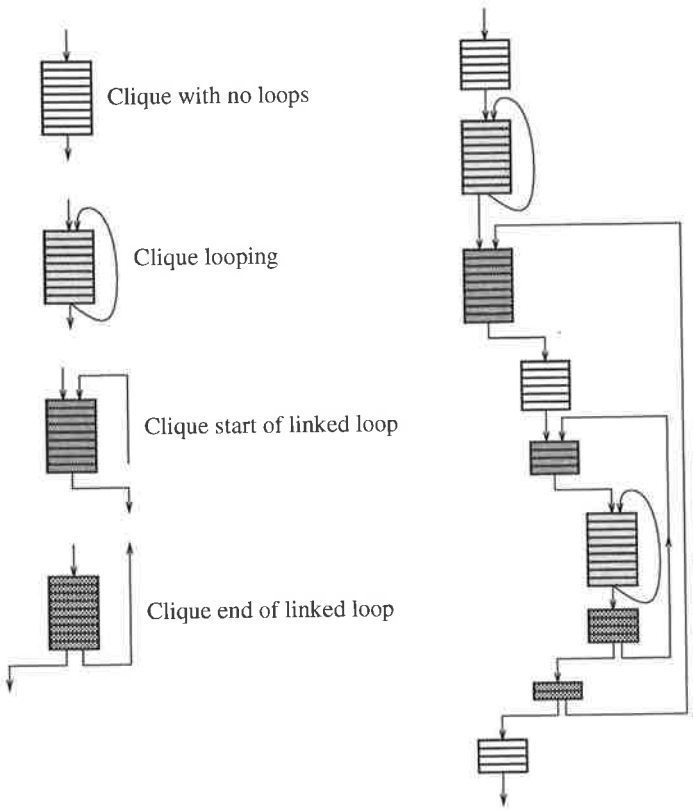


Figure 3.14 Instruction clique types and an example.

By analysing VLIWs with time for a particular algorithm, each bit can be categorised as:

- stationary
- cyclo-stationary
- random.

The GME can map stationary and cyclo-stationary sequences very efficiently and this can lead to very compact code. Sequences which appear to be random and contain little or

no periodicity or regular structures are difficult to compress into efficient GME instruction sequences and can be implemented as a single large clique which may overflow the GME memories. The aim of the GME is to reverse-map a VLIW code sequence through two memories into a linear VLIW clique address which can be implemented as a simple up counter. A start address and the length of execution (maximum counter value) need only be given to execute the VLIW program. A compiler which finds instruction cliques, stationary and cyclo-static schedules to program the VLIW store, clique store and instruction processor (to issue instructions to the GME) is presented in Chapter 4.

The GME is fully pipelined with no feedback paths. At the front end, a hardware nested loop generator unrolls sequential addresses of cliques which are linearly mapped through a writable clique store to access clique codes which form the nano-store addresses.

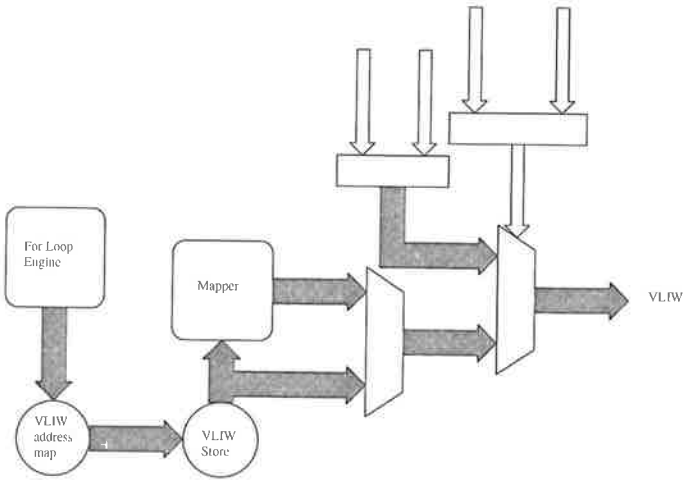


Figure 3.15 Generalised mapping engine.

For example, a clique instruction may be the following:

```
start address = 0, iterations = 5, length = 4
```

This is issued to the first clique counter by the instruction processor. The clique counter then executes the instruction and produces the following clique address sequence:

```
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```

The clique 0 1 2 3 is mapped through the clique store (C) to produce the following clique codes which are the VLIW store address sequence:

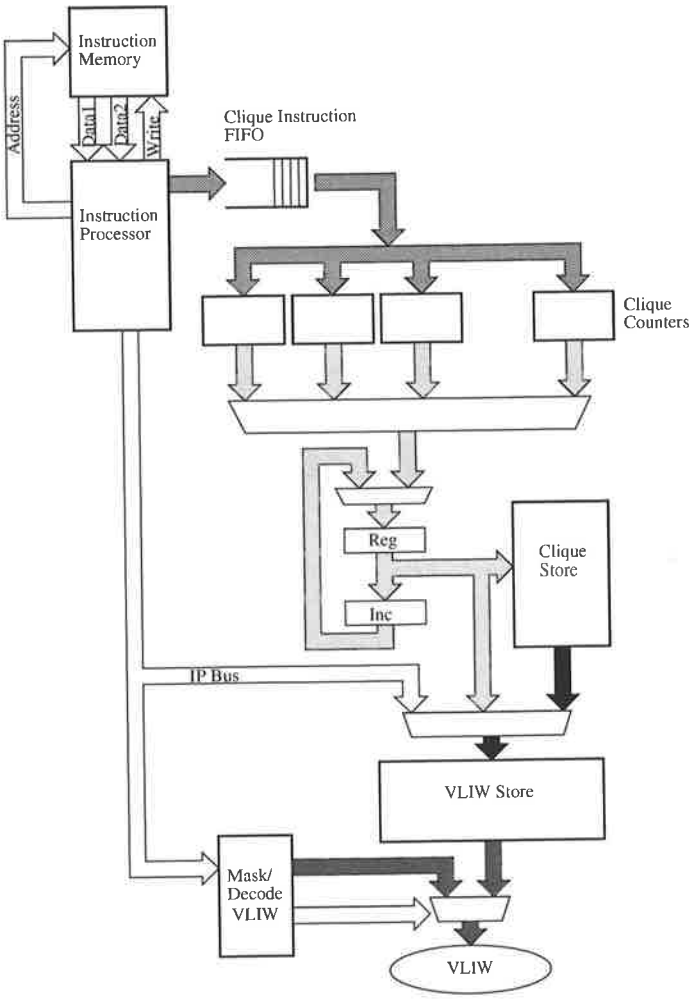


Figure 3.16 Generalised mapping engine architecture.

```
C(0) C(1) C(2) C(3) C(0) C(1) C(2) C(3)
C(0) C(1) C(2) C(3) C(0) C(1) C(2) C(3)
C(0) C(1) C(2) C(3)
```

The VLIW store addresses are used to access the VLIW (or nano-instruction) from the VLIW store (V) which are issued into the PE data-path.

```
V(C(0)) V(C(1)) V(C(2)) V(C(3)) V(C(0)) V(C(1)) V(C(2)) V(C(3))
V(C(0)) V(C(1)) V(C(2)) V(C(3)) V(C(0)) V(C(1)) V(C(2)) V(C(3))
V(C(0)) V(C(1)) V(C(2)) V(C(3))
```

One feature of the GME is that it implements a nested for-loop engine in hardware.

7.1 Nano-store

A writable nano-store is used to store the VLIWs (nano-instructions) that are sent to the 64-bit data-path and control the operation of the PE. The nano-store is relatively small and fast with a 64-bit VLIW available at the output. Each D-type nano-store micro-instruction that is issued by the IP is decoded into a VLIW by the nano-store decoder. A schematic of the nano-store is shown in Figure 3.17. A feature of the nano-store is that a decoupled processor can be made where the clock rate of the IP is higher than the data-path of the PE. A FIFO can be used to schedule instructions and the IP can periodically top up the FIFO. An interrupt alerts the IP if the FIFO is low. If the FIFO becomes empty or no new nano-instruction is issued, a default instruction is issued from address zero.

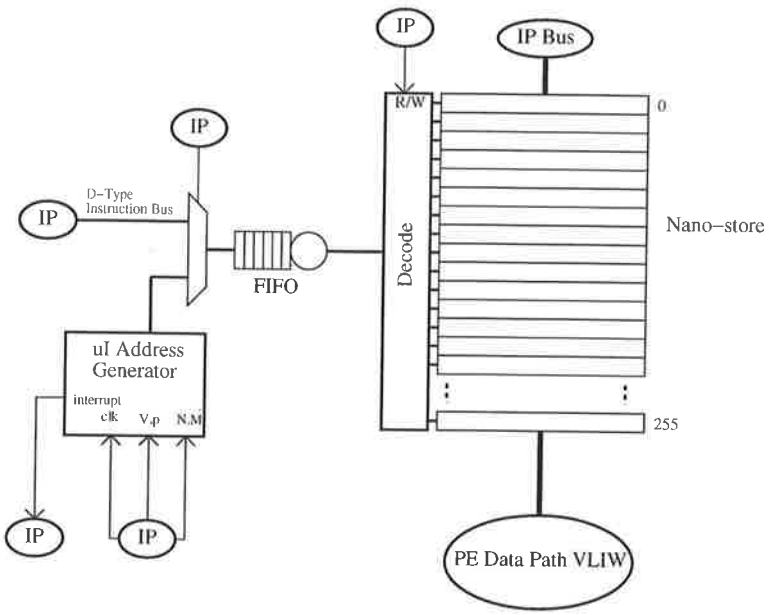


Figure 3.17 Nano-store schematic.

7.2 VLIW generation and loading

VLIWs are generated by the Mcode compiler (which is discussed in Chapter 4) in the scheduling phase where reservation tables to allocate each PEs resources are built. Each VLIW is checked for hazards with preceding VLIWs and allocated to a nano-store address based on the schedule and pre-existing instructions. The table of VLIWs are then sent to each PE in turn where the IP loads each instruction into the nano-store. Instructions in the nano-store can be fully or partially updated by the IP at any time. If longer tables are needed, extra instruction words can be stored in the data memory or instruction memory and loaded into

the nano-store. An example of this may be several algorithms requiring different tables in an interleaved execution sequence.

7.3 VLIW format

The nano-instructions which are required to drive the data-path formatted into the 145-bit VLIW shown in Figure 3.18. There are nano-instructions to drive each component of the PE. The rounding mode is also embedded into the VLIW for each DP which allows efficient implementation of interval arithmetic.

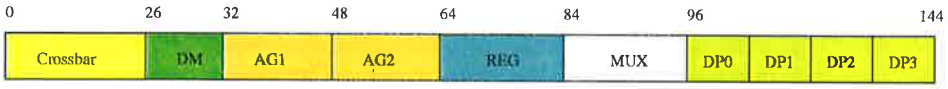


Figure 3.18 VLIW format.

The width of the VLIW is determined by the hardware requirements and the type of encoding. For compilation, the VLIW is formed in the compiler by merging sub-instructions (or VLIW fragments) for switches, registers, multiplexers, gated clock signals, address generator, data memory and floating-point unit control. The format for most sub-instructions is *S2D* plus a list of parameters where S is the source functional unit and D is the destination functional unit. They are described below in the order of the source functional unit. The crossbar switch (CBS) routing instruction 'Xcon' is the exception.

7.3.1 Crossbar switch sub-instructions

There are four enable signals for each of the four buses plus another four signals for the input register/direct multiplexer of the CBS. Reset and clock signals for the input registers are also included. There are a total of 22 signals to drive the crossbar. The configuration of the CBS is made through the 'Xcon' sub-instruction shown in Table 3.5.

Table 3.5 Crossbar switch connection sub-instruction field.

sub-instruction	Connection type	Description
Xcon(con,s)	star	one source bus, rest are destinations
Xcon(con)	star	all buses are destination
Xcon(con,s,d,s,d)	corner,through	two sources and two destinations
Xcon(con,s,d)	corner,through	one source and destination
Xcon(con,s,d,d)	tee	one source and two destinations

There are four basic constructs from which all switch configurations can be made (star, tee, corner and through). The 's' and 'd' parameters refer to the source and destination directions which can be one of the set of compass headings north, south, east and west (N, S, E, W). The constructs can be rotated and flipped to correspond to these bearings.

The CBS sub-instruction fields are shown in Table 3.6.

Table 3.6 Crossbar switch sub-instruction field.

Crossbar switch sub-instruction	Description
x2m(bus)	Write to memory from bus
x2r(bus)	Latch to memory register from bus
x2f(bus,fpu)	Input to FPU (x or y) from bus

7.3.2 Data memory sub-instructions

The DM is comprised of an SRAM with enable and read/write control signals. There is a memory register used to latch data on the data memory data bus, this has a clock and reset signals. The sub-instruction fields are shown in Table 3.7.

Table 3.7 Data memory sub-instruction field.

Data memory sub-instruction	Description
m2x(bus)	Write to crossbar bus bus from memory
m2r(bus)	Latch to memory register from memory
m2f(bus,fpu)	Input to FPU (x or y)

7.3.3 Address generator sub-instructions

The address generator (AG) unit has three control signals to select one of the eight internal counters for which the parameters can be loaded or the counter incremented. There are also reset and clock signals. The sub-instruction fields are shown in Table 3.8.

To allow the IP code to be re-used on different data sets, an indirect addressing mode can be used for the value of the AG register (*reg*). There are eight register sets in the AG, so a programmable register can transform the value of *reg* to another register.

Table 3.8 Address generator sub-instruction field.

Address generator sub-instruction	Description
ag(reg)	Select register set reg to be output and applied to the SRAM

7.3.4 Registers/multiplexers sub-instructions

There are five multiplexers to select either the memory or memory register to be placed on each of the four crossbar output buses and the X and Y DP operands. Another tristate output multiplexer is used to drive either the four crossbar input buses or the DP output to the data memory data bus.

7.3.5 Memory register sub-instructions

A memory register is needed to store temporary results. The sub-instruction fields are shown in Table 3.9.

Table 3.9 Memory register sub-instruction field.

Memory register sub-instruction	Description
r2x(bus)	Write to bus bus from memory register
r2f(fpu)	Input to floating-point unit input fpu (x or y) from memory register

7.3.6 Data processor sub-instructions

Each DP has two input operands and one result. The control signals depend on the type of data processing and number of pipelining stages. The sub-instruction fields are shown in Table 3.10.

Table 3.10 Data processor sub-instruction field.

Data processor sub-instruction	Description
f2x(bus)	Write to bus bus from data processor output
f2m	Latch to memory register the data processor output

7.4 Sub-instruction hazards

A set of sub-instructions may be specified at each machine cycle and some may be used several times. A table specifying the maximum number of allowable sub-instructions in a cycle is shown in Table 3.11. The table is symmetrical, so only half is shown and the maximum number of instances of each instruction are shown along the diagonal. An asterisk indicates that additional hazard rules need to be applied to check for bus dependencies.

Table 3.11 Sub-instruction maximum instances per VLIW and hazard table.

Instruction	x2m	x2r	x2f	m2x	m2r	m2f	f2m	f2x	r2f	r2x
x2m	1									
x2r	1*	1								
x2f	3	3	2							
m2x	-	-	6	4						
m2r	-	-	3	5*	1					
m2f	-	-	2*	6*	3*	2				
f2m	-	-	3	-	-	-	1			
f2x	-	-	6*	-	-	-	5*	4		
r2f	3	-	2*	6	-	2*	3	6	2	
r2x	5*	-	6*	4*	-	6	5	4*	6*	4

8 Clock Synchronisation Unit

The clock synchronisation unit (CSU) is responsible for the management of synchronisation between PEs to facilitate synchronous data transfer. A loss of synchronisation during computations can lead to a bus conflict with unpredictable consequences (data loss and possible crowbarring of the power rails). Discussions of phase-locked and delay-locked loops and circuits for microprocessor synchronisation can be found in [CS96, LDH+94, PR98, vKAPD96, JH88]. Several approaches have been tried to distribute the clock between an array of PEs. A balanced H-tree approach requires a large clock driver and the clock propagation time must be adjusted for each chip. A VCO was designed for use in a phase-locked loop which used a three stage ring oscillator compensated for supply voltage variations. Although the VCO has good linearity, the timing jitter and susceptibility to noise was too difficult to reconcile with the requirement of a large bandwidth.

A control feedback system has been devised to distribute the clocks and tested in an analogue and digital domain. The system clocks are propagated from chip-to-chip and the phase between two chips compared and adjusted using a control feedback loop to keep the phase variation low. An interrupt to the RISC IP Core if the CSU loses synchronisation ensures that no operand bus instructions are issued. An analogue version of the synchronisation circuit is shown in Figure 3.19. A digital version has the same principle of operation but has no analogue control components.

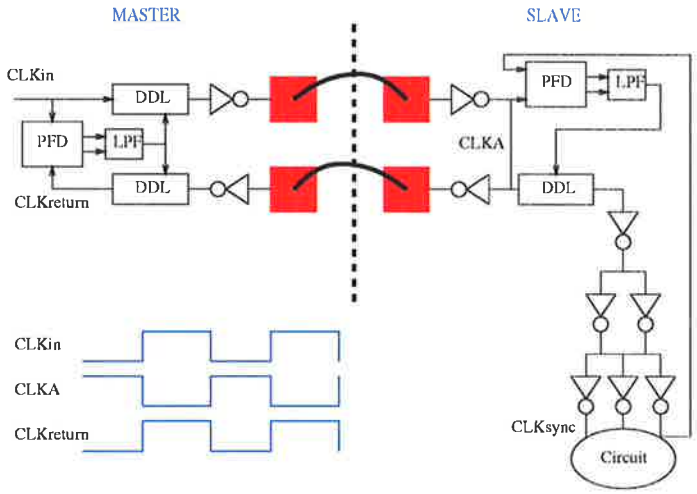


Figure 3.19 Clock synchronisation circuit.

The circuit accounts for the round trip delay of the clock signal between two chips by extending the phase length of the transmitted clock (*CLKreturn*) to be one period long. *CLKA* is then an inverted version of *CLKin* and in-phase. This assumes that the transmission path, including pads, parasitics and the delay line is exactly replicated for the return transmission path and the two VCDLs for propagating the master clock are matched. Once a clock with the same phase as the next chip has been established on-chip, a second delay line loop matches the phase of the chip clock at the end of the clock tree buffer (*CLKsync*) with *CLKA*. The chip clock is then kept in-phase with the *CLKA* thereby factoring out any chip-chip process variation in the clock tree buffers.

8.1 Analogue clock synchronisation unit

A $0.35\mu m$ CMOS chip was fabricated with a test clock system shown in Figure 3.20. The phase of the clocks is adjusted using voltage controlled delay lines (VCDL) connected in a feedback control path. A phase frequency detector drives the up/down signals of a charge pump which places or removes charge from a low pass filter (a large capacitor). This provides

a stable analogue control voltage to apply to the VCDL.

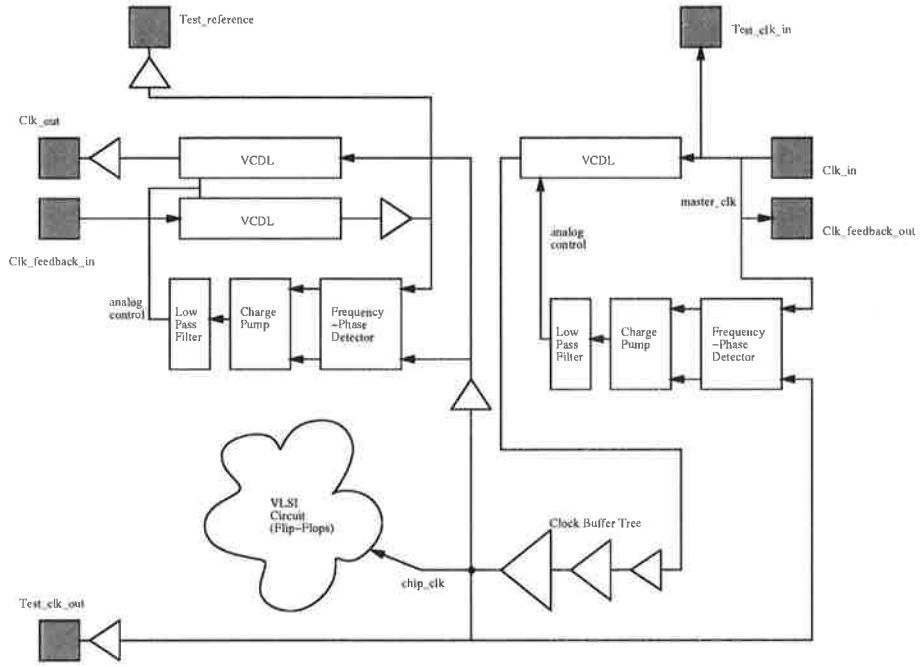


Figure 3.20 Analogue clock synchronisation circuit.

A test PCB containing a row of eight CSU chips was built and tested. The slave clocks did not have a stable lock range and it was found that noise on the PCB contributed to the instability. Two problems became evident, one was the restricted range of delay through the delay line which meant that the minimum frequency was too high for accurate debugging of the board. The other was the noise on the chip power supply buses and the analogue delay line control voltages caused additive clock jitter through each chip. These problems led to the design of the digital CSU.

8.2 Clock synchronisation unit using a digital delay line

A delay line can be constructed from digital elements and a multiplexer as shown in Figure 3.21. Although an analogue delay line with an analogue control voltage for delay has the advantage of a linear phase relationship with control voltage, it is subject to noise which causes phase noise in the output. The digital delay line (DDL) is not subject to the same noise sources but has the disadvantage of a discrete phase relationship with the digital control word as shown in Figure 3.22. This may increase the phase noise compared to an analogue delay line however the maximum phase noise may acceptable for synchronising clocks for use in digital circuits. If a DDL is used in a feedback control loop such as a delay locked loop (DLL),

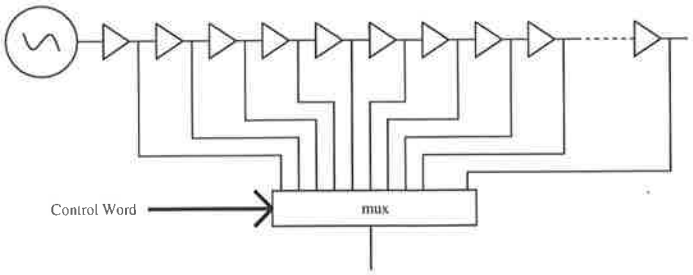


Figure 3.21 A digital delay line.

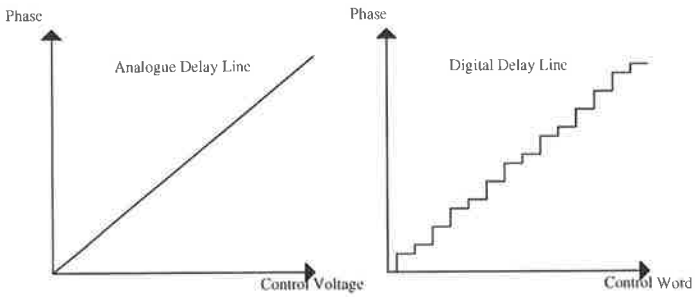


Figure 3.22 Phase characteristics of analogue and digital delay lines.

the control signal of a DDL must change between cycles in a continuous manner (monotonic) for the DLL to remain stable. For example, the update of a binary encoded control can change somewhat randomly and momentarily select output signals which may have already made a transition which cause glitches to appear on the output signal. A frequency-phase detector on the output then glitches because of the frequency transition. A digital loop filter could be designed to suppress this noise however digital filters can be expensive in terms of chip area (they are a cheap capacitor in the analogue domain). If the delay per element is assumed to be much less than the clock period then the time to switch the output signal between two phase stations is also much less than the input clock period. Under this condition, the updated signal will not cause a glitch if the phase change is negative (control word decreasing). A glitch may occur if the phase change is increasing and updates at the same instant the clock makes a transition. This can be overcome by specifying that control word updates occur immediately following a clock transition on the output, making the update of the control word synchronous.

The loop update time can be asynchronous or controlled by a clock signal. If the control word is sampled on a clock edge then the counter can be binary encoded. The loop gain can be controlled by the time delay through each delay cell, making the up-down counter count in programmable steps or making the loop gain a function of the input clock period. The last approach was chosen and a programmable divider latches a new up-down value. A

pass transistor multiplexer can be used which has either direct control from a non-glitching decoder or a left-right shift register with a 'hot-one'. The loop gain must be relatively high since the accumulated phase error can total a period in a relatively small number of clock cycles, depending on the delay per delay element. An XOR gate can be used as the phase detector with logic high indicating a phase difference, however these gates are usually slow. A synchroniser circuit which uses the master clock to latch the synchronising clock is used with an arbiter on the output to arrest metastability because there is a higher than normal probability that a latch will go into a metastable state due to a changing input at the sampling point. An arbiter circuit [Mar90] can force the output of a latch that is metastable to a stable value. A drawback of this system is that the up-down counter is forced to count up or down, it cannot remain stable and will oscillate even when locked to the master clock. A reset signal can be used to force the phase register into a pre-set state and open the feedback loop for several cycles to allow the system to settle. A design for a digital DLL is shown in Figure 3.23.

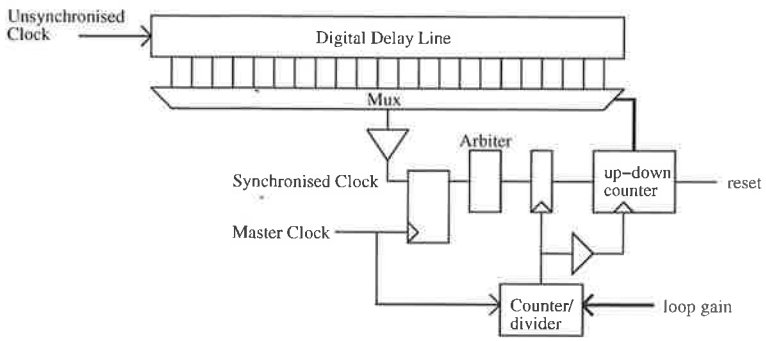


Figure 3.23 Digital delay line loop circuit.

8.3 Noise sources in digital delay lines

There are several sources of noise in MOSFET devices which include (in order of importance) power supply noise, thermal noise and flicker noise. The speed of a gate is proportional to the power supply voltage so the power supply should be stabilised with on and off chip capacitors. A change in voltage across the drain-bulk junction of the PMOS transistors changes the capacitance of the non-linear junction capacitance which changes the delay. This can be reduced by lowering the drain area. A SPICE simulation of the delay elements shows a $28ps$ change in delay per mV of supply fluctuation over the 64 delay elements. Modern power supplies have good regulation ($< 10mV$ peak-peak ripple) and so the jitter due to the power supply is expected to be less than $20ps$ over 64 elements. The power supply ripple is also a low frequency effect.

Thermal noise and flicker noise are negligible compared to noise due to power supply variation, however large delay variations can occur due to changes in the ambient temperature. Significant changes in delay can be expected with warming of the chip but these are slow changes and will be calibrated out. A SPICE simulation predicts an increase in delay for a delay element of $0.5ps$ per degree increase in temperature. Flicker noise can be reduced by making the transistors larger which is in contrast to reducing the drain area to minimise effects due to power supply fluctuations.

Another source of error in the delay calculation is the mismatch between the propagation delay through the delay elements. This is caused by minor differences in FET geometries (which can be minimised by making devices large), differences in gate loading (ensure all elements are identical and have the same load) and variations in doping which is minimised by placing the circuits close together on the chip.

8.4 Digital delay line test circuit

A $0.35\mu m$ CMOS chip has been fabricated which incorporates a 64 element delay line. Each element is a large pull-up to pull-down current balanced CMOS inverter. Registers were placed along the length of the delay line to capture its state using an asynchronous input signal. The delay line input is from a HP $500MHz$ pulse generator which has $100 - 200ps$ of jitter on the input. Capture pulses were generated using a HP16500C digital test system. Output results were collected from a LeCroy $5GSa/s$ CRO.

The number of delay elements to span a single input period along the delay line is captured and the delay per stage was calculated to be $105ps$ (averaged for seven measured devices). The linearity of the delay line is critical to the stability of the DLL. Generated pulses were captured at the start, mid-way and end of the delay line. The delays through both halves of the delay line varied by up to $10ps$. The period of the oscillator input was varied from $4.0ns$ to $6.3ns$ and the number of stage delays spanned by each period was calculated. The delay line is linear to within one delay stage for this technology and suitable for use in a DLL.

9 Summary

This chapter has presented the microarchitecture and design of MatRISC-1 which is an implementation of the MatRISC processor. The processor is implemented in $0.25\mu m$ CMOS and the PE chip contains around one million transistors. Designs have been produced for components including FPUs, GME, AG, digital delay line loop circuits for processor clock synchronisation and memories. The IP core with a minimal ISA and features single-cycle

execution and branch target pre-fetching was presented. Various components of the PE chip have been fabricated and tested.

Chapter 4

MatRISC Software Support

COMPILERS are an integral part of the MatRISC processor array. The program executed in each PE is statically scheduled for the VLIW hardware and the compiler for this program must perform all dependency and hazard checking. Software tools which support the mapping of algorithms to the MatRISC hardware and debugging are presented in this Chapter. A Matlab-like programming language called Mcode is defined which is used to describe the operation of the array as a whole and allows the type and storage pattern of data structures to be specified. Texts on compiler construction can be found in [ASU86, Pol88, Tse90]. A pre-requisite for this chapter is the discussion on MatRISC microarchitecture in Chapter 3.

1 Compiler Overview

A compiler is needed to generate code for the processing nodes in the MatRISC processor from a higher level language which encapsulates the algorithm description. The levels of code for the MatRISC processor are shown in Figure 4.1. A high-level language such as Matlab [Mat99] or parallel-C can be compiled into the intermediate code format called Mcode (MatRISC code) using a high-level language compiler. The high-level language compiler was not implemented, the focus of this work was to build a compiler and tools to support Mcode.

Mcode was created as an intermediate language to expose the programmer to the structure of the processor array so they can drive the algorithm mapping onto the processor array as a whole. A PEcode program, clique instructions, clique code and VLIWs (nano-instructions) are generated for each individual PE from Mcode. PEcode refers to the instruction processor

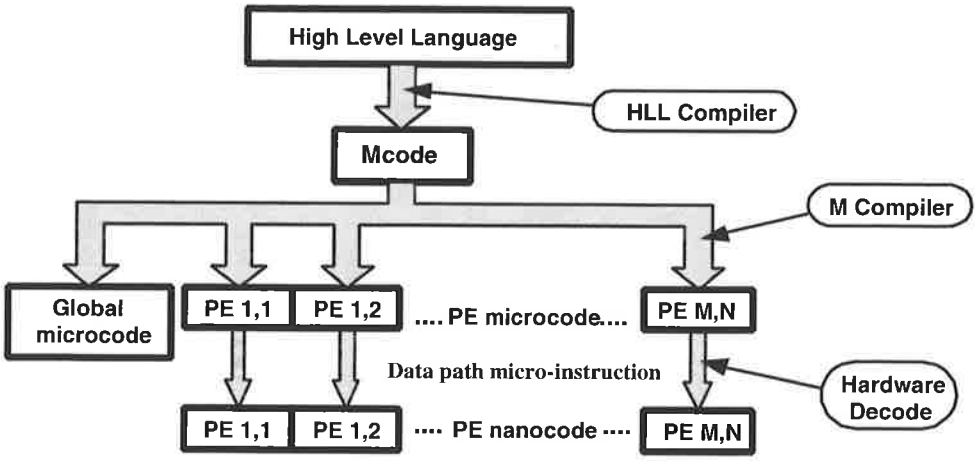


Figure 4.1 Code levels for the MatRISC processor.

(IP) program which is assembled for download into the instruction memory in the IP of each PE. The ISA for the IP was defined in Chapter 3, Section 6. Global microcode is a set of simple instructions issued to all processing nodes and is also generated from Mcode. Clique instructions, clique code and the VLIWs support the GME to issue VLIW sequences to the PE data-path and are defined in Chapter 3.

The Mcode compiler is called 'mc' and was written in Ada [Bar98]. The front-end of the mc compiler parses and scans Mcode then allocates data structures and calls the back-end. The back-end calls driver routines which generate PE code, global microcode and clique instructions based on the PEcode operators and data structure. The VLIWs and cliques are also generated in the back-end. The Mcode compilation process is shown in Figure 4.2. The compiler statically schedules the pipelines in each PE to avoid contention in buses, DMs and data processors. High code reuse is another feature of signal processing algorithms which assists in making the longer and more complex compilation process acceptable. Hence, code for kernel operations can be highly optimised and built into the compiler as a template. This approach has been applied to image-based signal processing algorithms [WOK+97]. The instruction set for the IP is 16-bits wide and includes a minimal set of DLX-like [PH96,SK96] RISC instructions. Each instruction is completed in a single cycle to ensure the processor array continues to execute instructions synchronously.

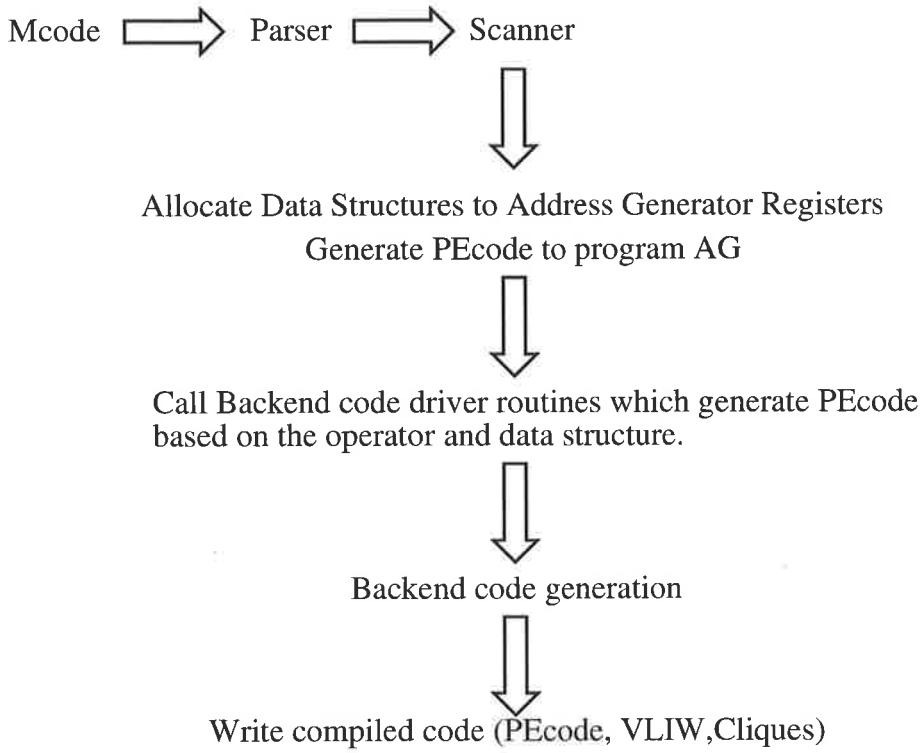


Figure 4.2 Mcode compilation process for the MatRISC processor.

The back-end code generator of the Mcode compiler merges the VLIW sub-instructions and performs code compaction to generate a program mapping for the GME:

Generate VLIWs from sub-instructions (VLIW fragments):

- insert VLIW sub-instructions, PEcode and data constants
- schedule operations based on the knowledge of the data-path architecture and available resources
- allocate registers where each instance of a variable has a unique identifier in a reservation table
- checks for hazards when adding VLIW sub-instructions

Code compaction:

- minimise the set of VLIWs

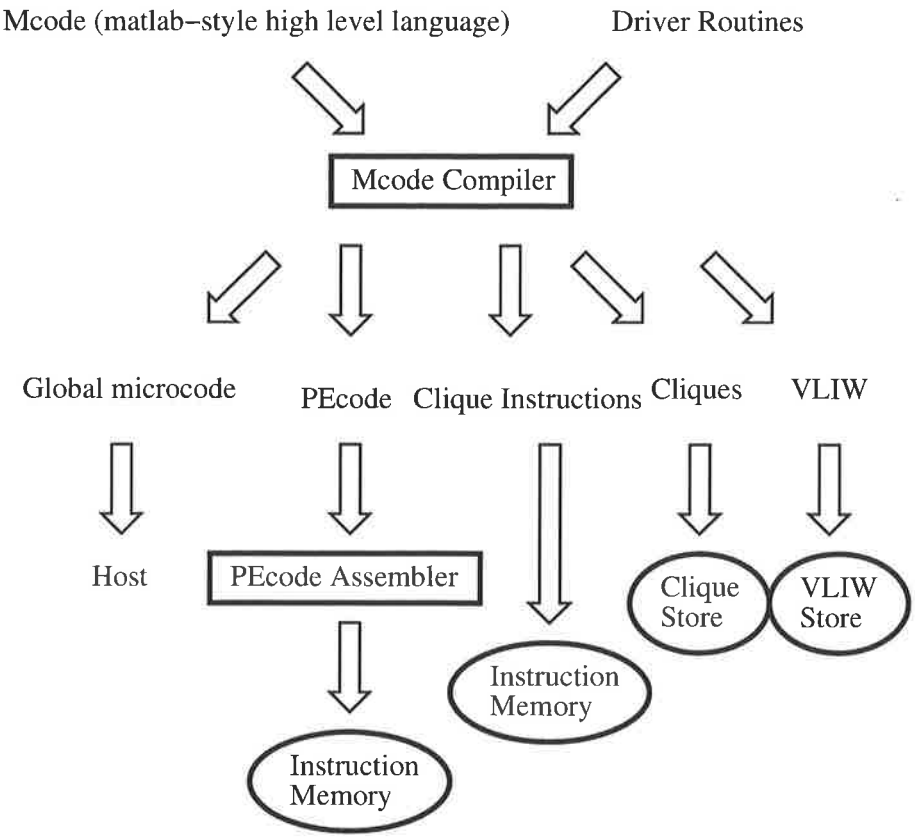


Figure 4.3 Code generation for the MatRISC processor.

- finds cliques of VLIWs which are formed inside loops
- minimise the set of cliques
- linearise the addressing of the cliques by mapping them to a clique store
- build the clique instructions.

The assembly of VLIWs and definition of sub-instructions is given in Chapter 3, Section 7.3.

2 Mcode Definition

Mcode (or MatRISC code) is defined as a set of instructions that describe the behaviour of the MatRISC processor as a whole or it can be used to specify concurrent operations on different PEs with either implicit or explicit inter-node communication. Mcode instructions are readily mapped to the hardware and expose the underlying hardware organisation to the programmer. An Mcode program consists of two parts, the declarations where the data

objects are defined and the body which describes the program flow and operations. All declaration statements must come before the body statements. Some features of Mcode are discussed below.

2.1 Declarations

Pragma declaration

Firstly the hardware configuration must be declared using a pragma statement. This defines the size of the processor array ($size1 \times size2 \times size3$) and the number of data processors for each type.

```
pragma(<machine>,<size1>,<size2>,<size3>,<MAC>,<ALU>)
```

Data classes and mappings

A set of data structure classes which can be stored in the processor are defined as:

- SCALAR : a single numerical quantity
- VECTOR : length n of SCALAR which is a special case of MATRIX
- MATRIX2 : two dimensional structure of SCALAR
- MATRIX3 : three dimensional structure of SCALAR

The compiler supports integer, floating-point and complex arithmetic. Identifiers which are pointers to data structures are declared using the following format:

```
<type> <mapping> <identifier> [<size>,<size>,<size>] @ <pointer>
```

Any identifier name used in the body must be declared along with its size. The `<mapping>` field defines the storage pattern in the memories of the PEs according to Table 4.1 and defines how parallel operations will be performed. A `<pointer>` value specifies the physical address where the data (or start of the structure) resides in the MDMA. Many variables can point to the same data set (eg. a matrix and a vector or a matrix and sub-matrix can point to the same data). `<pointer>` can take the values of an integer address or the set of indirect address registers. The class of data structure is implicit in the number of optional `<size>` fields specified. For example:

```
int () groovy@A1, tie@A2      -- are scalar integers stored in every PE.
float () Koala[100]@A6        -- is a vector of floating point numbers.
complex () BEAR6[100,500]@A4 -- is a 2-D matrix of complex numbers.
```

The @ <pointer> field can be used to specify that a variable points to a specific data structure in the MDMA. A data structure in data memory can be specified as:

```
int (col 2) Alpha[10]     -- a vector mapped into column 2.
```

The mapping can also be optionally specified. For example, SCALAR types can be distributed to the processing nodes that need that value, to all processing nodes or broadcast when required. It is assumed that the value is broadcast to save memory space since a constant matrix can be generated to store a distributed value. VECTORs can be mapped in a variety of ways also.

2.2 Program Body

Each line of the program body contains a three-address code which includes assignment statements, copy statements and control flow instructions. There are also statements which define constructs to parallelise and synchronise Mcode fragments across PEs and allow PEs to pass data explicitly. The compiler generates code from these statements, mostly with implicit communications depending on the identifier data structure class. The identifier can be expressed in one of the following formats:

```
ident
ident[<number1>,<number2>,<number3>]
```

where the number of arguments is determined by the declared data type and <number> can be a single integer, a colon to indicate the whole range or a sub-range expressed as <number1> .. <number2>. A stride can be specified by adding ;<number> after each dimension. Copying one variable to another is done with <ident1>:=<ident2>. Transpose operations can be defined using the transpose operator which is a special case of the copy statement:

```
<ident1>:=<ident2>'
<ident1>:=<ident2>' <op> <ident3>
<ident1>:=<idnet2> <op> <ident3>'
```

The align statement re-organises a data structure in memory so that it can be aligned to another mapping. This is similar to the copy statement except some of the storage used for the old data mapping may be used in the new mapping since the old data is effectively lost.

```
align(<ident>, <mapping>)
```

Table 4.1 Mapping data structures onto the MDMA.

<mapping>	SCALAR
($n1$ $n2$)	put into PE ($n1$,$n2$)
($n1$ $n2$,$n3$ $n4$,...)	put into these PEs
($n1$ $n2$;$n3$ $n4$)	($n1 = n3$ or $n3 = n4$ or $\|n3 - n1\| = \|n4 - n2\|$)
(;) or ()	place into all PEs
(')	place into all PEs
(col $n2$) or (: $n2$)	place into column $n2$
(row $n1$) or ($n1$:)	place into row $n1$
(block q1 q2)	-
<mapping>	VECTOR
($n1$ $n2$)	place whole vector into PE($n1$,$n2$)
($n1$ $n2$,$n3$ $n4$,...)	place whole vector into all these PEs
($n1$ $n2$;$n3$ $n4$)	put into a range of PEs (which must be diagonal or orthogonal)
(;) or ()	direct wrapped map into all rows
(')	direct wrapped map into all columns
(col $n2$) or (: $n2$)	direct wrapped mapping into column $n2$
(row $n1$) or ($n1$:)	direct wrapped mapping into row $n1$
(block q1 q2)	block scatter decomposition:
	$q1 = 1$ (map all rows) or $q2 = 1$(map all columns)
<mapping>	MATRIX
($n1$ $n2$)	place whole matrix into PE($n1$,$n2$)
($n1$ $n2$,$n3$ $n4$,...)	place whole matrix into all these PEs
($n1$ $n2$;$n3$ $n4$)	direct wrapped map into a range of PEs
(;) or ()	direct wrapped mapping into the whole array
(')	direct wrapped mapping into the whole array
(col $n2$) or (: $n2$)	direct wrapped mapping into column $n2$
(row $n1$) or ($n1$:)	direct wrapped mapping into row $n1$
(block q1 q2)	block scatter decomposition, $q1 \times q2$ sub-matrices into each PE

The assignment instruction describes the most common operation on the processor and is of the form:

```
<ident1>=<ident2> <op> <ident3>
```

The <op> tokens are any of the operators in Table 4.2 (which more or less follow the matlab conventions).

Table 4.2 Description of Mcode operators.

Operator	Description
*	multiply
.*	array multiply
+	add
−	subtract
/	divide
./	array divide
∧	power
&	logical and
|	logical or
xor	logical xor
∼	logical not

Array operations are element-wise between matrices and vectors and the dimensions of the two data structures must be the same. The `par` instruction indicates the following code is to be implemented in parallel on the specified processing nodes by each <PE> line. The <PE> field is specified in Table 4.3 to efficiently program groups of PEs. multiple statements can be specified after each <PE>: line. There must be a default field to ensure code is generated for all PEs. Synchronisation points between statements executing on different PEs are specified using [<sync-variable>] at the start of a line inside a `par` construct. The value of the synchronisation variable can be any legal variable name. The code generated at each <PE> line is implicitly synchronised. The break statement means the rest of the <PE> lines in this `par` should not be evaluated in case there is some <PE> statements are not disjoint. Two loops are available with the `par` which evaluate the `par` a number of iterations. The indices of the iteration can be used within the <PE> and body statements inside the `par` construct. These loops are unrolled when PEcode is generated.

```
par(<ident1>=<number1>..<number2>,stride1;<ident2>=<number3>)
 <PE>:
 [sync1]
 break
 <PE>:
 [sync1]
endpar
```

Table 4.3 Fields used to specify sets of PEs.

<PE>	Description
() or (all)	all PEs
(diag)	diagonal PEs only
(col n) or (: n)	PEs in column n
(row n) or (n :)	PEs in row n
($n1$ $n2$)	PE at position ($n1$,$n2$)
($n1$ $n2$,$n3$ $n4$,...)	group of PEs at positions specified
(default)	remaining PEs not specified above
($< aop >< var > < aop >< var >$)	PEs that meet the conditions set

The operations <aop> <var> indicate the PE identifier (*pe-id*) is to be evaluated against this where <aop> is optional and can be virtually any operator (eg. mod, abs) since these expressions are only evaluated inside the compiler. The *pe-id* is stored in the PE at boot time and contains the PEs location in the processor array. There is an additional parameter that can be used inside par statements to index another PE in a relative manner. The reserved words pe.s1, pe.s2 and pe.s3 refer to the currently evaluated PEs location. For example a body data structure A[pe-1,:] refers to the previous PE located in the s1 direction. The inline statement includes instructions as inline assembly for each specified set of PEs. The structure is the same as the par construct.

```
inline(<ident1>=<number1>..<number2>,stride1;<ident2>=<number3>)
<PE>:
<PE>:
endinline
```

Messages

Messages can be sent between individual PEs or groups of PEs using a **send** statement.

```
send(<PE> <ident1|number> -> <PE_dest> <ident2>)
```

where <PE> is the originator of the data <ident1> and <PE_dest> is the destination PE and <ident2> is the destination for the data. The identifier <ident1> can also take constant number values (eg. 1 and 0) which are passed from the IM to the data path. This can be used as a control message passing interface between PEs.

Independent program control flow

The following instructions affect the flow of serial execution of the Mcode.

```
  proc <label>      -- call procedure at {label}
  jump <label>      -- Jump to position {label}
  ...
{label}  ...
  return
```

A single register is used to store the return address and nested procedures are not available. Procedure calls past the first scope are replaced by contiguous code. A label is enclosed in parentheses and consists of any legal variable name.

Data dependent program control flow

Data dependent program control flow operations can affect the program flow of the whole processor array because the data dependencies are non-deterministic at compile time. A single PE or a group of PEs may have control over global branch conditions in the processor. Using the simple message passing protocol, PEs can communicate the local results to each other. A while, for or if statement is used to compare data structures in the DM with another data structure or a constant data structure.

```
while <ident1> <op> <ident2 | number>
 ...
endwhile
for <counter>=0 to <ident>
 ...
endfor
if <ident1> <logic_op> <ident2 | number>
```

```
 ...
endif
if <counter> <logic_op> <ident | number>
 ...
endif
```

The operator <logic_op> is one of the set $\{=, !=, <, <=, >, >=\}$. The identifiers <ident*> can refer to any data structure in the DM. The constant <number> can be embedded into the compiled code sent to the IM. When the number is needed, it is copied from the IM via the DM data path to the ALU.

3 Discussion

Tests showed the mc compiler generated routines produced very compact code for matrix multiplication and addition routines. After compaction, there were just eight VLIWs produced for the matrix multiplication routine which can be applied to any size problem. A major problem with the compiler is the long compile time which is a function of the number and complexity of PEcode operations and also the size of the data structures. Changes to the compiler could make the compile time less dependent on the data structure size. The compiler only operates on a subset of the possible memory mapping types (direct- and transposed-mapping) due to the high number of permutations of operators and mappings, not all were implemented.

Chapter 5

Integer and Floating–Point Adders

The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity which could otherwise be used to carry more digits per word.

John von Neumann, 1946.

ADDITION is the most frequently performed operation in the MatRISC processor. To provide sufficient numerical range, accuracy and to be compatible with most available computer arithmetic hardware, double precision IEEE-754 standard floating-point units are to be used in the MatRISC processor. In this chapter, a method for specifying and designing a wide range of parallel prefix integer adders with and without an end-around carry is reported. An algorithm is presented which is used to synthesise the adders and the delay versus area is mapped for 16, 32 and 64-bit adders. The flagged prefix adder is applied to a novel floating-point adder architecture which has been specifically designed for use with MatRISC processors. Other floating-point adder architectures which can improve the execution time for accumulations and reduce latency at the expense of chip area are also investigated. Implementation studies in CMOS technology are also given.

1 Parallel Prefix Adder Background

VLSI integer adders are critically important elements in the MatRISC processor. They are used in floating-point arithmetic units, the ALUs, address generators, for memory addressing and for program counter update in the instruction processor. The requirements of the adder are that it is primarily fast and secondarily efficient in terms of power consumption and chip area. Adders are also often responsible for setting the minimum clock cycle time in the machine. Discussions of addition techniques can be found in [Omo94, Hwa79, Kor93, Sch90]. Parallel prefix (or tree prefix) adders provide a good theoretical basis to make a wide range of design trade-offs in terms of delay, area, regularity and power. Previous work on parallel prefix adders can be found in [Kno99, Zim98, BK82, KS73, KTM91, HC87, LF80] and a recent implementation study can be found in [MKA+01]. The addition problem can be expressed in terms of kill (k_i), generate (g_i) and carry (c_i) signals at each bit position, i for a width, w adder and with the following equations (where $i = 0, \ldots, w-1$ and c_0 is the carry-in):

$$k_i = \overline{a_i}.\overline{b_i} \tag{5.1}$$

$$g_i = a_i.b_i \tag{5.2}$$

$$c_i = g_{i-1} + \sum_{j=0}^{i-2}(g_j.\overline{\sum_{k=j+1}^{i-1} k_k}) \tag{5.3}$$

$$s_i = p_i \oplus c_i \tag{5.4}$$

The direct implementation of these expressions creates an adder with large complex gate toward the *MSB* position of the carry assimilation path. This single large complex gate will be too slow in CMOS VLSI, so the design is modularised by breaking it into trees of smaller and faster adders which are more readily implemented [BK82].

The *group* generate, G_n^m and *group* kill, K_n^m signals at significance n, are calculated from bit positions m to n:

$$G_n^m = g_n + \sum_{i=m}^{n-1}(\overline{K_n^{i+1}}.g_i) \qquad m < n \tag{5.5}$$

$$G_n^m = g_n \qquad m = n \tag{5.6}$$

$$K_n^m = \sum_{i=m}^{n} k_i \qquad m \leq n \tag{5.7}$$

The carry expression in equation 5.3 can now be expressed recursively using the group generate and group kill signals:

$$c_i = G_{i-1}^0 = G_{i-1}^n + \overline{K_{i-1}^n}.G_{n-1}^0 \qquad 1 \leq n \leq i-1$$

This allows modular sub-adders to be constructed and combined to form trees. At one extreme case, the slowest and smallest ripple carry adder can be recursively constructed using equations 5.5 and 5.7 for $m = n - 1$, although it is *not* a parallel prefix adder. All adders in the parallel prefix adder design space differ only in the carry tree structure - the bitwise kill, generate and sum signals are the same.

Brent and Kung's '•' operator [BK82] is now introduced. Let GK_n^m represent the pair (G_n^m, K_n^m), then the following expression:

$$GK_n^m = (G_n^k + (\overline{K_n^k}.G_{k-1}^m), K_n^k + K_{k-1}^m) = GK_n^k \bullet GK_{k-1}^m \tag{5.8}$$

holds for $0 \leq m \leq k-1$ and $k \leq n \leq w-1$.

1.1 Properties

There are three properties that can be applied to modify the structure of the carry assimilation process using the '•' operator:

- **associativity**: $GK_n^m = (GK_n^k \bullet GK_{k-1}^j) \bullet GK_{j-1}^m$
  $= GK_n^k \bullet (GK_{k-1}^j \bullet GK_{j-1}^m)$
  GK sub-expressions have no serial dependence and can be evaluated in parallel. This is the most important property as it allows the design of *parallel* prefix adders.

- **non-commutativity**:
  $GK_n^k \bullet GK_{k-1}^m \neq GK_{k-1}^m \bullet GK_n^k$
  A generate signal can only be propagated in the direction of increasing significance (group kill is commutative, however this property is not particularly useful).

- **idempotency**: $GK_n^m = GK_n^k \bullet GK_j^m$
  for $m \leq k < j \leq n$ or $m < k = j < n$.
  Two sub-expressions must meet or overlap if they are to form a valid contiguous GK expression.

1.2 Prefix cell operators

Six prefix cells (three black and three grey) can now be defined which implement the • operator as a function of the number of GK inputs (or valency) of the prefix cell. The valency–2, valency–3 and valency–4 black prefix cells (denoted as $\bullet_2$, $\bullet_3$ and $\bullet_4$, respectively) are shown in Figure 5.1 and represent a range of implementable gates in sub-micron VLSI technology with a fan-in limit of four. A static CMOS $\bullet_4$ prefix cell schematic is shown in Figure 5.2 with optional inverters at the outputs to buffer high fan-outs. Grey prefix cells (denoted as $\circ_2$, $\circ_3$ and $\circ_4$) replace black prefix cells at the final cell position in the carry tree

before the sum is calculated as only the group generate term is required to calculate the final carry vector $C = c_0, \ldots, c_{w-1}$.

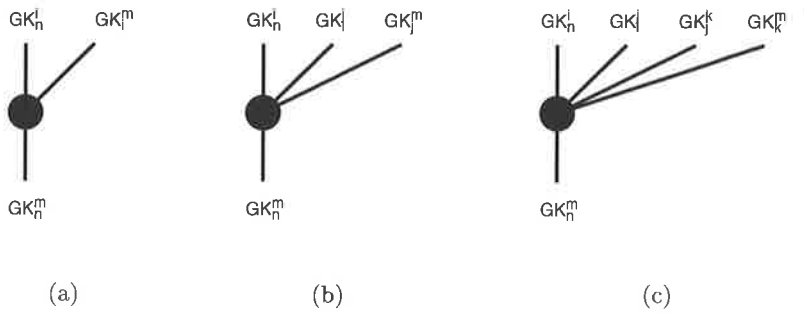


Figure 5.1 Black Prefix Cell • Operators. (a) $\bullet_2$ (valency–2), (b) $\bullet_3$ (valency–3) and (c) $\bullet_4$ (valency–4).

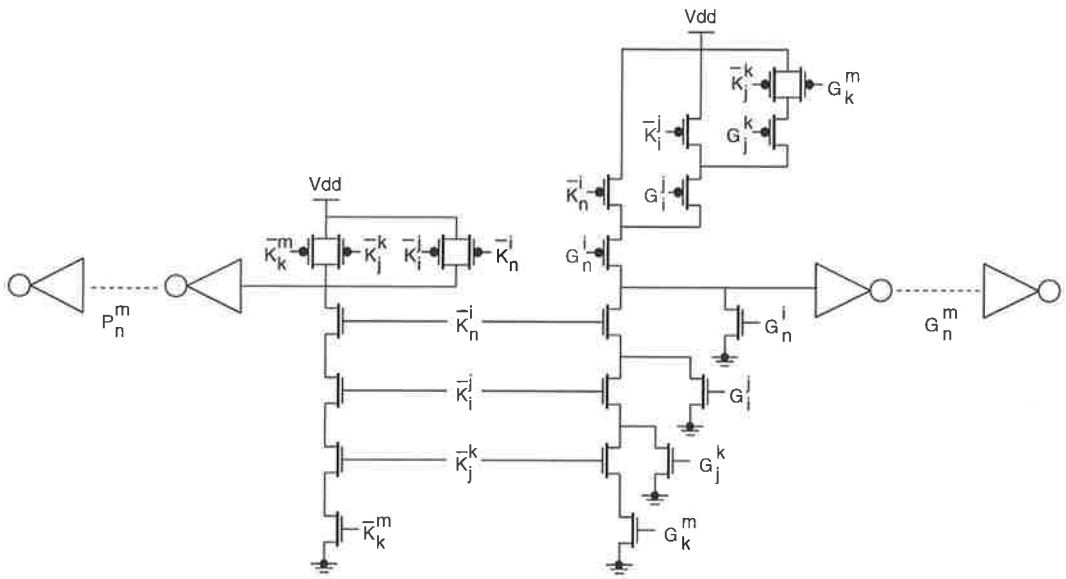


Figure 5.2 A CMOS $\bullet_4$ (valency–4) black prefix cell schematic with optional output buffers.

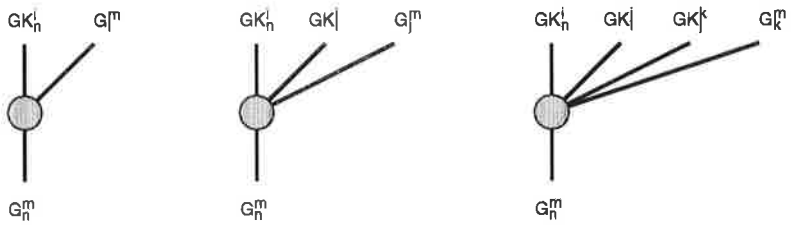


Figure 5.3 **Grey Cell ◯ operators.** (a) $\bigcirc_2$ (valency–2), (b) $\bigcirc_3$ (valency–3) and (c) $\bigcirc_4$ (valency–4).

1.3 Adder implementation issues

The relevant issues to be considered when designing adders for VLSI implementation include chip area, input fan-in, impact of gate fan-out and wire length to the speed, power and logic implementation. The arrival profile of input operands may be non-uniform (eg. output of a Wallace tree in a multiplier) which can lead to the employment of hybrid adder architectures.

1.3.1 Increasing effect of wiring capacitance

As device dimensions continue to shrink in deep sub-micron CMOS technology, the sensitivity of a logic gate to wire loading increases relative to the sensitivity to MOSFET gate capacitance (fan-out). The increased sensitivity of gate delay to interconnect is due to the capacitance per unit length of the interconnect not scaling down with feature size, whereas the gate capacitance does scale down with feature size as shown in Figure 5.4. These trends can be related to adder design such that the width of the adder is largely determined by interconnect pitch which scales down with minimum feature size. Although the interconnects across the adder may be shorter they are also more densely packed and fringing fields will start to dominate the total capacitance loading. Consequently, capacitive loading due to interconnect is significant in the design of wide high speed adders, particularly in carry-select adders where the long wires are in lower levels of metal and drive a large distributed fan-out.

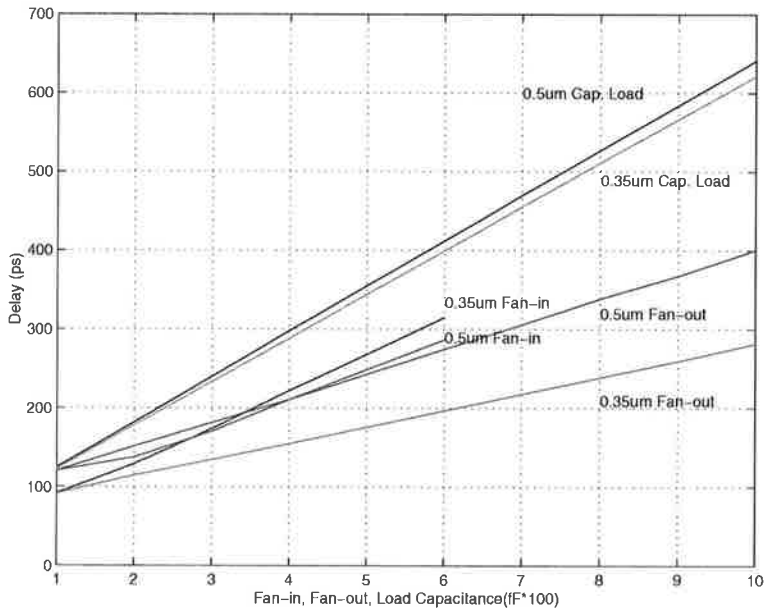


Figure 5.4 Effect of fan-in, fan-out and capacitive load on CMOS gate delay in $0.5\mu m$ and $0.35\mu m$ technologies.

1.3.2 Logic class

Implementations of CMOS adders may be carried out in either a static or dynamic logic class. For a dynamic adder, the exclusive-or gate in Equation 5.1 must be replaced by a non-unate gate.[1] Fortunately, there is a 'don't care' state for p_i in the carry tree if $g_i = 1$ equation 5.1 becomes $p_i = a_i + b_i$, however the p_i propagated to the sum must still be an exclusive-or.

1.4 Previous prefix carry tree designs

Parallel prefix adders which represent a range of the near-minimum logic depth design space are discussed below. All of the adders use valency–2 prefix cells and exploit the associativity property to parallelise the carry assimilation. The Brent-Kung parallel prefix adder [BK82] has a low fan-out from each prefix cell but has a long critical path and is not capable of very high speed addition. The Ladner-Fischer adder [LF80] has a minimum logic depth and recursively doubles the fan-out at each gate until the last stage has a fan-out of $w/2$, and a wire that runs half the adder width. It is the parallel prefix version of the carry-select adder. The Kogge-Stone prefix architecture [KS73] uses more than twice as many prefix cells as the Brent-Kung adder to achieve the theoretical minimum logic depth, $O(log_2 w)$ (using valency–2 prefix cells).

A 32-bit Kogge-Stone adder is shown in Figure 5.5. Each carry bit is determined by a binary tree of black cells where the fan-out is limited to 2 cells. It has a regular structure, high hardware usage and many wires are half of the adder width representing a significant output loading. The Han-Carlson carry tree [HC87] shown in Figure 5.6 reduces the number of cells in the Kogge-Stone adder by adding a final row of cells that lengthens the critical path of the adder by one cell. It also halves the width of the adder thus halving the length of the interconnects relative to the Kogge-Stone adder by employing a radix-4 scheme. The longest wires in the Han-Carlson adder are one quarter of the width of the Kogge-Stone adder. The Kowalczuk-Tudor-Mlynek [KTM91] carry tree shown in Figure 5.8 replaces cells driving long wires by a series of rows with shorter wires which are also one quarter the width of the Kogge-Stone adder, serialising the addition and increasing the critical path of the adder by one cell.

A design example of a new 32-bit adder architecture is given in the next section which illustrates the impact of fan-out and wiring load on the adder delay.

[1] unate gates can glitch if both inputs do not arrive simultaneously causing a false evaluation to occur in dynamic logic.

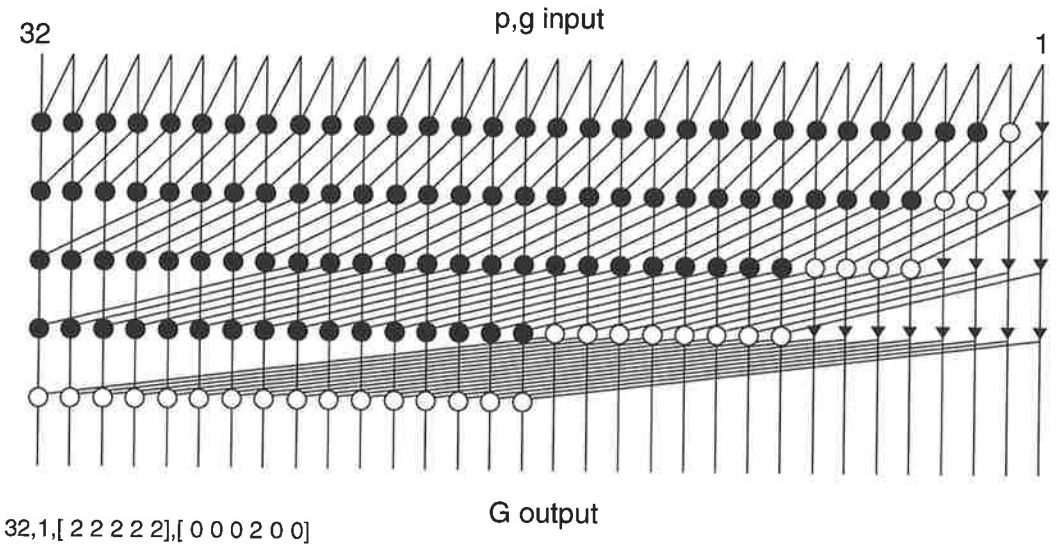


Figure 5.5 32-bit Kogge-Stone adder.

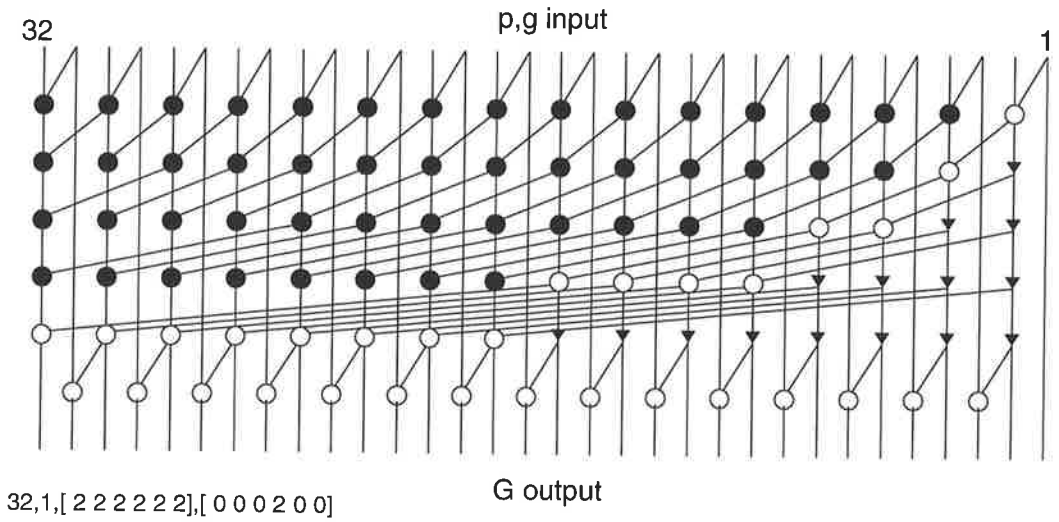


Figure 5.6 32-bit Han-Carlson adder carry tree.

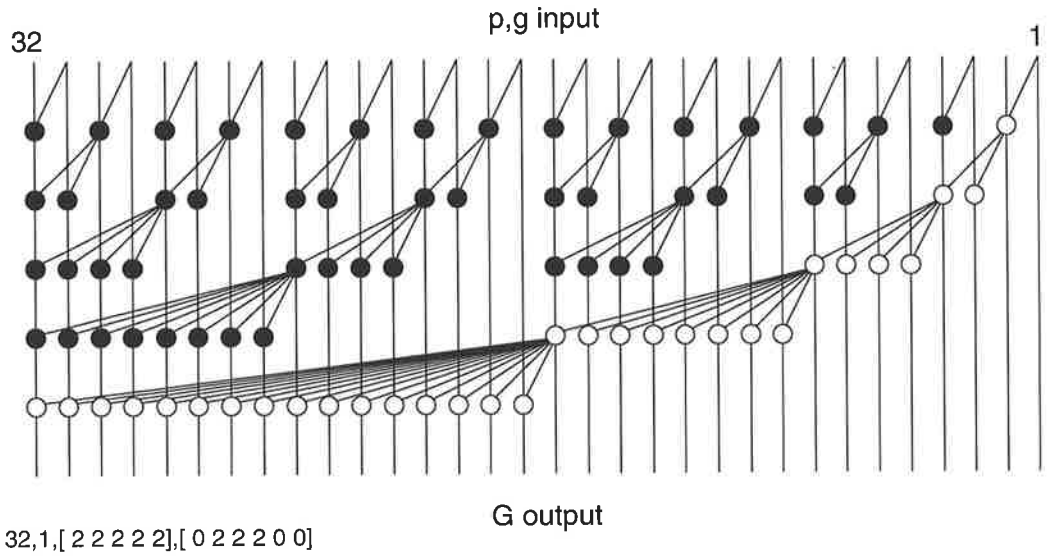


Figure 5.7 32-bit conditional sum adder carry tree.

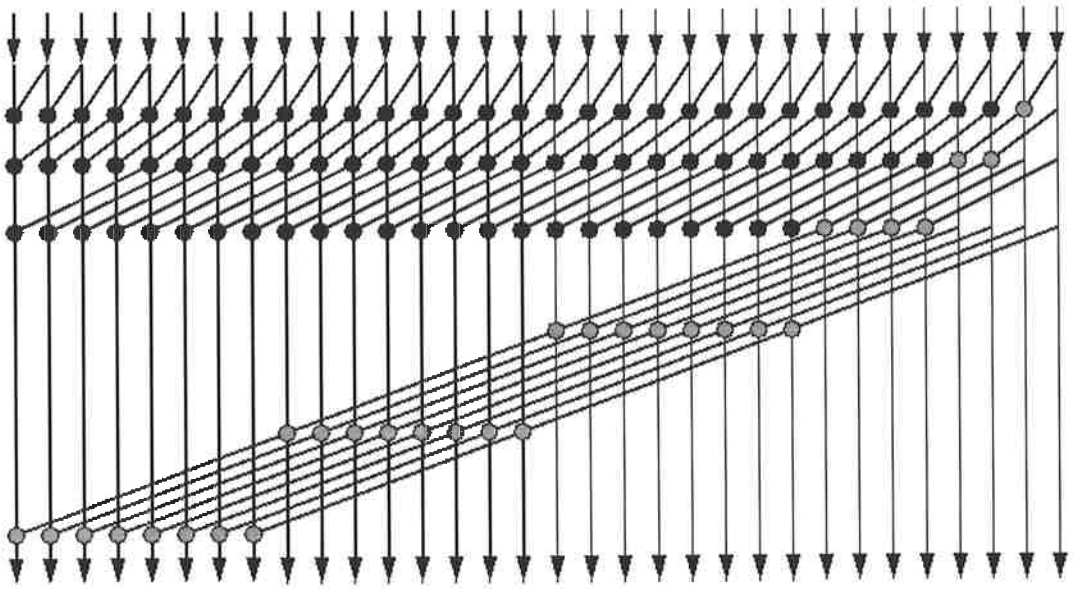


Figure 5.8 Kowalczuk-Tudor-Mlynek carry tree.

2 A Hybrid 32-bit Adder Architecture Based on Reduced Interconnect Lengths

A new 32-bit adder was constructed by merging the Han–Carlson and Kowalczuk-Tudor-Mlynek approaches as shown in Figure 5.9. The delay is one grey cell more than the Han-Carlson and Kowalczuk-Tudor-Mlynek adders and can be made the same by adding four extra black cells at the high order end of the adder as shown in Figure 5.10 which also shows the input and output cells. This new 32-bit adder has 33% less cells than the Kowalczuk-Tudor-Mlynek adder and shorter interconnects than the Han-Carlson adder for the same critical path length.

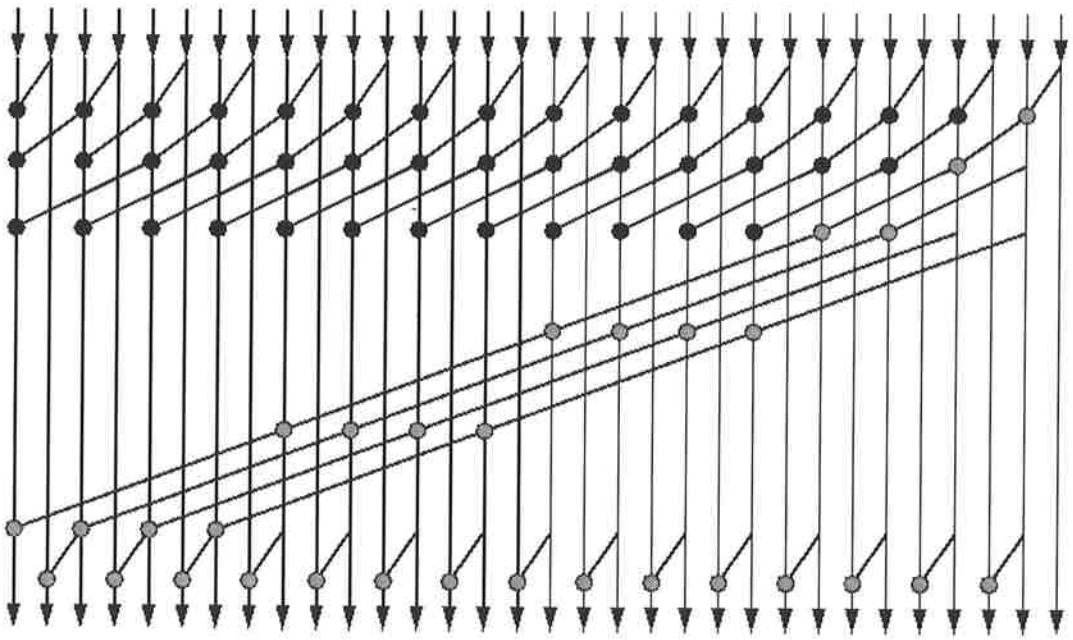


Figure 5.9 New combined Han-Carlson/Kowalczuk-Tudor-Mlynek carry tree.

Table 5.1 Comparison of metrics for different 32-bit adder architectures.

Adder	Kogge-Stone	Han-Carlson	Kowalczuk *et.al.*	New Adder
number of cells	129	80	113	75
Delay (cells)	5	6	6	6
$Area \times time$	645	480	678	450
$Area \times time^2$	3225	2880	4068	2700

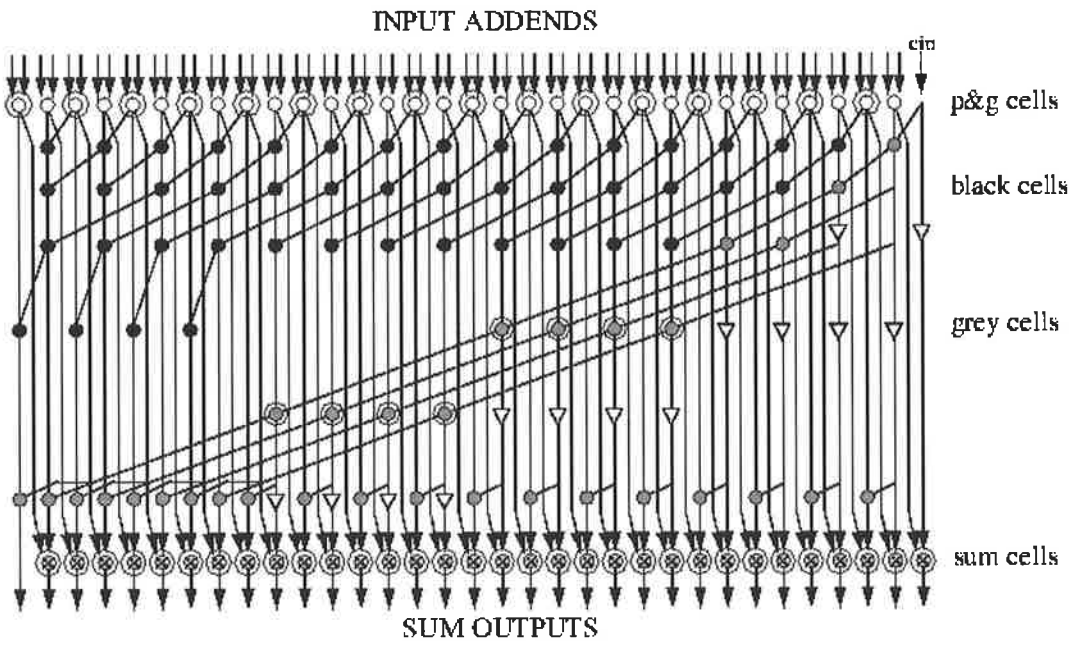


Figure 5.10 Modified adder to reduce delay.

Table 5.1 compares the proposed adder with the other 32-bit adder architectures. The new adder has $area - time$ and $area - time^2$ characteristics that are superior to the other three, however it is still theoretically one cell delay slower than the Kogge-Stone adder. Implementation of this adder in a $0.8\mu m$ Gallium Arsenide technology [LB90] was reported in [BSB97,BSBCL97, BSB96] and showed that it was *faster* than the Kogge-Stone adder, with a delay of $1.25ns$. This is due to the considerable hardware savings and lower capacitive load on the gates because of the shorter wire lengths.

3 Parallel Prefix Adder Design Space

The parallel prefix technique can be applied to a wide range of adder structures, of particular interest are 56-bit, 64-bit and 114-bit adders, which are used in integer and floating-point arithmetic units. Wide adders require large buffers to be implemented as a Fischer-Ladner adder whereas the Kogge-Stone adder uses a large number of cells. The Ladner-Fischer and Kogge-Stone carry trees represent two end designs of minimum depth adders using valency–2 prefix cells. Knowles [Kno99] gives an insight into the parallel prefix adder design space and how they may be specified and studies several key 32-bit minimum logic depth carry tree designs using commercial VLSI layout tools. The study was restricted to using valency–2 prefix cells, in common with all of the other published work presented so far. This work has been extended to larger word-length adders using higher valency prefix cells, and an algorithm is presented in the next section to generate parallel prefix adders. The work presented in this section was reported in [BSL01].

3.1 Synthesis of parallel prefix carry trees

An algorithm is now presented which can construct the design space of prefix carry trees. The carry tree is specified by the width of the adder (w), the maximum number of prefix cell inputs (q_i) in each row (i), and the stride (d) of the repeated carry chains. A set of rules is defined for the iterative construction of parallel prefix adders (with the minimum number of prefix cell rows and the maximum carry assimilation distance per row) below:

- **Rule 1:** Each $\bullet_p$ prefix cell in row i has a number of inputs, $p \leq q_i$ and of the set $p = \{2, 3, 4\}$ and prefix cells have a constant stride across the row.
- **Rule 2:** The span of bits assimilated in row i is recursively multiplied by q_i. This maximises carry propagation distance and minimises logic depth of the carry tree.
- **Rule 3:** The product of all q_i equals the width. This guarantees that all q_i are a factor of the width (w).
- **Rule 4:** Black prefix cells are converted to grey prefix cells if they are in the final position of the carry tree (output to a sum cell) *and/or* drive another grey prefix cell input at the least significance.

Carry trees may also be truncated to a width less than the product of all q_i (rule 3) although this may not lead to the best solution for small delay. Additional conditions for the straightforward construction of end-around carry trees include forcing the carry tree stride to be a factor of the width (q_1 is then a common factor of the stride and width) and enforcing rule 3. This ensures that repeated and shifted carry trees are correctly aligned when prefix cells are placed in positions which are modulo the adder width. A carry tree is specified by the set:

$$w, d, eac, [q_1 q_2 \ldots] \tag{5.9}$$

where $eac = 1$ to generate an end-around carry adder (see Section 3). It was not necessary to specify the lateral fan-out as reported in other work [Kno99] as this can be determined for a minimum depth tree. The iterative algorithm describing how each adder is built is given in Figure 5.11. The carry tree is constructed by creating a stride width slice and copying it contiguously across the carry tree from the LSB to MSB. Sub-adders are needed to complete the carry assimilation in the first rows if the stride is greater than q_1. The prefix cells which extend past the end of the carry tree are placed in cell locations modulo the width. Redundant inputs or cells are trimmed from the carry tree near the LSB if no carry-in is required. A carry-in may be incorporated by retaining these inputs and injecting the carry into them.

3.2 Algorithm implementation and design study

A MATLAB program was written to implement the algorithm and run to find the area and delay for all valid 8-bit, 32-bit and 64-bit adder carry trees for a commercial $0.25\mu m$ CMOS technology. This study is intended to compare relative delays of carry trees under a set of implementation assumptions and is not an absolute indicator of adder delay.

The design strategy is that prefix cells are constructed using static CMOS gates and consist of a complex gate and zero or more inverters to buffer the outputs. For higher valency prefix cells beyond 4, the complex gates need to be split to reduce the delay. The transistor sizing for each complex gate is the transistor stack height times minimum size and follows standard CMOS sizing rules [WE95]. Zero or more inverters are added to each prefix cell output to minimise the delay based on this model. The buffers are individually sized based on three times the previous stage size to minimise delay. Although this is a simplistic approach to transistor sizing, it achieves an acceptable trade-off with design complexity and delay. The alternative is to minimise the delay of each adder using linear programming techniques to tune each transistor size which is impractical for such a large number of designs.

The prefix cells were characterised by building a table of delays and areas based on HSPICE simulations derived from layouts for each prefix cell using typical process parameters at

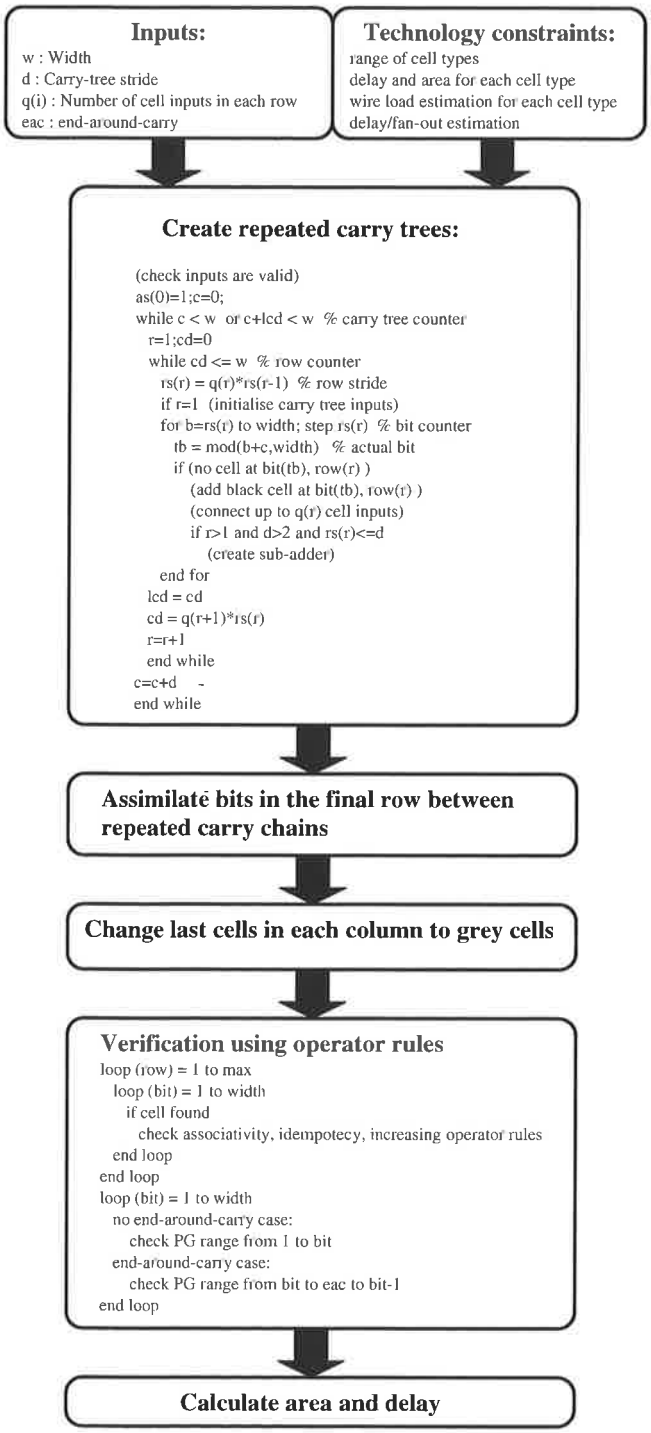


Figure 5.11 Algorithm for generating parallel prefix adders.

$105C$. The prefix cell delay parameters (as a function of driving prefix cell valency) include the delay of the complex gate and per fan-out delay for each receiving cell valency or inverter as a function of drive strength. The effect of wiring was taken into account based on RC delays by adding a delay per cell pitch traversed for each wire as a function of driving cell valency or inverter strength. It was determined that the time of flight delay is negligible for the short distances traversed (up to 0.5mm) and it is assumed a good ground return strategy is in place to reduce its effect [CBF01]. It is also assumed that all input operands arrive simultaneously.

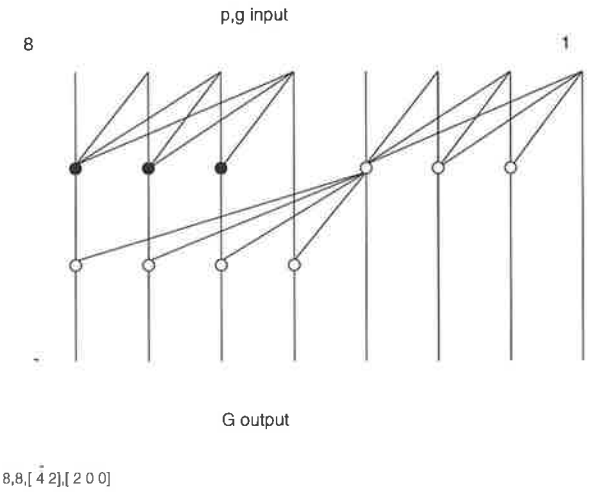


Figure 5.12 **An 8-bit $0.25\mu m$ carry tree: good $area - time$ and $area - time^2$.**

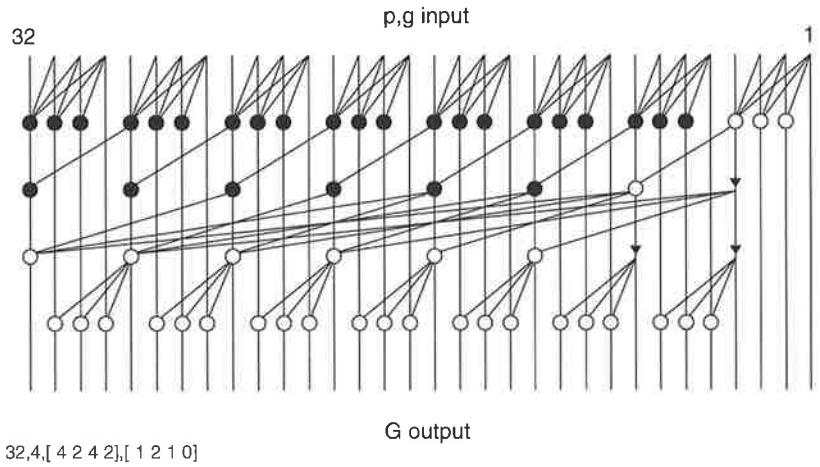


Figure 5.13 **A 32-bit $0.25\mu m$ carry tree: good $area - time$ and $area - time^2$.**

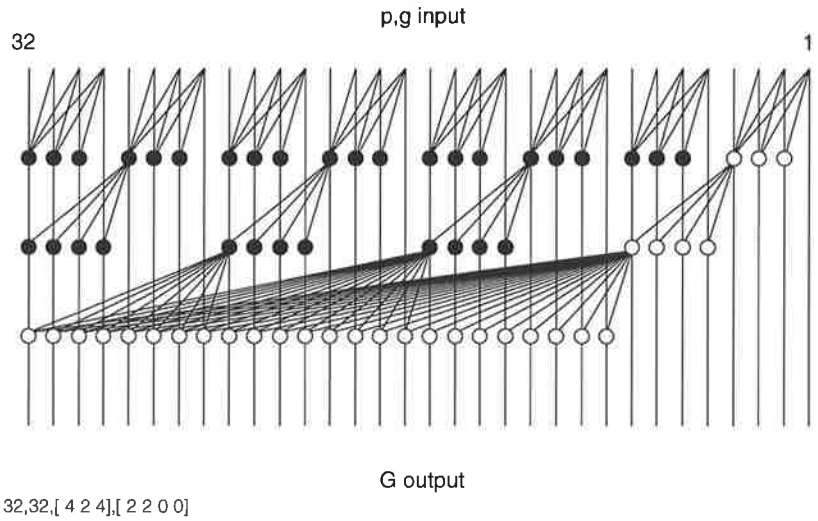


Figure 5.14 A 32-bit $0.25\mu m$ carry tree: smallest delay after Kogge-Stone carry tree.

Generated drawings of noteworthy carry trees are shown in Figures 5.13 to 5.15. The string in the bottom left-hand corner of each carry tree is the input specification (5.9) followed by the number of inverters needed to minimise the delay for each row (top to bottom) using the model described. Each adder is functionally verified by the program after construction by applying the $\bullet$ operator properties from each output bit to check the range of G.

A map of the area-delay relationship for around 120 generated carry trees are shown in Figures 5.17, 5.18 and 5.19 for nominal widths of 8-bit, 32-bit and 64-bit, respectively. The delay does not include the bitwise propagate and generate or the sum cells. Two curves are also shown in each figure for $area - delay = constant$ and $area - delay^2 = constant$ where the constant value is the adder design at the minimum of the respective metric. Other solutions close to these curves are also a good choice if this is the design metric. The end point carry tree designs of Kogge-Stone and conditional sum along with the Han-Carlson carry tree are marked. Wider adders than that specified also appear in these figures and are due to the product of the row input sizes being larger than the specified width (see Rule 4). These wider adders may be truncated to the specified width, however, this does not significantly reduce the delay. Although the $64, 1, [4\ 4\ 4]$ adder in Figure 5.19 has the smallest delay in this study (three rows of valency–4 prefix cells with low fan-out), it probably contains too many wires to be practically implemented.

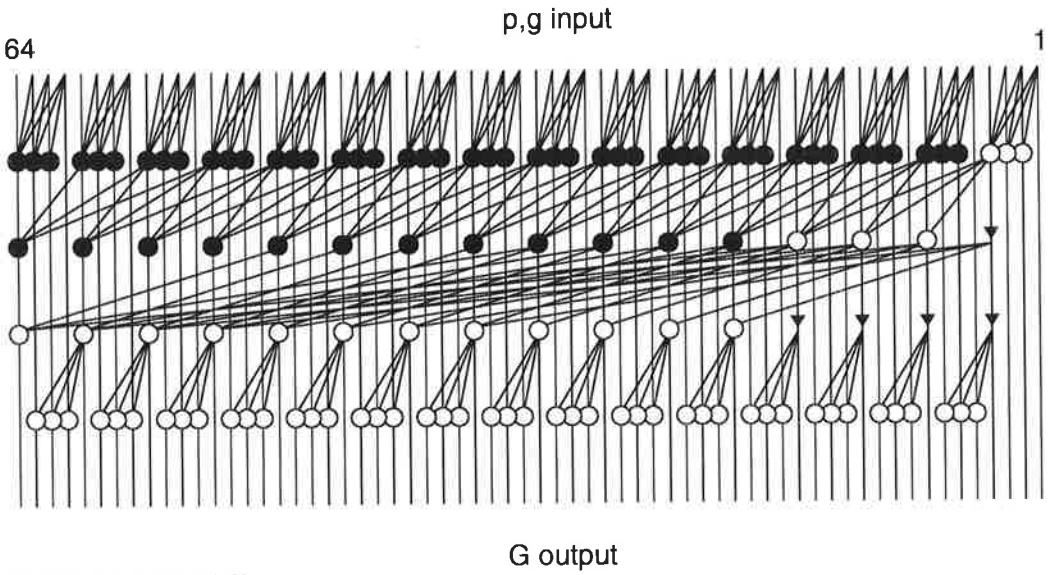


Figure 5.15 **A 64-bit $0.25\mu m$ carry tree: good $area - time$ and $area - time^2$.**

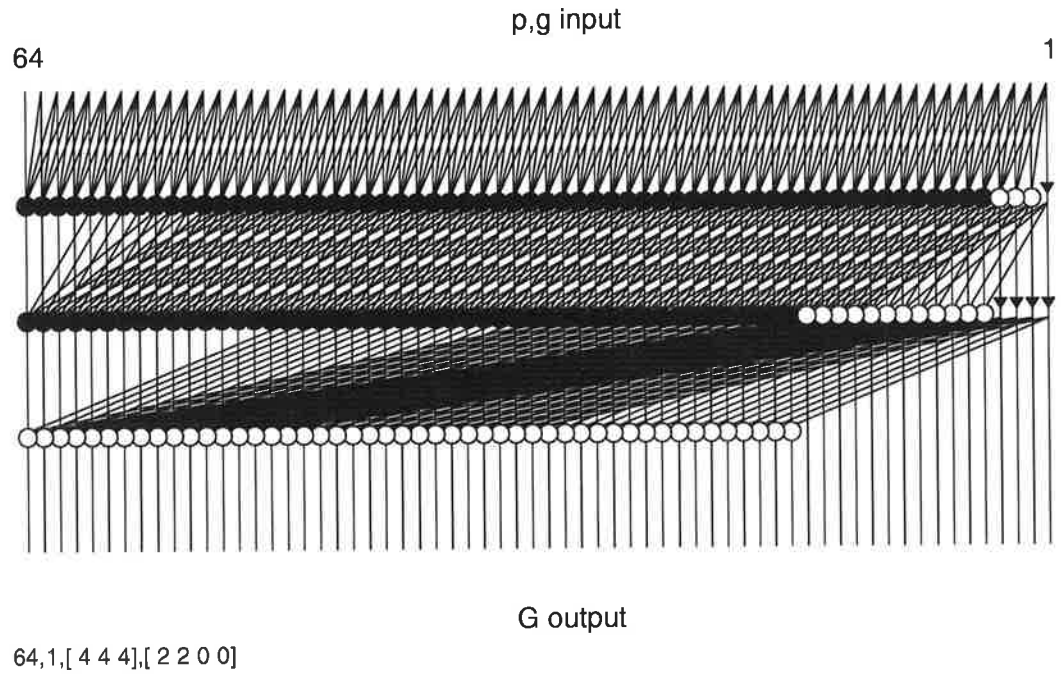


Figure 5.16 **A 64-bit $0.25\mu m$ carry tree: smallest delay.**

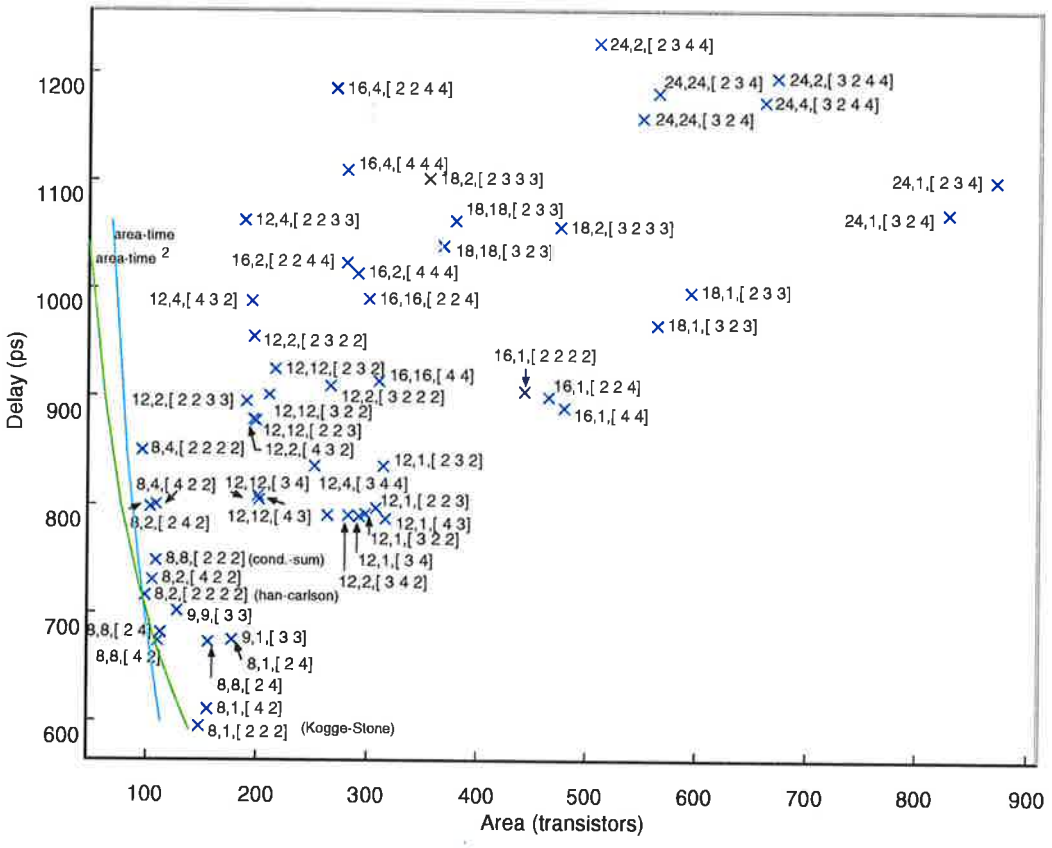


Figure 5.17 Area vs delay for 8-bit carry trees.

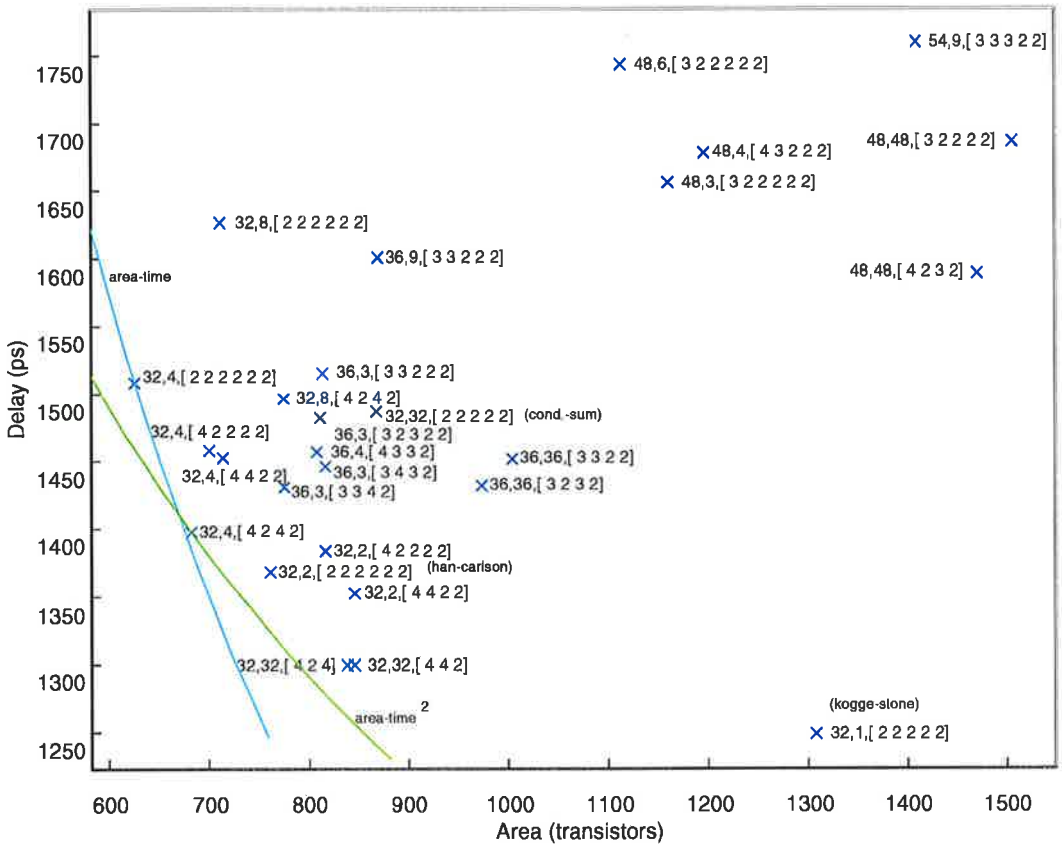


Figure 5.18 Area vs delay for 32-bit carry trees.

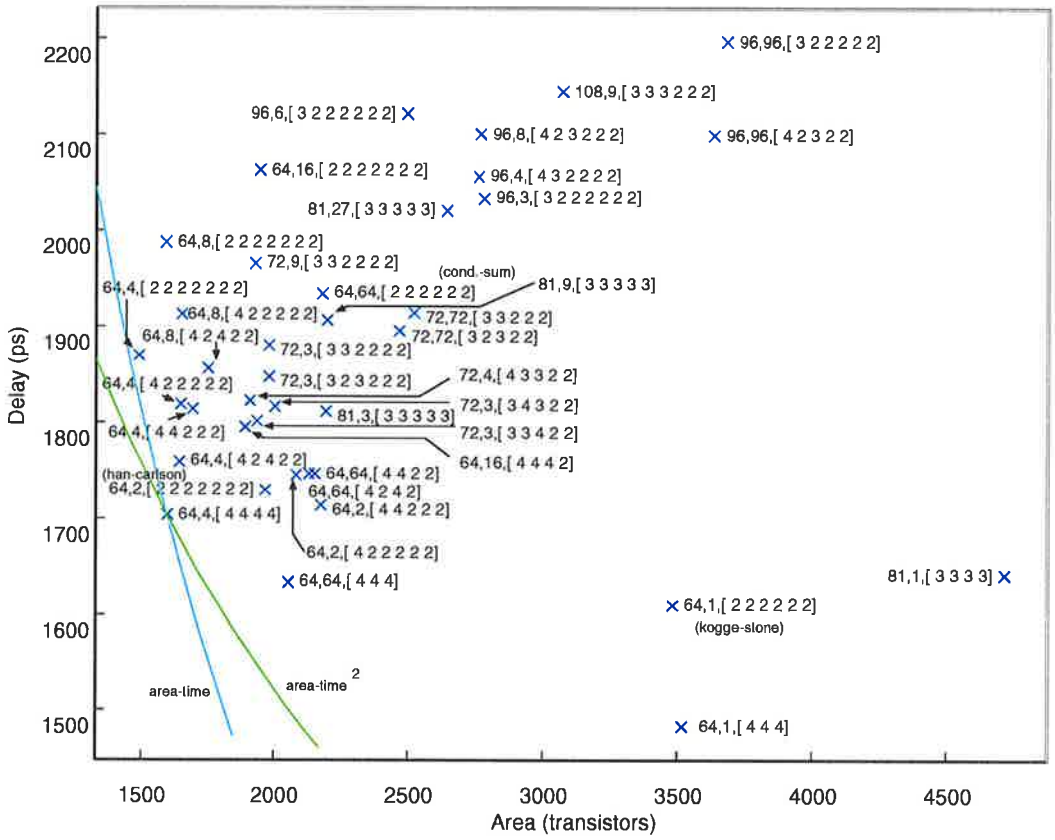


Figure 5.19 Area vs delay for 64-bit carry trees.

4 Implementations of 56-bit and 64-bit Parallel Prefix Adders

A 56-bit, radix-8 carry tree implementation of a new hybrid adder architecture is shown in Figure 5.20. Although this adder was implemented before the study in the previous section was carried out and is slightly slower because of the serial carry path, it serves as a useful data point.

There are six delay cells in the critical path and this word-length allows the adder to be constructed as a direct combination of the Han-Carlson and Kowalczuk-Tudor-Mlynek adders without the need for regaining an extraneous cell delay, as was the case in the radix-4 32-bit adder presented in Section 2. High fan-out points are buffered along the paths of non-critical nodes to speed up the carry propagation across the adder.

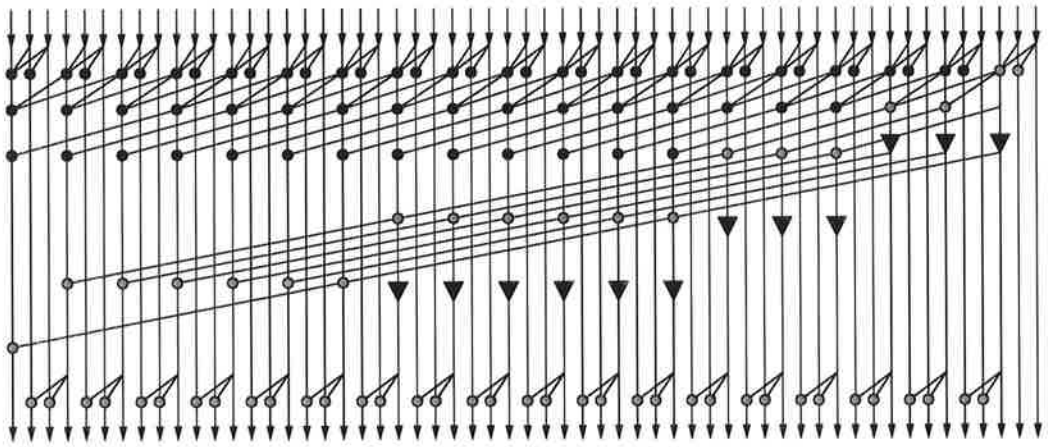
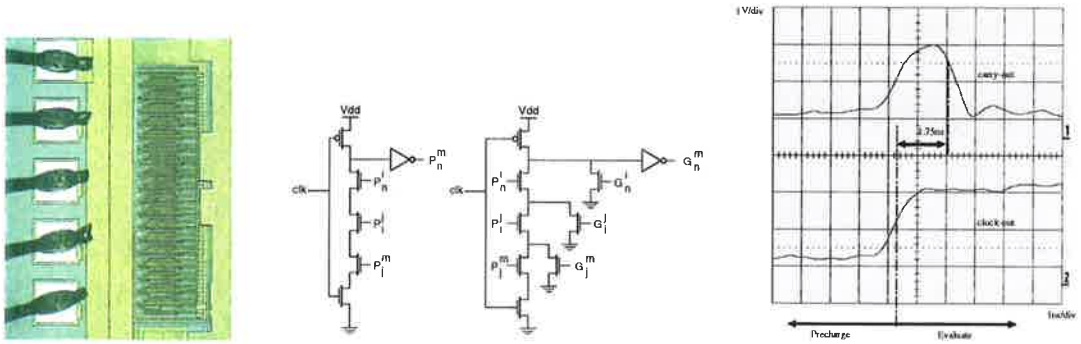


Figure 5.20 A 56-bit mixed radix-8/2 carry tree.

This adder was implemented using dynamic domino logic (Figure 5.21b shows a radix-8 gate) in a $0.35\mu m$ CMOS process. It has a pre-charge time of $0.6ns$, an evaluation time of $1.75ns$ as shown in Figure 5.21c and an area of $0.08mm^2$, dissipates $97mW$ at a power supply voltage of $3.3V$ with a $200MHz$ cycle time and nearly 100% circuit activity. Figure 5.21a is a micrograph of the circuit.

A radix-16, 64-bit parallel prefix adder was designed to determine if an even higher radix will give any speed benefit. Three radix-16 adders are summarised in Table 5.2 and one is shown in Figure 5.22. All three employ radix-16 cells in the first row and radix-8 cells in the second



(a) (b) (c)

Figure 5.21 Implementation of a $0.35\mu m$ dynamic CMOS 56-bit adder. (a) micrograph, (b) radix-8 dynamic CMOS domino gate and (c) trace showing evaluation time.

row. The main carry path across each adder differs in cell length, fan-out and interconnect length. Layouts of each adder were tuned for transistor sizing, then extracted and simulated using HSPICE to determine the delay. The fabrication process used for this comparison was $0.35\mu m$ CMOS. This shows that radix-8 based adder designs provide the smallest delay for this particular technology.

Table 5.2 Comparison of new radix-8 and radix-16 64-bit adders.

Adder	Adder A	Adder B	Adder C	Adder D	Fig. 5.22	Adder E
Radix	8	8	8	16	16	16
Cells	146	132	140	120	126	124
Delay (cells)	6	5	5	6	5	5
Delay ns	1.77	1.5	1.48	1.75	1.60	1.58
Pre-charge time ns	0.8	0.8	0.8	0.8	0.8	0.8
Area mm^2	0.1	0.1	0.1	0.095	0.095	0.095
$Area \times time$	876	660	700	720	630	620
$Area \times time^2$	5256	3300	3500	4320	3150	3100

Table 5.3 shows a comparison of the new parallel prefix adder with published work including a two-stage carry lookahead adder by Ueda *et.al.* [USS+96] and a fast adder for the HP PA-RISC microprocessor [Naf96] based on Ling's work [Lin81].

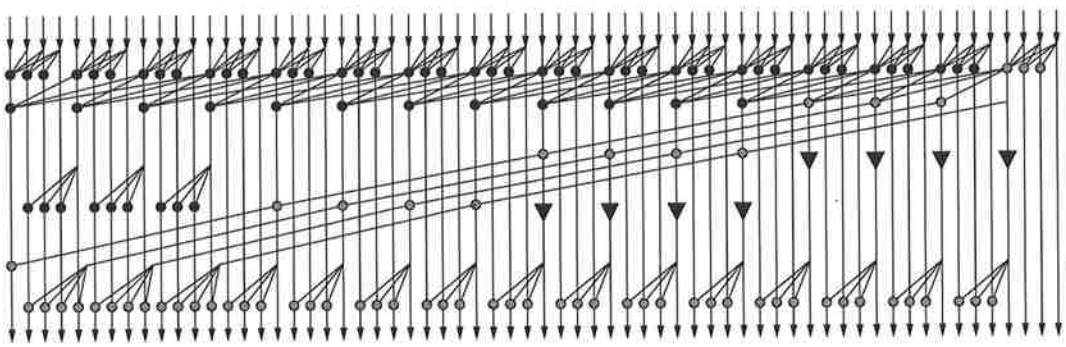


Figure 5.22 A radix-16 carry tree implementation of the new adder with five delays.

Table 5.3 Comparison of 64-bit adders.

Adder	[USS+96]	[USS+96]	[Naf96]	Fig.5.20
Technology	BiCMOS	CMOS	CMOS	CMOS
gate length (μm)	0.5	0.5	0.5	0.35
Logic	static	static	dynamic	dynamic
Radix				8
Delay (ns)	3.5	4.7	0.93	1.75
Pre-charge (ns)	-	-	1.12	0.8
Power (mW)	80	70	375	97
Area (mm^2)	0.45	0.4	0.25	0.08

The newly generated CMOS adders are significantly smaller than the published work, even when allowing for process scaling, although the cycle time is 25% slower compared to a dynamic Ling adder [Naf96], which has an effective gate length of $0.35\mu m$ and was optimised for fast evaluation (long pre-charge time).

5 End-around Carry Adders

Section 9 will demonstrate that floating-point adders that speculatively calculate two results based on exponent differences (near and far path) use a significand adder in the near path which provides the magnitude only of the result and performs no rounding. In this case, an end-around carry (EAC) adder [Tya93] can be used by effectively connecting the carry-out to carry-in. In the context of a prefix adder [Bur98], the delay is the carry propagation time across the adder plus the extra delay of the selection logic (two gate delays but a large fan-out). The focus of the prefix EAC (or flagged) adder design is to reduce the EAC operation

to a single signal (carry-out) and drive a large fan-out (carry-in) which can also be used for incrementing the result in the case of a rounding adder. An EAC adder has the property that the carry distance is bounded by the width of the adder. For example, if the carry-out was as a result of a carry generated at the i^{th} bit and propagated along the width of the adder, the carry-in may propagate at most to bit position $i-1$. Flagged prefix and dual carry chain (carry select) adders propagate the carry along twice the adder width. A parallel prefix carry tree analogous to this is shown in Figure 5.23(a). Equations 5.5 and 5.7 can be extended in equations 5.10 and 5.11 to account for the EAC propagation.

$$G_n^m = g_n + \sum_{i=m}^{n+w-1} (P_n^{<i+1>_w} . g_{<i>_w}) \quad m > n \tag{5.10}$$

$$P_n^m = \prod_{i=m}^{n+w-1} p_{<i>_w} \quad m > n \tag{5.11}$$

Example 16-bit EAC parallel prefix adders were constructed using the algorithm in the previous section and are shown in Figure 5.23. The first three EAC adders in Figure 5.23 are EAC

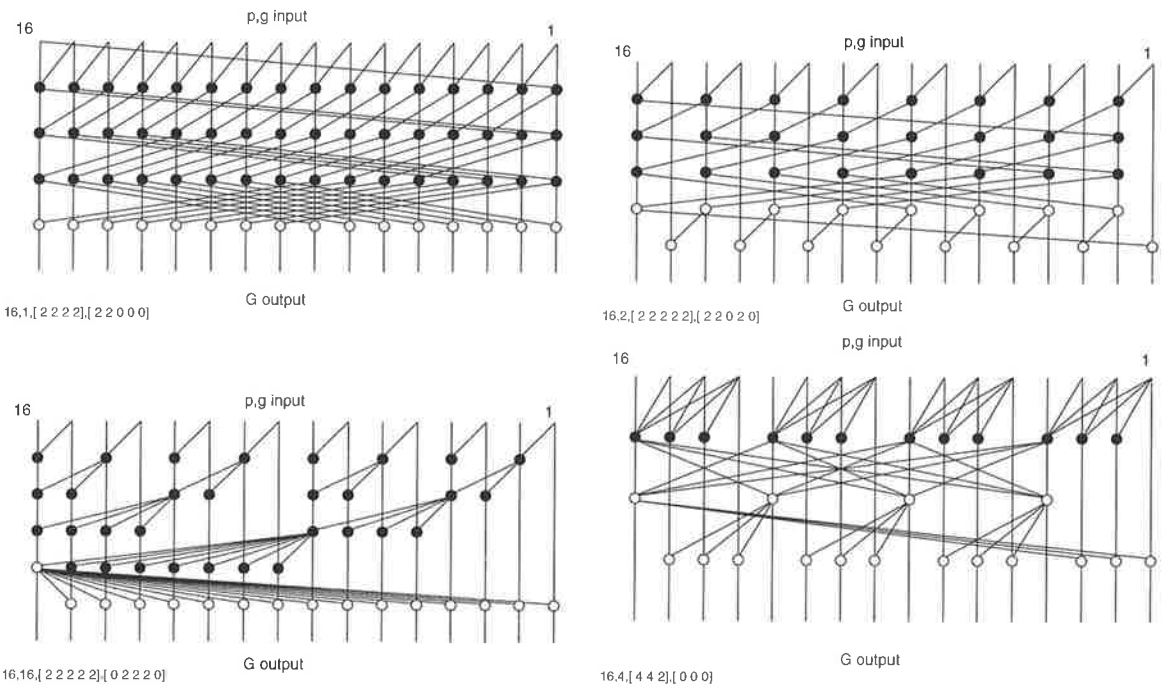


Figure 5.23 16-bit end-around carry parallel prefix adder trees.

variants of the Kogge-Stone, Han-Carlson and conditional-sum adders. The fourth (valency–4) new EAC adder is the fastest of these carry trees since it does not perform a reduction to a single carry-out signal and it has fewer cells in the critical path.

6 Introduction to Floating-Point Addition

Floating-point (FP) addition is the most frequent FP operation in the MatRISC processor and FP adders are therefore critically important components in modern microprocessors [CMB+99, HIO+97, HCB96, KWF+96, Gre95] and digital signal processors [Sim98]. FP adders must be fast to match the increasing clock rates demanded by deep sub-micron technologies with a small number of pipelining stages to minimise latency and improve branch resolution time. FP adders must also be small, particularly for use in parallel processing systems [KKA+96] with multiple FP units (FPUs).

The design of FP adders is considered more difficult than most other arithmetic units due to the relatively large number of sequentially dependent operations required for a single addition (addition is denoted here as an add *or* subtract operation) and the extra circuits to deal with special cases such as infinity arithmetic, zeros and NaNs, as demanded by the IEEE-754 standard. As a result, there is scope for investigating smaller and faster FP adders by re-organising the algorithm and using new circuit techniques.

This section discusses the design and implementation of an IEEE-754 compliant double precision three cycle floating-point adder which uses minimal hardware for use in the MatRISC processor. This is achieved by merging the rounding stage with the significand addition by using a flagged prefix adder [Bur98] which provides a plus "1" or plus "2" of the significand result for a small increase (one complex gate plus a half adder delay) in critical path length with much less hardware than two adders or one compound adder. A modified version of this FP adder that supports two cycle latency accumulation by overlapping the alignment and normalisation shifts into the first pipeline stage is then presented. A further algorithm is proposed to perform FP addition in two cycles which can match the latency of FP multipliers. Providing matched latencies of multipliers, adders and accumulators in multiple FPUs in a microprocessor simplifies the pipeline scheduling and improves branch resolution times. This new FP adder also incorporates a fast prediction scheme for the true subtraction of significands with an exponent difference of "1", with one less adder.

The work presented in this section was reported in [BSBLL99].

6.1 Background

Traditional methods for performing FP addition can be found in Omondi [Omo94] and Goldberg [Gol90], who describe algorithms based on the sequence of significand operations: swap, shift, add, normalise and round. They also discuss how to construct faster FP adders. Implementations of FP adders are reported in [HIO+97, HCB96, KWF+96, Gre95, Kno91, MHR90, KF89]. Algorithms and circuits which have been used to improve adder design are described in [Okl94, HM90, Far81, PSG87, OATF97, QF91, NMLE97, QTF91, PLK+96].

Some of these improvements are as follows:

- the serial computations such as additions can be reduced by placing extra adders (or parts thereof) in parallel to compute speculative results for exponent subtraction ($E_a - E_b$ and $E_b - E_a$) and rounding ($M_a + M_b$ and $M_a + M_b + 1$) then selecting the correct result (eg. T9000 [Kno91]).
- the calculation of the exponent differences ($E_a - E_b$ and $E_b - E_a$) can be done in parallel with the significand alignment shift by using a binary weighted alignment shifter.
- the position of where the round bit is to be added when performing a true addition depends on whether the significand result overflows, so speculative computation of the two cases may be conducted [KF89].
- calculation of the normalisation distance can be carried out by a leading zero anticipator to provide the normalisation shift to within one bit, in parallel with the significand addition [HM90].
- a fast integer adder is crucial to the design of FP adders for calculating the result significand and sets the minimum cycle time [Naf96].

Further improvements in speed can be made by splitting the algorithm into two parallel datapaths based on the exponent difference [Gre95, KF89, Far81, OATF97, NMLE97], namely near ($|E_a - E_b| \leq 1$) and far ($|E_a - E_b| > 1$) computations, by noting that the alignment and normalisation phases are disjoint operations for large shifts. However there is a significant increase in hardware cost since the significand addition hardware cannot be shared as is the case with the traditional three stage pipeline.

Other FP adder designs have moved the rounding stage before the normalisation [PLK+96, KWF+96, QF91]. Quach and Flynn describe a FP adder [QF91] which uses a compound significand adder with two outputs plus a row of half adders and effectively has a duplicated carry chain. Kowaleski *et al.* [KWF+96] describe an adder as part of a 263,000 transistor FP

unit which contains separate add and multiply pipelines with a latency of 4-cycles at $433MHz$ in a $0.35\mu m$ process. The significands are simultaneously added and rounded by employing a half adder which makes available a LSB for the case where a "1" is added for taking the two's complement of one input. Decode logic on two LSBs of both operands calculates if there will be a carry out of this section as a result of rounding, either by adding one or two. A circuit is used to determine if the MSB is "1" as a result of adding the significands in which case the round bit is added to the second LSB to compensate for the subsequent normalisation by 1-bit to the right. The significands are added by using a combination of carry look-ahead and carry select logic. Precomputed signals predict how far the rounding carry will propagate into the adder. The MSB computed before rounding is used to select the bitwise sums for the result.

Nielsen *et al.* describe a redundant-add FP adder [NMLE97] with addition and rounding separated by normalisation. By using redundant arithmetic, the increment for the rounding is not costly. A variable latency architecture [OATF97] has been proposed which results in a one, two, or three clock cycle data dependent addition. However, this implies that the host system can take advantage of up to three simultaneously emerging results, which represents a considerable scheduling difficulty.

FP adders that have an accumulation mode or can by-pass incomplete results are of interest in vector and deeply pipelined machines. Hagihara *et al.* [HIO+97] report a $0.35\mu m$, $125MHz$ FPU with a three cycle FP adder which adds three operands simultaneously, so two new operands can be accumulated per cycle, saving 25% of the accumulation execution time in a vector pipelined processor for use in supercomputers. Heikes and Colon-Bonet [HCB96] report a FMAC architecture incorporating a floating-point adder path with 4-cycle latency at $250MHz$ in a $0.5\mu m$ CMOS process employing dual-rail domino logic. The rounding is delayed and can produce a bypassable unrounded result in three cycles.

6.2 IEEE-754 double precision floating-point format

Throughout the rest of this chapter, all operands are assumed to be 64-bit IEEE-754 double precision format floating-point numbers [Soc85] with a 52-bit fractional significand ($M = s_{51}s_{50}\ldots s_0$), an 11-bit biased exponent ($E = e_{10}e_9\ldots e_0$) and a 1-bit sign (S).

A normalised floating-point number, N is represented by:

$$N = S \times R^E = -1^S \times 1.s_{51}s_{50}\ldots s_0 \times 2^{e_{10}e_9\ldots e_0 - bias}$$

where E is non-zero and less than 11111111111 and the radix, R is 2 for digital computers. For the case where the biased exponent E is zero the number may be a signed zero ($M = 0$)

or a subnormal number ($M \neq 0$). Infinities and not-a-number (NaN) are defined for $E =$ 11111111111.

For all examples in this section, the floating-point numbers A and B have significands M_a and M_b, exponents E_a and E_b and signs S_a and S_b respectively. The significand is in the range $1.0 \leq M < 2.0$ for normalised numbers. The prefix *true* is used to describe the underlying addition and subtraction process of the significand adder, i.e. (1.0) minus (−1.5) is a true addition.

7 A Three Cycle Floating-Point Adder using a Flagged Prefix Integer Adder

A three cycle IEEE-754 double precision FP adder is shown in Figure 5.24. The control signals have been omitted for clarity and the pipelining stages are indicated by horizontal bars. This adder accepts normal, subnormal, zero, NaN and infinity IEEE compliant floating-point numbers, an add/subtract signal and one of the four rounding modes as input. It can handle all exceptions, NaNs, infinities and zeros as defined in the standard.

It is similar in design to the traditional three stage pipeline adder where the first stage is an exponent subtraction, significand swap, unpacking and alignment shift, the second stage performs the significand addition and the third stage performs the post normalisation shift. Rounding of the significand, which is normally done after the normalisation shift, is merged with the significand addition in the second pipeline stage using a flagged prefix adder.

7.1 Unpacking and alignment

To align the significands, the exponents are compared to determine which number may be the smallest and moving its significand into the M_b data path through the swapper. Both numbers are packed with the hidden bit and M_b is then right shifted in a barrel shifter by the difference of the two exponents. If the exponents are equal, then no swap is performed and the positive result (can be either $M_a - M_b$ or $M_b - M_a$) is selected after significand addition in the second stage.

To speed up the operation of the first stage, two adders subtract 54 from the twin exponent differences to provide a valid signal for the barrel shifter if the shift is within range. The critical path for the first stage is through the exponent subtracters, alignment decoder and

the select circuits in the shifter.

7.2 Significand addition and rounding

In the case of true subtraction, the smaller number can be negated using two's complement form and added using a fast carry propagate adder to produce the magnitude of the result. Finding the smallest significand can be very costly if the exponents are equal, so one technique to speed up the operation employs two adders to calculate $M_a - M_b$ and $M_b - M_a$ in parallel and select the positive result. Another similar method is to calculate $M_a + \overline{M_b} + 1$ $(M_a - M_b)$ and $M_a + \overline{M_b}$ which can be negated to give $M_b - M_a$. The same circuit can be used to perform rounding for true addition. Both methods generally use two adders in parallel.

There are four implemented rounding modes, round toward zero, nearest and plus/minus infinity. Rounding of the result to 53-bits must be carried out as if the result was exact (to infinite precision) and then the rounding rules applied to determine if the significand of the result should be truncated or incremented. The 53-bit result, pre-shift guard, round and sticky bits are to be passed to the rounder circuit. The final guard, round and sticky bits are determined from the post-shift amount. The guard bit is the bit below the LSB of the 53-bit result and the round bit is to the right of the guard bit. The pre-shift sticky bit is obtained by OR'ing the pre-shifted significand bits below the pre-shift round bit.

The true subtraction operation requires that a "1" be added into the LSB of the smallest number for the two's complement operation of M_b since the positive result is needed. To produce an exact representation of the result, the "1" must be added into a bit infinitely below the number so that the "1" propagates through the infinite string of trailing zeroes up to the pre-shifted LSB. The only case where the "1" can propagate to the round bit is when all of the pre-shifted bits of M_b below the round bit are zero (which are then inverted). So "1" is added into the round bit if the pre-shift sticky bit is zero for true-subtraction.

True addition can either:

- produce a significand result greater than or equal to one and less than two and the pre-shift guard, pre-shift round and pre-shift sticky bits become the guard, round and sticky bits for the rounding circuit, or
- cause an overflow to occur if the result is greater than or equal to two and less than four, in which case the LSB becomes the guard bit, the pre-shift guard becomes the round bit, the pre-shift round and pre-shift sticky bits are OR'd to form the sticky bit for the rounding circuit.

True subtraction can produce (assuming $A \geq B$):

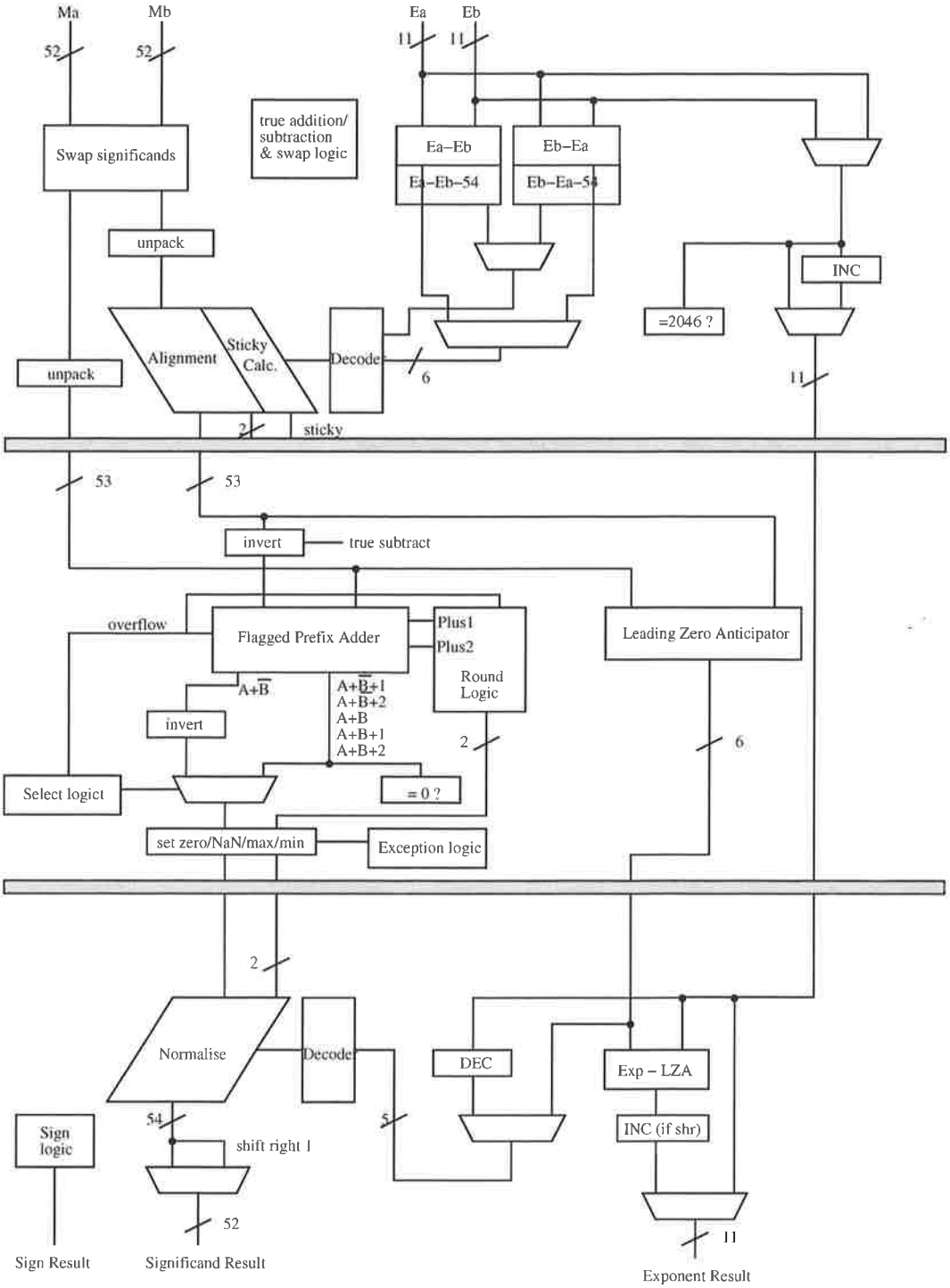


Figure 5.24 A three-cycle floating-point adder.

- $E_a = E_b$: no rounding is possible, result is exact
- $E_a - E_b = 1$: no rounding is carried out if the normalisation shift is ≥ 1 since the round and sticky bits must be zero. If the normalisation shift is zero then rounding is performed (only the guard bit, which has become the round bit, is used)
- $E_a - E_b \geq 2$: the normalisation shift will be at most one, and rounding is needed.

The rounding circuit drives the control signals for the flagged prefix adder, described below, to instantly change the significand result by plus "1" (for a true addition round increment or a true subtraction with no increment) or plus "2" (if the true addition result had overflowed or true subtraction and a round increment occurred).

It is possible for the rounding incrementer to produce an overflow of the significand and possibly the exponent. This is detected and corrected with a shift right in the normalisation phase and uses the same multiplexer to do this as the correction for the LZA.

7.3 A flagged prefix adder for merged significand addition/rounding

In this section only, A and B are referred to as significands. To merge the rounding process with the significand addition, the following results need to be computed: $A+B$, $A+B+1$, $A+B+2$, $A-B$, $A-B+1$, $B-A$. To compute this (in two's complement form), the actual calculations are: $A+B$, $A+B+1$, $A+B+2$, $A+\overline{B}+1$, $A+\overline{B}+2$, $\overline{A+\overline{B}}$. Burgess proposed a flagged-prefix adder [Bur98] which computes $A+B$, $A+B+1$, $-(A+B+2)$, $-(A+B+1)$, $A-B-1$, $A-B$, $B-A-1$ and $B-A$ for a minor increase in delay from a single addition (1 complex gate plus a half adder). A flagged prefix adder shown in Figure 5.25(a) is a modified parallel prefix-type carry look-ahead adder which calculates the longest string of one's (up to and including the first "0") from the second LSB upward and sets them as "flag bits" to indicate those bits which can be inverted for incrementing a two's complement number. The overhead is approximately 1 extra complex gate plus a half adder per bit over a prefix type adder. By way of comparison, a compound adder replaces each prefix cell by a pair of multiplexers and has an extra row of output multiplexers. Therefore flagged prefix adders use approximately 70% of the logic of a compound adder.

A subset of the available functions in a flagged prefix adder was used which simplified the flagged inversion cells shown in Figure 5.25(a)(b). The 2 least significant flagged inversion cells were replaced with flag logic which provides a plus "2" function in addition to the standard plus "1" function.

The flag bits start from the second LSB and must detect the following strings of bit propagate

(p), bit generate (g) and bit kill (k) signals to facilitate a "plus 2" operation:

$$....ppppppp, ...pppppk,$$

$$....pppppkg, ...ppppkgg, etc.$$

To simplify the detection of these bit strings a row of half adders is placed before the prefix adder to convert the "$kgg...$" conditions to strings of bit propagate signals. The flag bits are then defined as $f_i = f_{i-1}.p_{i-1} = P^0_{i-1}$, the $i - 1^{th}$ group propagate signal available from the prefix tree.

The control signals, *plus1* and *plus2* for the flagged prefix adder are generated from the rounding mode, sticky bit, result LSB, round bits and the true addition/subtraction signals. These control signals together with the bit-propagate and bit-generate signals ($p0$, $g0$, $p1$, $g1$) and the overflow signal are used to drive the *inc* signal for the flagged inversion cells.

In a true addition, $A + B$ needs to be calculated. If the result of the rounding circuit is an increment then *plus1* is set. If $A + B$ overflows, then adding a "1" becomes adding "2" to the un-shifted significand, the LSB becomes the round bit and *plus2* is set. If the rounded result ($A + B + 2$ or $A + B + 1$) produces an overflow then this is shifted right one place and truncated.

For the case of true subtraction the *plus1* signal is automatically set (for the negation of the second operand). If the result also needs to be incremented due to rounding, then the *plus2* signal is set. We could have added the compulsory "1" as a carry input to the LSB of the adder but we may need $\overline{A + \overline{B}}$ (actually $B - A + 1$) if $A + \overline{B} + 1$ overflows and the requirement for $A + B + 2$ for true addition means that all operations can be performed with a single flagged prefix adder.

7.4 Normalisation

The normalisation stage left shifts the significand to obtain a leading "1" in the MSB, and adjusts the exponent by the normalisation distance. For true subtraction, the position of the leading "1" of the significand result is predicted from the input significands to within 1-bit using a LZA [HM90, Okl94] which is calculated concurrently with the significand addition. The normalisation distance is passed to a normalisation barrel shifter and subtracted from the exponent. The LZA circuit is arranged so the error is always larger than the required shift and correction circuits right shift the significand and add "1" to the exponent if the normalisation distance is too large. For true addition, the only possible post-shifting that can be performed is a single shift right if the significand addition and rounding operation produced a supernormal result. The multiplexer to perform this is shared with the LZA

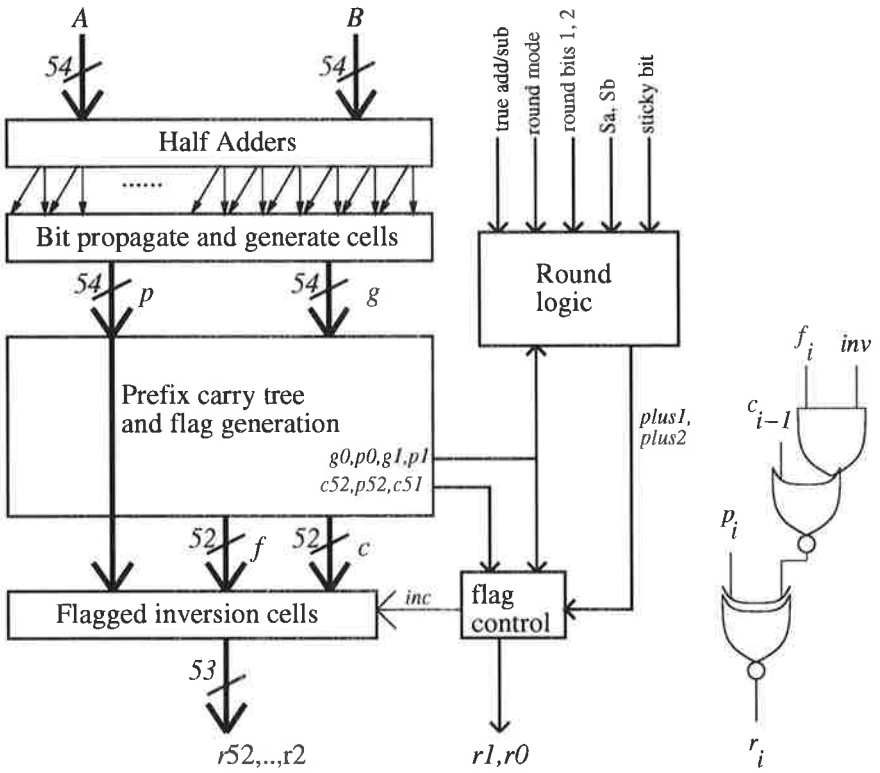


Figure 5.25 Flagged prefix adder. (a) for merged significand addition and rounding and (b) the flag inversion cell.

correction function. The fractional significand can now be split off from the implied "1" and packed with the exponent and sign bit to produce the 64-bit result.

7.5 Implementation

The three cycle adder described in the previous section was implemented as a test chip for the MatRISC processor. This is a minimal hardware implementation since the only speculative computation is to calculate both exponent differences in the first stage. In the first stage, a pass transistor type shifter was used which consumes less area than a binary weighted multiplexer.

A micrograph of the fabricated $0.5\mu m$ FP adder chip described above is shown in Figure 5.26 and the main functional blocks are highlighted. Most of the FP adder was constructed using static CMOS logic and it contains 33,000 transistors. It was full custom designed using VHDL for high-level design and verification with a suite of C test routines. The adder was functionally verified against the Berkeley IEEE-754 test vector set and sets of directed and

random vectors generated by a golden device (SUN UltraSPARC 60 [Gre95]). A graphical user interface was developed for a PC and the adder was controlled through the parallel port to allow interactive testing.

FP numbers are loaded into the FP adder through boundary scan cells and tap points along the critical path are bought out to pads. The worst case delay of the flagged prefix adder is $6ns$. The speed of the FP adder chip was tested up to $100MHz$ (the limit of the HP16500c digital test system).

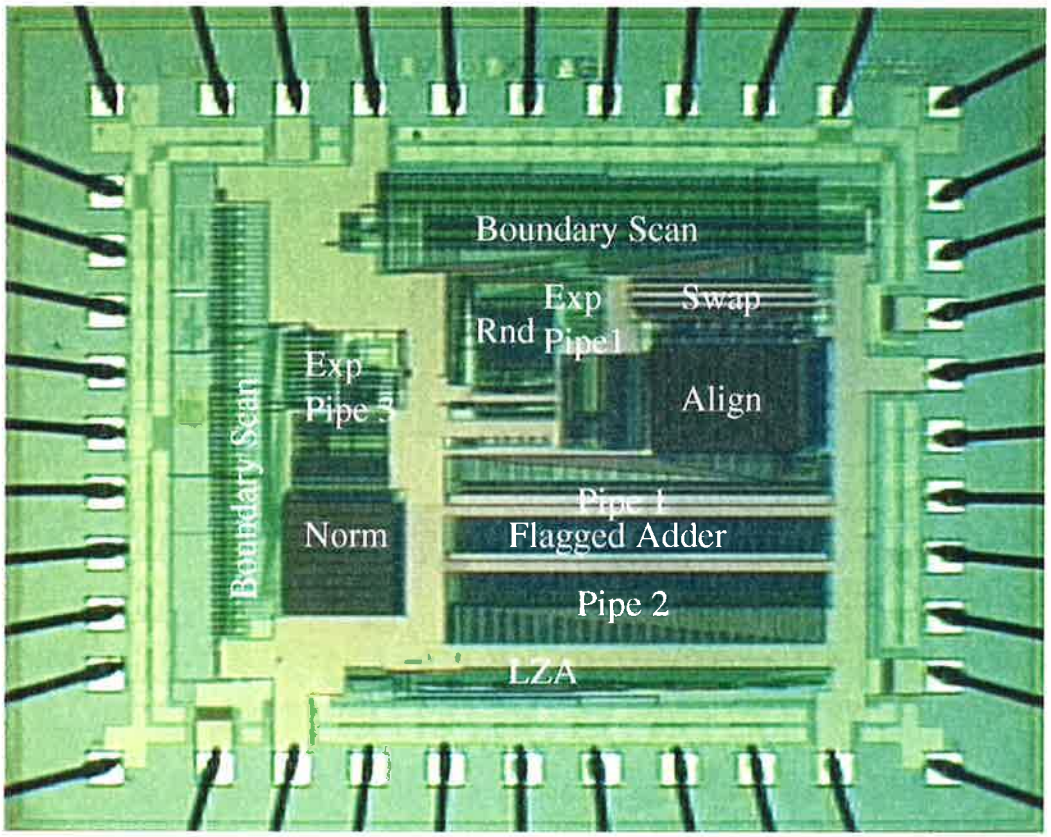


Figure 5.26 Micrograph of the $0.5\mu m$ CMOS FP adder chip.

8 A Two Cycle Accumulator Architecture

A new FP adder which supports two cycle accumulation whilst still being able to perform three cycle pipelined additions is shown in Figure 5.27. A correctly rounded, possibly un-normalised result is produced at the output of the second stage of the FP adder described in the previous section, together with the normalisation distance (which may be in error by 1-bit). This number may be either normalised in the third stage of an adder or accumulated with the next but one input using only a small amount of extra hardware (four multiplexers). This is useful in processors where many short accumulations are performed as it can reduce the pipeline start-up and finishing cost.

In the accumulation mode, the normalisation of the accumulator and the alignment shift operations can be overlapped so the critical path does not pass through both shifters. The possibly un-normalised accumulator significand, M_u is fed back to the start of the adder along with its exponent, E_u, sign, S_u and a normalisation distance, E_{dist}. There are three cases to be considered to align the accumulator and addend significands:

- $E_a \geq E_u$: the normalisation distance, E_{dist} can be ignored and M_u can be shifted further to the right by the exponent difference $(E_a - E_u)$. The M_u significand bypasses the normalise shifter through the bypass-1 multiplexer.

- $E_a < E_u < E_a + E_{dist}$: the M_u significand is shifted left by the exponent difference $(E_u - E_a)$ but not enough to make it normalised. It then bypasses the alignment shifter through the bypass-2 multiplexer.

- $E_u \geq E_a + E_{dist}$: in this case, both significands need to be shifted. M_u must be normalised and M_a must be right shifted by the exponent difference less the normalisation distance $(E_u - E_a - E_{dist})$. E_{dist} can be in error by 1-bit which can be detected by the multiplexer directly after the normalisation shifter and is applied to the bypass-2 multiplexer to correct the shifted significand.

The length of the critical path has increased by an 11-bit adder delay. However, speculative calculations for $E_a - E_b$, $E_b - E_a$, $E_a - E_u$ and $E_u - E_{dist} - E_a$ and multiplexing the result would only increase the critical path by two multiplexer delays.

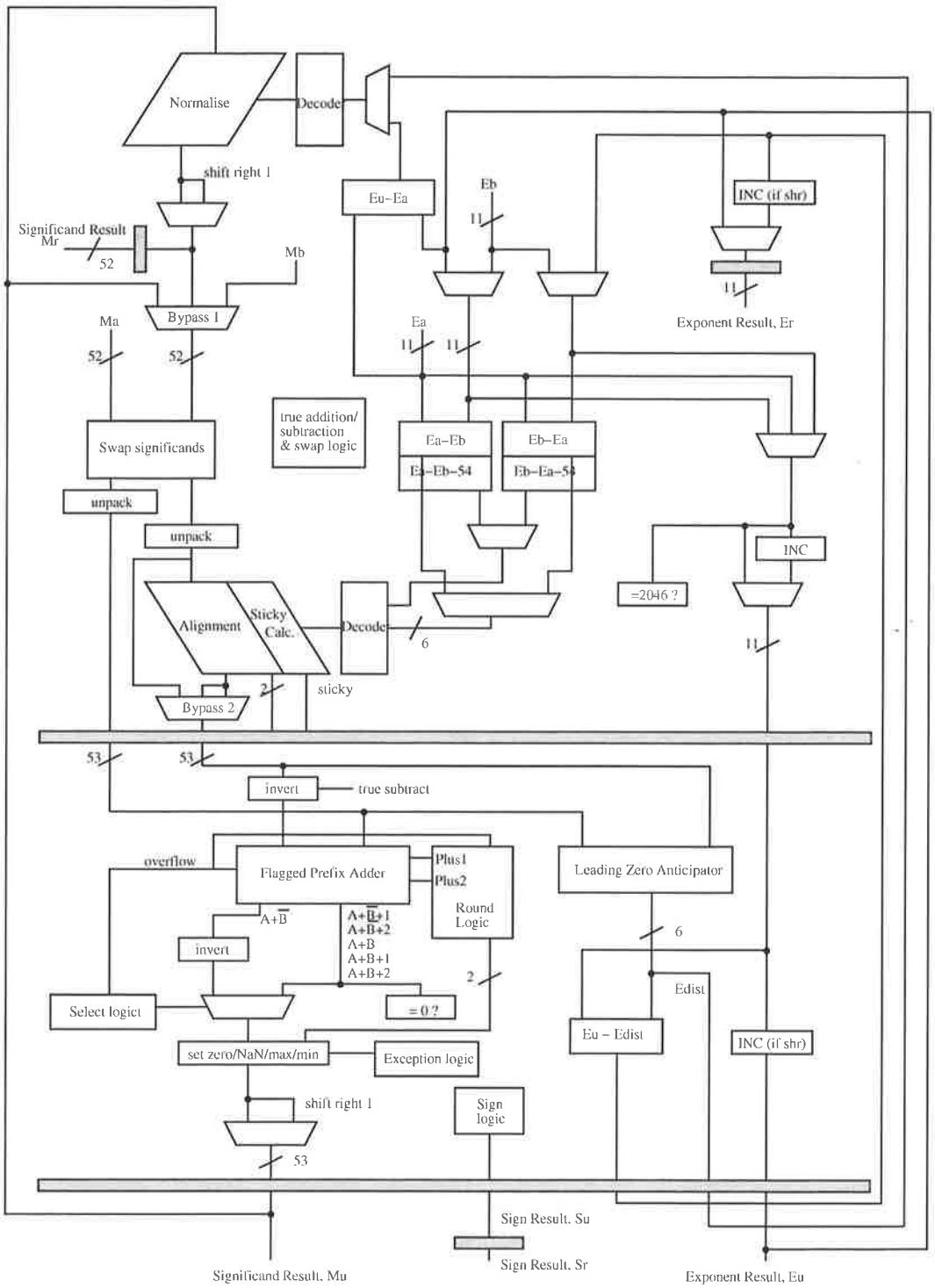


Figure 5.27 A three cycle FP adder which can be used as a two cycle accumulator.

In non-accumulation mode, the adder has the same architecture and three cycle latency as the FP adder in the previous section. The post-normalisation step is independent of the in-coming operands and the result is taken from M_r.

9 A Two Cycle Floating-Point Adder Architecture

In the previous section, an FP adder was designed which merged rounding with the significand addition phase. A two cycle FP adder can be constructed which has separate data-paths depending on whether or not the data requires a large alignment shift or large normalisation shift as shown in Figure 5.28. This is a speculative FP adder architecture and uses more hardware than the previous design. A multiplexer at the output of the near and far data-paths selects the correct result at the completion of the second pipeline stage. As noted earlier, some recent microprocessors have used this method to speed up FP addition [KWF+96, Gre95].

The far data-path calculates the result for all true addition operations and true subtraction operations which have an exponent difference greater than or equal to 2. In this case true subtraction of the significands leads to a result which is always greater than one half and at most needs to be left shifted by one bit, whereas a true addition can only lead to the significand that is shifted one place to the right if there is an overflow. This is carried out with multiplexers at the output of the far pipe-2 flagged prefix adder. The far data-path also computes results for some cases where the exponent difference is "1" which are discussed below.

The near data-path computes the true subtraction cases where the exponent difference is "0" and "1" in some cases. These computations may need a large number of result significand left shifts for normalisation due to the massive cancellation of the significands. The result of the near data-path is calculated speculatively as it is not known until well into the first pipeline stage if the result will be needed.

The near pipe-1 stage uses a carry propagate adder to add the significands which forms the critical path. To start the addition as early as possible, thereby minimising the cycle time, it is only necessary to check if the exponent LSBs differ to speculate if the exponents differ by "1" and one of the significands is right shifted by 1-bit. If the exponent LSBs are equal and it turns out at the end of far pipe-1 that the exponents are equal, the significand result is exact, requires no rounding, and a normalisation shift then occurs in near pipe-2.

If the exponent LSBs are different, it then needs to be determined which of the significands is to be right shifted by one bit. A speculative regime could calculate both $M_a - M_b/2$ and

$M_b - M_a/2$ using two significand adders and multiplex the correct result once the exponent difference is known, which has been calculated in parallel with the significands.

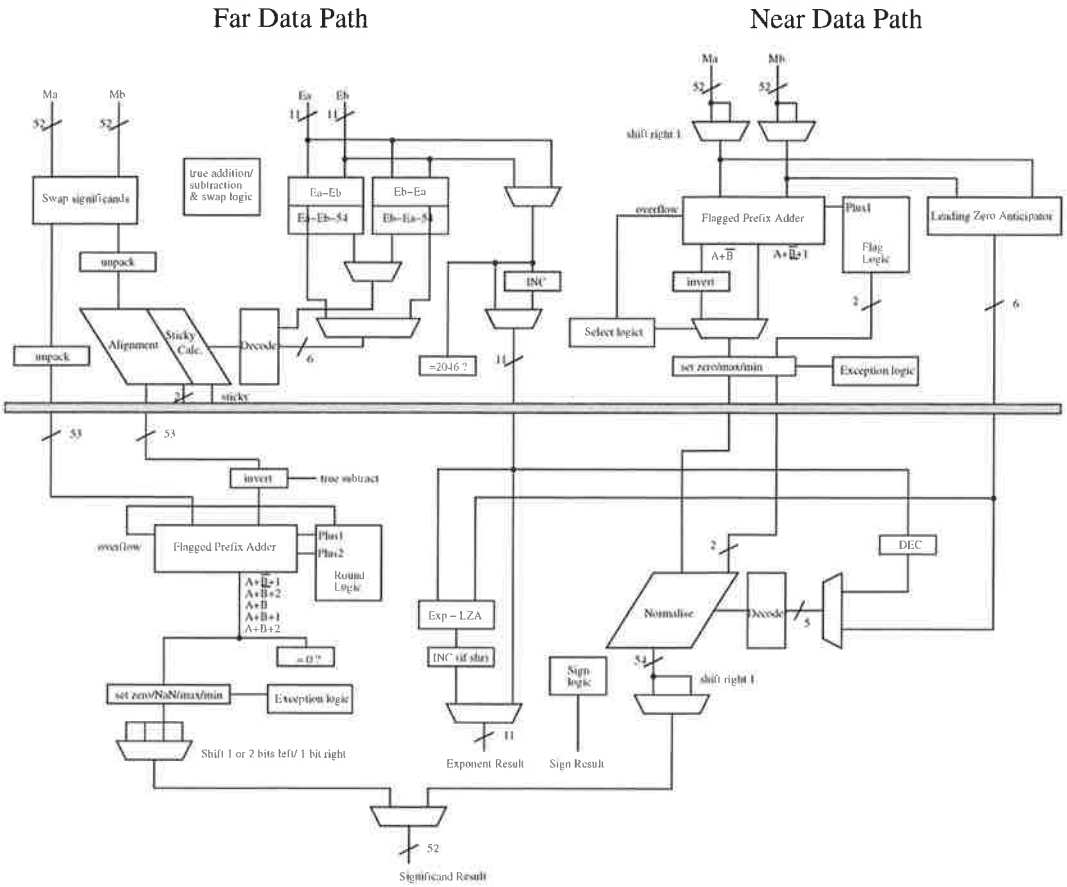


Figure 5.28 A two cycle floating-point adder.

A new method is proposed to determine which of the significands should be right shifted by one bit, thereby eliminating one of the adders and the multiplexer.

If the MSBs of the significands (before unpacking) are inspected, there are four cases to consider which depend on the sign of the exponent difference as shown in Table 5.4. Only one of the cases where the exponent bits differ and massive cancellation of leading result bits can occur requires a large normalisation shift. The other case where the exponent bits differ (eg. "10" for $E_a - E_b = 1$) and the "11" case result in at most 1-bit left shift and the "00" case results in at most 2-bits left shift. The cases where the MSBs are equal ("00" and "11") can be handled by the far data-path providing the normalising multiplexer in far pipe-2 is modified to perform a 2-bit left shift. If the MSBs are not equal ("01" and "10"), only one of these two cases can lead to large normalisation shifts, so speculatively calculate $M_a - M_b/2$

Table 5.4 Normalisation shifts needed for an exponent difference of one.

Sig.MSBs $M_a(51)$, $M_b(51)$	$E_a - E_b = 1$		$E_a - E_b = -1$	
	Range (greater,less)	Norm. Shift	Range (greater,less)	Norm. Shift
00	0.25, 1	2	0.25, 1	2
01	0, 0.75	54	0.75, 1.5	1
10	0.75, 1.5	1	0, 0.75	54
11	0.5, 1.25	1	0.5, 1.25	1

when M_a has a "0" in the MSB and M_b has an MSB equal to "1", or calculate $M_b - M_a/2$ when M_a has a "1" in the MSB and M_b has a "0" in the MSB. The operand with a "1" in the MSB is always right shifted and the case where the MSB bits are reversed is dealt with in the far data-path. These results are always exact requiring no rounding thus simplifying the near path flagged prefix adder. Choosing which data-path to use is summarised in Table 5.5.

A circuit to determine which operand to shift can be built with one CMOS complex gate and the need for an 11-bit exponent subtraction to determine the sign of the exponent difference has been avoided. The only extra hardware needed for the two cycle FP adder compared to the one shown in Figure 5.24 is an extra flagged prefix adder. Further speed improvements can be made in far pipe-1 by duplicating the alignment shifter and computing the two exponent differences in parallel with the alignment shifts [Kno91].

Table 5.5 Significand shift function.

Exp.LSBs A_{e0} B_{e0}	Sig.MSBs $A_m(51)$, $B_m(51)$	near/far data-path	near data-path alignment shift
same (00, 11)	-	near	no shift
differ (10, 01)	00	far	-
"	01	near	B_m right 1-bit
"	10	near	A_m right 1-bit
"	11	far	-

10 Discussion

In this chapter, novel integer parallel prefix adders for carry-propagate, end-around carry and flagged prefix structures have been generated and optimised for radix and take into account the increasingly important effects of wire loading and fan-out on deep sub-micron CMOS VLSI circuits. An algorithm was written which can generate a complete family of parallel prefix carry trees and can be used to assess designs under a particular technology constraint. A new dynamic parallel prefix adder circuit was implemented in a $0.35\mu m$ CMOS.

Implementations of floating-point adders were reviewed. The parallel prefix adder designs were applied to the design of floating-point adders which are used in the MatRISC processor. In particular, the flagged prefix adder provides a fast increment of the significand addition for use with rounding or end-around carry propagation, thereby merging the rounding with significand addition without the need for two significand adders or duplicated carry trees.

A minimal hardware three cycle floating-point adder was designed, implemented in a $0.5\mu m$ CMOS process with an area of $1.8mm^2$ and comprehensively tested. This adder architecture was improved with small additional hardware to include a two cycle accumulation mode by feeding back the previous un-normalised but correctly rounded result together with the normalisation distance.

A faster two cycle speculative adder which uses more hardware was developed based on the normalisation shift distance of the significand result. It also incorporated a fast prediction scheme for the true subtraction of significands with an exponent difference of one, with one less adder.

Chapter 6

Algorithms

THE mapping and performance of several signal processing and computational routines onto the MatRISC-1 processor are presented in this chapter.

Volterra models can represent nonlinear distortion with memory, and its inverse can be used to reduce the distortion by nonlinear compensation. However, the associated computational complexity makes conventional implementations impractical. In this chapter, a matrix representation for a second order Volterra system is presented. Higher order Volterra systems are represented in terms of the second order matrix-based representation which allows efficient implementation and a simpler update process than previously published structures. Real-time implementations using the MatRISC processor are then proposed. Performance estimates are given for various processor cycle times and interprocessor bandwidths. The Fourier transform and Kalman filter are also studied.

The execution time and compute rate of matrix multiplication and element-wise operations (addition) was derived in Chapter 2 and is shown in Figures 2.27 and 2.28 respectively. This assumes a MatRISC processor with parameters: $p = 4$, $V = 4$, $nDP = 4$ and $T_a = 10ns$ for equal sized matrices. The PE used for the performance simulations in this chapter has four multipliers and four adders in each PE with a peak theoretical performance of 1.6 $GFLOPS$ and an I/O bandwidth of 800 $Moperands/s$. The 16 processor MatRISC module has a peak theoretical compute rate of 25.6 $GFLOPS$. This work in this chapter has been reported in [TLBS+99, BSTM+97]. Other related work in the area of nonlinear compensation done by the author is reported in [BSTLM01, BSLTM99, BSTL+99].

1 Volterra Model

Nonlinear distortion can limit the performance of communications systems. Various nonlinear signal processing techniques can be used to model and compensate nonlinear distortion. The Volterra model [Sch89] uses a set of functionals and kernels to model a wide class of nonlinear systems with memory. A nonlinear compensation method is the use of a Volterra inverse to apply pre- or post-distortion. For example, the Volterra model has been used to represent and compensate nonlinearities in the analogue stages of radio communications systems [LPS94], [Dal89] and nonlinear communications channels in [BBD79]-[BBC88]. Compensating timing jitter nonlinearities typical of samplers and analogue to digital converters is shown in [TL93], [Tsi95] and loudspeaker compensation is demonstrated in [CKHP90], [Fra94]. The Volterra representation has also been used to model biological systems [MM78] and ocean-wave/offshore-platform interactions [RPA87], [KP79].

The discrete-time Q^{th} order Volterra model with memory length M for all orders can be written as:

$$\begin{aligned} y_{(Q)} &= H_{(Q)}[x(n)] \\ &= H_0 + H_1[x(n)] + ... + H_q[x(n)] + ... + H_Q[x(n)] \end{aligned} \tag{6.1}$$

where $H_0 = h_0$ is the DC term, $H_q[.]$ is the q^{th} order Volterra operator given by (6.2), and is the q^{th} order Volterra kernel.

$$H_q[x(n)] = \sum_{m_1=0}^{M-1} ... \sum_{m_q=m_{q-1}}^{M-1} h_q(m_1, ..., m_q)x(n-m_1)...x(n-m_q) \tag{6.2}$$

Vector representations of Volterra systems have been considered in the literature [NV95]. Given the first order delayed input vector $x_1^T = [x(n), x(n-1), ..., x(n-M+1)]$, the higher order delayed input product vectors are obtained using Tensor products:

$$x_2 = x_1 \otimes x_1 \tag{6.3}$$

$$x_3 = x_2 \otimes x_1 \tag{6.4}$$

$$\vdots$$

$$x_q = x_{q-1} \otimes x_1 \tag{6.5}$$

where $\otimes$ represents a tensor product operation. The resulting elements of the product vector are weighted by a Volterra kernel vector. Given a single order kernel vector:

$$H_q^T = [h_{00...0}, \quad h_{00...1}, \quad \ldots \quad , H_{MM...M}] \tag{6.6}$$

The q^{th} order Volterra output component is obtained by multiplying the input product vector with the Volterra kernel vector:

$$Y_q = H_q^T X_q \tag{6.7}$$

While this form may seem the most obvious way of representing the Volterra model components, it may not necessarily be best for fast implementations. The computational complexity of the Volterra representation increases rapidly with nonlinear order and memory [TL96]. The high computational complexity places much emphasis on the development of fast kernel estimation algorithms and reduced implementations [Fra94], [NV95], [Mer94]. Implementation using conventional processors is often impractical. For real-time implementation, parallel processing techniques need to be considered. A new representation based on first order matrices of the input samples and Volterra kernel matrices is presented which are better suited to real-time implementation. The work in [Mer94] developed a decomposition of Volterra systems into lower order terms. Using this approach, matrix structures for the higher order homogeneous Volterra model components up to fifth order in terms of the second order matrix structure are presented. The first order case is not considered since this is a standard finite impulse response (FIR) filter which has a relatively low computational complexity. The matrix representation is used as the basis for evaluating real-time Volterra system implementations on a MatRISC processor. The objective is to study these implementations for modelling and compensating for undesired nonlinear distortion.

1.1 Matrix formulation of a Volterra model

A second order Volterra representation is derived based on a Volterra kernel matrix and two first order input vectors (a matrix-vector structure). This representation is based on the first order input vector $X_{1(vect)} = [x(n), x(n-1), ..., x(n-M+1)]$ rather than the delayed input product vector, X_2^T. The second order Volterra kernel matrix (with symmetric kernels and

without redundancy) is represented by:

$$H_2 = \begin{bmatrix} h_{00} & h_{01} & h_{02} & h_{03} & h_{04} & ... & h_{0M-1} \\ 0 & h_{11} & h_{12} & h_{13} & h_{14} & ... & h_{1M-1} \\ 0 & 0 & h_{22} & h_{23} & h_{24} & ... & h_{2M-1} \\ 0 & 0 & 0 & h_{33} & h_{34} & ... & h_{3M-1} \\ 0 & 0 & 0 & 0 & h_{44} & ... & h_{4M-1} \\ . & . & . & . & . & ... & . \\ . & . & . & . & . & ... & . \\ . & . & . & . & . & ... & . \\ 0 & 0 & 0 & 0 & 0 & ... & h_{M-1M-1} \end{bmatrix} \tag{6.8}$$

The output is given by:

$$Y_2 = X_{1(vect)}^T H_2 X_{1(vect)} \tag{6.9}$$

This can be extended to a matrix-matrix structure by processing N samples as a block and considering a Toeplitz matrix of input samples:

$$X_1 = \begin{bmatrix} x(n) & x(n+1) & ... & x(n+N-1) \\ x(n-1) & x(n) & ... & x(n+N-2) \\ x(n-2) & x(n-1) & ... & x(n+n-3) \\ x(n-3) & x(n-2) & ... & x(n+n-4) \\ . & . & ... & . \\ . & . & ... & . \\ . & . & ... & . \\ x(n-M+1) & . & ... & x(n+N-M) \end{bmatrix} \tag{6.10}$$

The output vector Y_2 is given by:

$$\begin{aligned} Y_2 &= [y_2(n), y_2(n+1), ..., y_2(n+N-1)] \\ &= VectDiag[X_1^T H_2 X_1] \end{aligned} \tag{6.11}$$

where *VectDiag* generates a vector of the diagonal elements of a matrix. This matrix-matrix structure does not require higher order input product vectors or matrices, making implementation simpler. Updating the input matrices is trivial compared to updating a higher order input product vector, as in the case of previously published structures. The matrix-matrix structure in (6.11) is the basis for the implementation of the Volterra model components in this chapter.

1.1.1 Higher order Volterra models

Higher order Volterra models can be decomposed in terms of the second order representation. For example, a matrix-vector representation of a third order system decomposed into second

order terms is given. The third order discrete time Volterra component is given by:

$$y_3(n) = H_3[x(n)] = \sum_{m_1=0}^{M-1} \sum_{m_2=m_1}^{M-1} \sum_{m_3=m_2}^{M-1} h_3(m_1, m_2, m_3) \\ x(n-m_1)x(n-m_2)x(n-m_3) \tag{6.12}$$

This can be decomposed in terms of M second order terms:

$$y_3(n) = \sum_{m_1=0}^{M-1} \left[\sum_{m_2=m_1}^{M-1} \sum_{m_3=m_2}^{M-1} h_3(m_1, m_2, m_3)x(n-m_2)x(n-m_3) \right] x(n-m_1) \tag{6.13}$$

The second order terms are denoted by:

$$y_2^{m_1}(n-m_1) = \sum_{m_2=m_1}^{M-1} \sum_{m_3=m_2}^{M-1} h_3(m_1, m_2, m_3)x(n-m_2)x(n-m_3) \tag{6.14}$$

and the set of M Volterra kernel matrices are of the form:

$$H_2^{m_1} = \begin{bmatrix} h_{m_1 00} & h_{m_1 01} & h_{m_1 02} & h_{m_1 03} & \cdots & h_{m_1 0M-1} \\ 0 & h_{m_1 11} & h_{m_1 12} & h_{m_1 13} & \cdots & h_{m_1 1M-1} \\ 0 & 0 & h_{m_1 22} & h_{m_1 23} & \cdots & h_{m_1 2M-1} \\ 0 & 0 & 0 & h_{m_1 33} & \cdots & h_{m_1 3M-1} \\ \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ 0 & 0 & 0 & 0 & \cdots & h_{m_1 M-1 M-1} \end{bmatrix} \tag{6.15}$$

where $m_1 = 0, 1, 2, ..., M-1$. For each $m_1 = 0, 1, 2, ..., M-1$, the second order terms are given by $y_2^{m_1}(n-m_1) = X_1^t H_2^{m_1} X_1$ and a vector of these terms is given by:

$$Y_2^T(n) = [y_2^0(n), y_2^1(n-1), ..., y_2^{m_1}(n-m_1), ..., y_2^M(n-M+1)] \tag{6.16}$$

The third order output is given by:

$$\begin{aligned} y_3(n) &= \sum_{m_1=0}^{M-1} y_2^{m_1}(n-m_1)x(n-m_1) \\ &= Y_2^T(n)X_1 \end{aligned} \tag{6.17}$$

Similar decompositions in terms of the second order representation can be obtained for the higher (4th and 5th) order Volterra components. By processing N samples and considering the matrix of input samples as given in [RPA87], these can also be extended to a matrix-matrix structure.

1.2 A MatRISC processor implementation

A MatRISC processor can be constructed to implement the Volterra filter with the constraint that the size of the processor array must accommodate the dimensions of the kernel matrices and the input matrix, X_1. The optimisation of the processor architecture is different than for the general case where matrices are of an arbitrary size. This leads to the following observations:

- The target time to process a block of input data is proportional to the size of the block. The processor array size should be the same size as the block input matrix to maximise execution speed.
- There would only be one data processor per processing node since block matrix processing is not performed. This means the inter-processing node bandwidth must be maximised (by maximising the bus width and speed) since the array will be bandwidth limited.
- Since there are relatively few kernel matrices ($M^{(Q-2)}$ for a Q^{th} order system of memory depth M), they can be stored in an on-chip register file or local SRAM with the input matrix.

The operations that are required to implement Volterra systems on the MatRISC processor for matrices of the same size as the processor array are presented below with the number of processor cycles (C) required to implement each operation:

- matrix multiplication (performed in $C_{mm} = M$ processor cycles on an $M \times N$ processor array)
- matrix addition (performed in $C_{ad} = 1$ processor cycle)
- element-wise multiplication (performed in $C_{em} = 1$ processor cycle)
- element-wise multiplication from a broadcast row or column (performed in $C_{em} = 1$ processor cycle)
- accumulation of rows or columns to produce a vector (performed in $C_{acc} = [\log_2 i]$ processor cycles where i is the number of operands to be added).

1.3 Second order block Volterra algorithm

Using equations 6.8, 6.10 and 6.11, the operation can proceed in one of two ways, either by computing $H_2 X_1$ or $X_1^T H_2$ depending on whether the transpose or normal matrix for X_1 is

available. The post-multiplication of H_2 by X_1 will be used in the following examples. In the first step, H_2X_1 forms a dense matrix. To calculate the diagonal vectors only, inner products can be formed by the element-wise multiplication of H_2X_1 with X_1 and accumulating the columns of the processor array to form the result vector, Y_2. There is no need to store or map X_1 to X_1^T. Conversely, if pre-multiplication is used then X_1^T must be stored, X_1 is not required and rows must be accumulated. Each element, y_j in the result vector, Y_2 can be calculated by:

$$y_2(j) = \sum_{i=0}^{M-1} (x_{ij} \sum_{k=0}^{M-1} h_{ik} x_{kj}) \tag{6.18}$$

The steps required to compute the second order Volterra system output vector Y_2 for a $M \times N$ matrix X_1 of N samples with memory span M, are presented below for an $M \times N$ processor array.

1. Generate the $M \times M$ kernel matrix and store in the $M \times N$ processor array if $N > M$, or an $M \times M$ processor array if $N < M$. Element h_{ij} is mapped to processing element P_{ij}.

$$H_2 = \begin{bmatrix} h_{00} & h_{01} & h_{02} & h_{03} & h_{04} & \dots & h_{0M-1} \\ 0 & h_{11} & h_{12} & h_{13} & h_{14} & \dots & h_{1M-1} \\ 0 & 0 & h_{22} & h_{23} & h_{24} & \dots & h_{2M-1} \\ 0 & 0 & 0 & h_{33} & h_{34} & \dots & h_{3M-1} \\ 0 & 0 & 0 & 0 & h_{44} & \dots & h_{4M-1} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ 0 & 0 & 0 & 0 & 0 & \dots & h_{M-1M-1} \end{bmatrix}$$

2. An $M \times N$ Toeplitz matrix, X_1 can be generated from $M + N$ input samples stored in a FIFO by diagonal broadcast across the processor array. X_1 can be updated each iteration by loading N new samples into the FIFO and broadcasting the samples as shown in Figure 6.1.

3. Matrix multiply: $T = H_2X_1$ where T is an $M \times N$ matrix.

4. Element-wise multiply (represented by $*$): $W = X_1 * T$

5. Accumulate the columns of W to form a result vector, Y_2 of length N where $y_2(j) = \sum_{i=0}^{M-1} w_{ij}$ performed in parallel in $\lceil \log_2 M \rceil$ processor cycles.

The total time to perform one block Volterra computation and the update is given by:

$$T_B = Tcy(C_{mm} + C_{em} + C_{acc} + C_{update}) \tag{6.19}$$

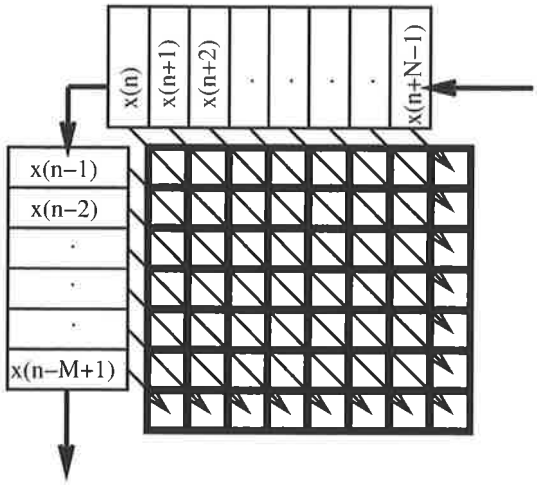


Figure 6.1 Update of the input sample matrix, X_1.

where T_{cy} is the processor cycle time of the MatRISC processor and $C_{update} = 1$ is the number of processor cycles needed to update the input matrix X_1.

1.4 Third order block Volterra algorithm

The second order algorithm can be extended to the third order by multiplying x_1 by a set of M kernel matrices, $H_2^j, j = 0, 1, ..., M-1$. The elements, $y_3(j)$ of the result vector, Y_3 are calculated as:

$$y_3(j) = \sum_{i=0}^{M-1} \left(\sum_{m=0}^{M-1} (x_{mj} x_{ij} \sum_{k=0}^{M-1} (h_{ik}^m x_{kj})) \right) \tag{6.20}$$

The computation steps are given below.

1. Matrix multiply: $T_i = H_2^i X_1, i = 0, 1, ..., M-1$ where T_i is an $M \times N$ matrix as in Figure 6.2.

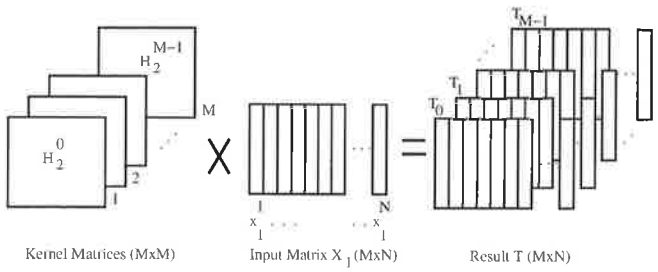


Figure 6.2 Generation of the matrices, T_i.

2. Form an array of matrices R_i, each from M copies of the i^{th} row vector of X_1 where the i^{th} matrix in the array is:

$$R_i = \begin{bmatrix} x_{i0}x_{i1}...x_{iN-1} \\ x_{i0}x_{i1}...x_{iN-1} \\ ... \\ ... \\ ... \\ x_{i0}x_{i1}...x_{iN-1} \end{bmatrix} \tag{6.21}$$

3. Element-wise multiply $(*)$ matrices X, T_i and R_i and accumulate the result matrices to form an $M \times N$ matrix, W as shown in Figure 6.3:

$$W = \sum_{i=0}^{M-1} X_1 * T_i * R_i \tag{6.22}$$

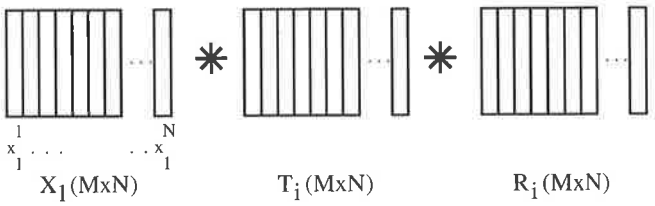


Figure 6.3 Generation of the matrix, W.

4. Accumulate the columns of W to form a result vector Y_3 of length N, where:

$$y_3(j) = \sum_{i=0}^{M-1} w_{ij}, j = 0, 1, ..., N-1 \tag{6.23}$$

1.5 q^{th} order block Volterra algorithm

The second order algorithm can be extended to a q^{th} order Volterra system $q \geq 2$ by multiplying X_1 by a set of $M^{(q-2)}$ kernel matrices, $H_2^{m_0 m_1 ... m_{q-3}}$. The elements, $y_q(j)$ of the result vector, Y_q are calculated as:

$$y_q(j) = \sum_{i=0}^{M-1} \left(\sum_{m_{q-3}=0}^{M-1} \left(... \sum_{m_1=0}^{M-1} \left(\sum_{m_0=0}^{M-1} x_{m_0 j} x_{m_1 j} \cdots x_{m_{q-3} j} x_{ij} \sum_{k=0}^{M-1} \left(h_{ik}^{m_0 m_1 ... m_{q-3}} x_{kj} \right) \right) \right) \right) \tag{6.24}$$

1.6 Performance estimates on a 2-D MatRISC processor

Given a processor cycle time for a MatRISC processor, T_{cy} the time to compute a second order system on an $M \times N$ processor array is:

$$T_{2D} = T_{cy}(M + 1 + [\log_2 M] + C_{update}) \tag{6.25}$$

The time to compute a third order system on an $M \times N$ processor array is:

$$T_{3D} = T_{cy}\left(M^2 + M + [\log_2 M] + 2 + C_{update}\right) \tag{6.26}$$

This includes the terms for M matrix multiplications, M element-wise multiply-accumulates, accumulation of columns in logarithmic time, two cycles for pipeline finishing and the processor cycles required to update the new input matrix X_1. The execution time can be generalised for a third and higher (q^{th}) order systems as:

$$T_{qD} = T_{cy}\left(M^{(q-1)} + (q-2)M^{(q-2)} + [\log_2 M] + 2 + C_{update}\right) \tag{6.27}$$

All performance estimates for an $M \times N$ processor array are independent of N. Table 6.1 shows the execution time (latency, ns) and throughput ($MSamples/s$) of a single update versus the sample depth (and processor size), N for a second order system where $C_{update} = 1$. Figures in bold type indicate that the performance exceeds the target specification. The

Table 6.1 Latency and throughput of a second order Volterra algorithm on a MatRISC processor for $T_{cy} = 5ns$.

Sample Depth, N	Target latency(ns)/ Throughput (MSa/s)	Latency (ns)/ Throughput (MSa/s), $M = 10$	Latency (ns)/ Throughput (MSa/s), $M = 5$
5	53(75)	-	50(100)
10	133(75)	80(125)	50(200)
20	267(75)	80(250)	50(400)
50	667(75)	80(625)	50(1000)
100	1333(75)	80(1250)	50(2000)

second order Volterra system has a fixed latency as it scales with sample depth, N. Given an array cycle time and a memory depth, the size of the processor array ($M \times N$) can be selected to meet the performance requirements. Substituting (6.25) in (6.28) and specifying the target throughput rate, R ($MSamples/s$), the minimum value for N can be calculated where the compute rate yields the target throughput:

$$N = \frac{RT_{2D}}{1000} \tag{6.28}$$

Table 6.2 shows the execution time (latency, ns) and throughput ($MSamples/s$) of a single update versus the sample depth (and processor size), N for a third order Volterra system where $T_{update} = 1$.

Table 6.2 Latency and throughput of a third order Volterra algorithm on a MatRISC processor for $T_{cy} = 5ns$.

Sample Depth, N	Target latency(ns)\ Throughput (MSa\s)	Latency (ns)\ Throughput (MSa\s), $M = 10$	Latency (ns)\ Throughput (MSa\s), $M = 5$
5	53(75)	-	180(28)
10	133(75)	585(17)	180(56)
20	267(75)	585(34)	180(111)
50	667(75)	585(86)	180(278)
100	1333(75)	585(171)	180(556)

The optimum array size (sample depth) for each processor specification is $N = 44$ and $N = 14$ for memory sizes of $M = 10$ and $M = 5$ respectively. Figure 6.4 gives the results of a simulation of the execution time of various order Volterra models against the memory span on a 2-D MatRISC processor where $T_{cy} = 5ns$ and $T_{update} = 1$. The target execution times for four array sizes (N=10, 20, 50 and 100) are also shown for an input sample rate of $75MSamples/s$. Where the execution time for a Volterra system is below a target line, the array size will meet the target sampling rate. Very large processor arrays are needed to implement high order systems with large memory spans.

The techniques that can be used to improve the speed of the system are:

1. Scale the processor array size to process more samples concurrently to improve through-

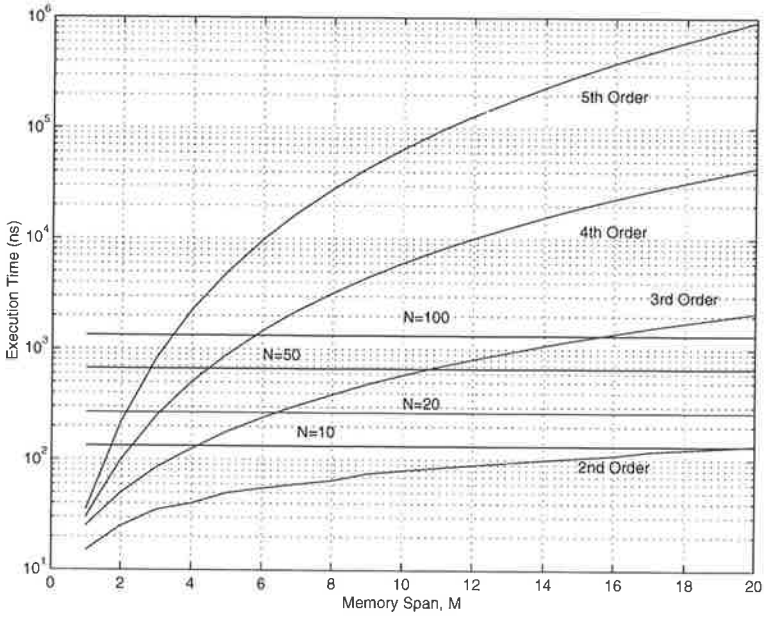


Figure 6.4 Execution time vs memory span for Volterra models on an $M \times N$ MatRISC processor and target execution times for $N = 10, 20, 50, 100$ at $75MSamples/s$.

put at the expense of latency. The scaling limits of the system will cause problems in this case.

2. Use multiple MatRISC processor systems in parallel, each operating on a smaller batch of input data. This would meet the throughput constraints and have a smaller latency than the first option. A drawback is the system control complexity is higher than the first option (eg. scheduling sample blocks).

3. Reduce the sampling rate by either combining (eg. averaging) or skipping samples.

In practice, the size of a single MatRISC processor will be constrained by the data transmission distance in a processor cycle which means that a single 50×10 processor array is an unlikely option for implementing the third order Volterra system using current technology. A combination of options 2 and 3 seem the most likely way of implementing such a system.

1.7 Performance estimates on a 3-D MatRISC processor

From the simulations of the Volterra algorithm on a 2-D MatRISC processor, it can be seen that for systems greater than second order most of the time is spent performing matrix multiplications of the input matrix with the kernel matrices. A 3-D processor array which can be viewed as n_3 2-D MatRISC processors stacked vertically can be used to improve the performance of the system. The speed-up over the 2-D MatRISC processor will be adversely affected by the reduced interprocessor bandwidth because there are now six I/O channels per

node instead of four, given a fixed I/O bandwidth. The planar (n_1 and n_2 plane) and vertical (n_3 direction) bandwidths can be optimised for a particular order Volterra algorithm. The third order Volterra system is used as the starting point since lower order systems will not have any speed-up on a 3-D mesh MatRISC processor.

1.7.1 Third order system

A simple way to implement the third order system is to form the input matrix across one plane and then distribute it vertically amongst all planes (C_{update} now has a horizontal component, $C_{updateP}$ and a vertical component, $C_{updateV}$). On a system where $n_3 = M$, the M kernel matrices are each assigned to a plane and the matrix multiplications are carried out in parallel. The element-wise multiplications are achieved in a single step and accumulation is conducted vertically in logarithmic time, the last accumulation of columns is carried out in the same way. The execution time for a third order system on a 3-D mesh becomes:

$$T_{3D} = T_{cyp}(M + 1 + [\log_2 M] + 2 + C_{updateP}) + T_{cyv}([\log_2 M] + C_{updateV}) \tag{6.29}$$

where the processor cycle time which allows for vertical and planar communications is T_{cyv} and T_{cyp} respectively. For example, the execution time is minimised for $M = 10$ and $C_{update(V,P)} = 1$ when T_{cyp} has 1.6 times the bandwidth of T_{cyv}.

1.7.2 General expression and simulation

A general expression for the execution time of a q^{th} order Volterra filter ($q \geq 4$) implemented on a 3-D mesh is:

$$T_{qD} = T_{cyp}\left(M^{q-2} + (q-2)M^{(q-3)} + [\log_2 M] + 2 + C_{updateP}\right) + T_{cyv}\left([\log_2 M] + C_{updateV}\right) \tag{6.30}$$

Figure 6.5 gives the results of a simulation of the execution time of various order Volterra systems against the memory span on a 3-D MatRISC processor. The target execution times for four array sizes (N=10, 20, 50 and 100) are also shown for an input sample rate of $75MSamples/s$. In this simulation, T_{cyp} and T_{cyv} are equal to $5ns$, however in an optimised system they would differ, the planar bandwidth would be higher than the vertical bandwidth because the planar processor arrays perform most of the operand passing. Where the execution time for a Volterra system is below a target line in Figure 6.5, the array size will meet the target sampling rate. Execution times have improved in proportion to the vertical dimension of the array (n_3) compared to the 2-D MatRISC processor case (Figure 6.4). Despite this improvement, very large processor arrays are still needed to implement high order filters with large memory spans.

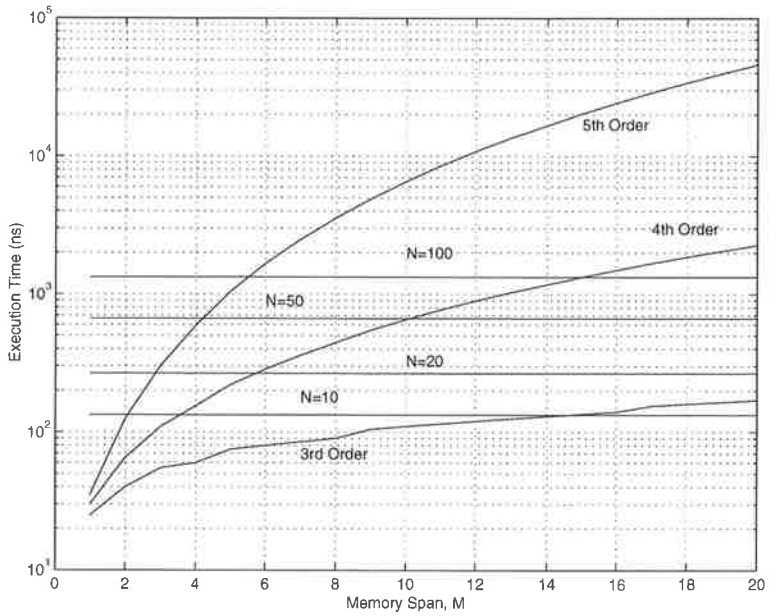


Figure 6.5 Execution time vs memory span for Volterra models on an $M \times M \times N$ MatRISC processor and target execution times for $N = 10, 20, 50, 100$ at $75MSamples/s$.

2 Kalman Filter

Kalman filters [Sor85] are used to generate minimum variance estimates of the state vector of a system when it is subject to noise disturbances. The state, $x(k)$, and the measured output, $y(k)$, at discrete time, k, are given by:

$$x(k+1) = \Phi(k)x(k) + w(k)$$

$$y(k) = H(k)x(k) + v(k)$$

where $y(k)$ is of dimension $(p \times 1)$, the state vector $x(k)$ is of dimension $(n \times 1)$, $w(k)$ and $v(k)$ are white noise processes, $\Phi(k)$ is $(n \times n)$ and $H(k)$ is $(p \times n)$. The six recursive equations for the measurement update and time update computations are given in [Sor85]. Figure 6.6 shows the compute time versus the number of state variables to perform a single Kalman filter update when executed on a MatRISC processor ($p = 4$, $V = 4$) for three output vector lengths. It is clear that real-time implementation of high dimensional Kalman filtering can be performed by this processor.

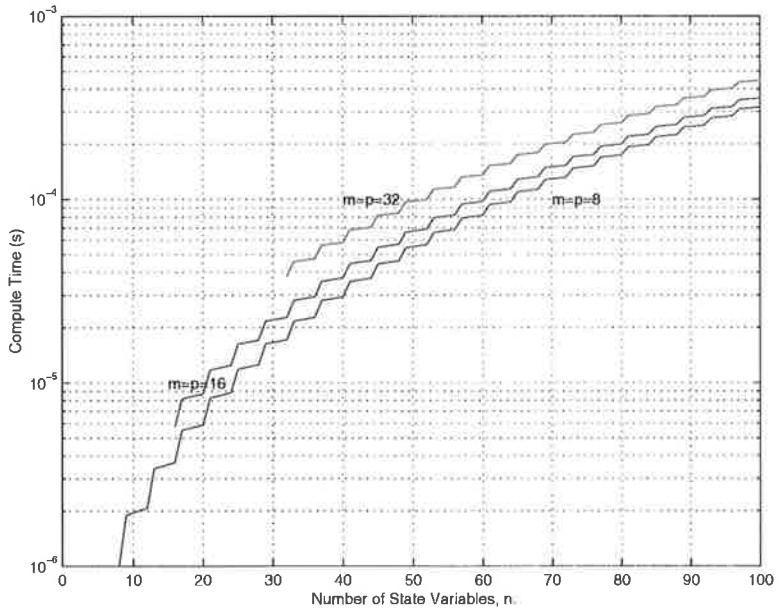


Figure 6.6 Kalman filter time and measurement update execution time.

3 Fourier Transform

Figure 6.7 shows the compute time to perform a prime-factor mapped complex discrete Fourier transform (DFT) for various dimensions (log-log scale) executed on a MatRISC processor ($p = 4$, $V = 4$). The matrices are assumed to be as square as possible for all dimensions which means the execution time is over estimated in some cases. The execution time for a $1K$ point complex DFT is approximately $10\mu s$ which is nearly an order of magnitude faster than the fastest processor system in Table 1.4. The performance graph for the Fourier transform was done by Kiet To.

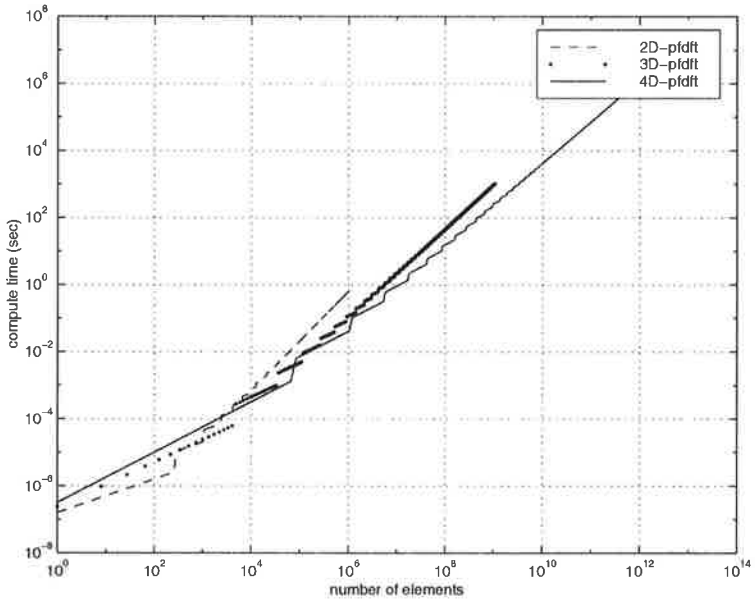


Figure 6.7 Prime-factor mapped discrete Fourier transform execution time.

4 Matrix Inversion

Matrix inversion is used to solve linear systems of equations [GL96] and is one of the more common numerical algorithms. This was mapped to the MatRISC processor and this implementation uses an LU decomposition algorithm with row and column pivoting followed by back substitution. Figure 6.8 shows the compute rate and Figure 6.9 shows the execution time for matrix inversion performed on the MatRISC-1 processor (4×4 array, 200MHz clock,

$nDP = 4$) for a range of matrix sizes from 1-1000 (step size 50). This assumes 15 processor cycles are needed to perform a floating-point division operation. The pipeline is not filled optimally when performing division of large matrices and the results may improved slightly.

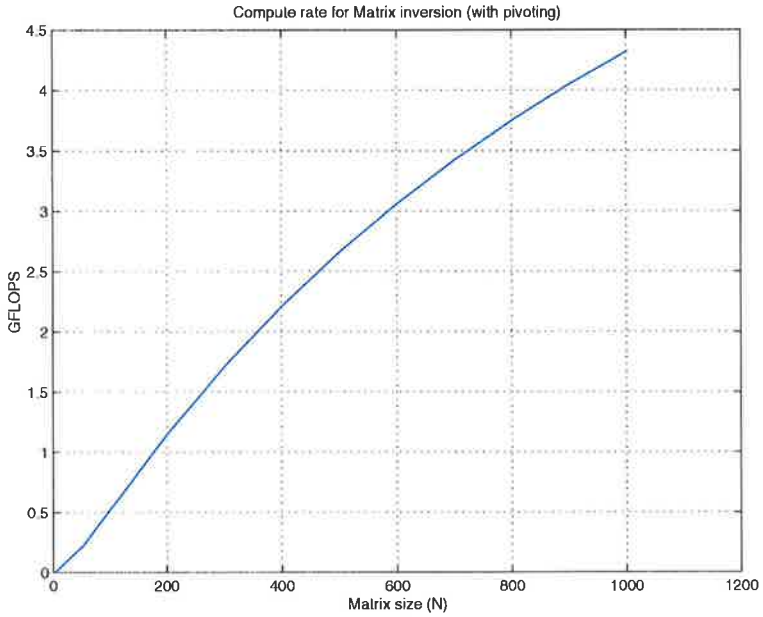


Figure 6.8 Matrix inversion algorithm compute rate for matrix sizes 1-1000.

The asymptotic increase in the compute rate is due to the increasing dominance of an $O(n^3)$ computation over an $O(n^2)$ computation, hence the low compute rate for smaller sized matrices which are more addition-like in performance. For very large matrices, the peak compute rate will saturate just below the theoretical peak compute rate. The performance graph for the matrix inversion was done by Kiet To.

5 Discussion

A number of algorithms have been presented in this chapter that map well onto the MatRISC processor and have good performance.

A Volterra model can represent nonlinear distortion with memory, and its inverse can be used to reduce the distortion by nonlinear compensation. The matrix representation for a second order Volterra model was presented and used to represent higher order Volterra systems which allows efficient implementation and a simpler update process than previously published structures. A real-time implementation using a MatRISC architecture was proposed. Performance

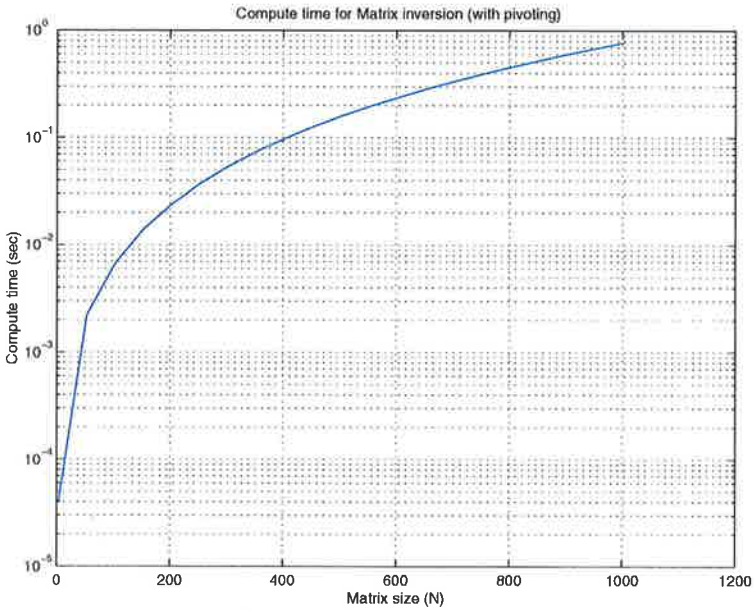


Figure 6.9 Matrix inversion algorithm execution time for matrix sizes 1-1000.

estimates were given for various processor processor cycle times and inter-processor bandwidths. It was found that high order Volterra models with large memory depths require a very large and fast 2-D processor array to compute and update in real-time for signal rates of $75MSamples/s$. A 3-D processor would be needed for the practical implementation of higher nonlinearity order and high memory depth cases. These implementations should allow real-time modelling and compensation of nonlinear distortion.

The Fourier transform, Kalman filter and matrix inversion (using LU decomposition and back-substitution) were also studied and found to give good theoretical performance.

Chapter 7

Summary and Conclusion

THIS thesis presented a generalised architecture for a MatRISC array processor and a CMOS implementation of a specific machine which extends the notion of a vector processor to a matrix processor. In summary, the thesis was organised around the areas of MatRISC processor architecture and algorithms, microarchitecture and CMOS implementation, compilers, computer arithmetic and application algorithms.

The background work in Chapter 1 described the design space of a microprocessor based around three elements: technology, architecture and software. A taxonomy of functional- and data-parallel machines was given and applications that utilise matrix computations were discussed and decomposed into driver, computational and library routines. The block linear algebra subroutines were studied to determine the percentage of time spent in the subroutines by a range of applications. It was found that the speed-up potential achieved by adding matrix hardware support varied between 1.4 to 20 times depending on the size and complexity of the problem. Matrix operators exhibit a high degree of data parallelism and were found to benefit greatly from the addition of a MatRISC processor, in particular the BLAS DGEMM (generalised matrix multiplication) routine due to the $O(N^3)$ compute-time complexity. Processor nodes, interconnect topologies and memories for a parallel matrix processor were discussed and it was found that mesh-connected arrays of processing elements provide the best support for parallel matrix computations. Several key machines including the InMOS T9000 Transputer, TMS Viper-5 (DSP system) and SCAP (scalable array processor - SIMD) were presented and their performance was summarised.

In Chapter 2 the generalised architecture of the MatRISC processor array, technology constraints and exploration of matrix kernel algorithms was presented. Techniques to improve the performance of data parallel machines included pipelining, replication and low and con-

stant network latency. The limit of processor scaling was investigated and to overcome this asynchronous interfaces between synchronous processor modules were discussed. The size of a MatRISC processor integrated with the on-chip cache of a microprocessor will have a synchronous clock, the rate of which will limit the size of the MatRISC processor array. Asynchronous techniques are more suited to system level design. Matrix multiplication and element-wise kernel algorithms were developed for the MatRISC processor and the performance of these algorithms was presented using an abstract machine representation. A matrix multiplication algorithm was developed which introduces a virtual factor to improve performance. A high-level design study of the generalised MatRISC architecture was given showing peak and average performance for a range of matrix sizes from 1-1000. It was shown that a range of architectures are available by varying the array size, number of data processors and virtual factor. Peak performance scales linearly with the virtual factor (which determines the number of data processors) but the average performance degrades relative to the peak performance with increasing virtual factor. The processor was shown to be rate- and processor-optimal over a wide range of matrix sizes, averaging over 90% of the peak performance.

An implementation of the MatRISC architecture was described in Chapters 3 and 4. The microarchitecture and CMOS implementation of MatRISC-1, which is a 4×4 array of processing elements, was presented in Chapter 3. Each component of the processing element was discussed in detail and a VHDL model of the processor array was developed. A minimal 16-bit RISC processor core was developed which featured single cycle program execution for all instructions (including branches) to guarantee that all processing elements remain in lock step. The trade-off was a small instruction memory. A hardware control scheme for each processing element was developed which uses a generalised mapping engine (GME) to issue VLIWs to the data-path with minimal intervention by the RISC instruction processor. The compiler maps a VLIW sequence into VLIWs, cliques and clique instructions which the GME executes with nested for-loop counters implemented in hardware and memories. This allows the instruction processor to perform other tasks, particularly when data dependencies arise. One other microarchitected part included a 4-D address generator which was used to sequence data structures from the data memories while referencing a different data structure each cycle. The results of fabricating parts of the processing element in a CMOS process were presented which include a $3.5ns$ 4kb SRAM, a dynamic bus and crossbar switch with $320ps$ delay and a clock synchronisation unit using digital delay lines.

The software hierarchy of the MatRISC processor was presented in Chapter 4. A set of software tools was developed to support compilation, execution and debugging of code on the processor array. A language called Mcode was presented for use with the MatRISC

processor. Mcode is used to express matrix operations in a program and describes data structure mappings to the data memories.

The design of integer and floating-point (FP) adders which were used to support the hardware design effort of the MatRISC processor were presented in Chapter 5. A detailed discussion of integer parallel prefix adders and IEEE FP adder architectures was given. The term valency was introduced to define the number of cell operator inputs for a parallel prefix adder. This greatly expanded the number of possible adder architectures. An algorithm to synthesise a wide range of parallel prefix adder architectures was developed which covered the design space. The delay versus area of 16, 32 and 64-bit adders was graphed for a $0.25\mu m$ technology which confirmed that the Han-Carlson parallel prefix adder was a good design in terms of area-delay trade-off. Some new architectures with alternating high and low valency rows of prefix cells were also found which may compete with the more well known designs but these require further simulation. New end-around carry parallel prefix adders were also synthesised and presented. A 56-bit dynamic CMOS parallel prefix adder was fabricated in a $0.35\mu m$ process and achieved a $1.75ns$ evaluation time.

Three IEEE double precision FP adders were presented which use a flagged prefix adder combining the addition and rounding of the significand. A minimal hardware, three cycle latency FP adder was designed and implemented in $0.5\mu m$ CMOS which meets the IEEE-754 standard, including support for de-normalised FP numbers. This design contained 33,000 transistors and was successfully tested to a maximum clock rate of $100MHz$. A faster two cycle dual-path FP adder was presented which splits the computation into a near and far path based on the exponent difference and also used a flagged prefix adder. A three cycle FP adder which accumulates operands in a two cycle loop was also presented.

The mapping of application algorithms to the MatRISC architecture and their subsequent performance was presented in Chapter 6. The emphasis was on mapping the Volterra model, a non-linear system with memory, which is an example of a real-time signal processing application. The second order Volterra model was expressed in terms of two dimensional matrices and it was shown that higher order models are also expressed in terms of the second order model. This was then written for implementation on the MatRISC processor and performance estimates were given. It was found that a real-time system incorporating a 2-dimensional MatRISC processor can evaluate second and third order Volterra model in under $1\mu s$ at a data rate of $75MHz$ with memory depths from 1 to 16. The major problem is the large size of the processor array needed to meet these deadline constraints which are particularly bad for high order Volterra models. Four other algorithms were mapped to the MatRISC processor and studied. These included the Fourier transform, Kalman filter, LU decomposition and Gauss-

Jordan elimination. The Fourier transform was found to execute an order of magnitude faster on a MatRISC processor than the workstation systems tested in Chapter 1.

1 The Future of MatRISC Processors

It is clear that microprocessor designers will continue to have more transistors available on a single die and making all of these transistors perform useful work is a challenging task. The most promising design for data parallel processing could be an integrated top-level data cache and MatRISC processor array with a host microprocessor on a single die. This was discussed in Chapter 2. It was shown that less than 10% of the die area is devoted to the MatRISC processor array while typically more than 50% of the die area is second-level cache memory (or the highest level on-chip cache memory). As VLSI feature sizes shrink, the reticle size remains constant and on-chip cache memories grow. The performance of most applications does not scale linearly with cache memory size and a limit will be reached beyond which extra cache memory does not provide a measurable performance benefit. At this limit, MatRISC processor arrays can be employed to build a more intelligent on-chip cache memory system and provide support for many signal processing and real-time applications with a high degree of data parallelism.

The key issues for developing an integrated MatRISC processor array continue to be:

- how the host processor operating system will interact with the processor array. A second-level cache memory controller can process cache requests, however, calls to the processor array must be through either instructions on the host processor or a driver and port.
- developing a better MatRISC compiler. The Mcode compiler back-end should be modified to reduce the compile time and to be independent of the size of data structures.
- investigating other application areas such as image processing which have not been adequately addressed in this thesis but is a promising area for future research.
- investigating low power and low area techniques and reduce the size of a processing element.

Bibliography

[ABB+] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D.Sorensen. LAPACK Version 3.0. Web Reference *www.netlib.org/lapack*.

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users Guide*. SIAM, May 1992.

[Adv95a] Advanced Microelectronics Inc. *IRSIM Users Manual*, 1995.

[Adv95b] Advanced Microelectronics Inc. *MAGIC Users Manual*, 1995.

[AG94] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings, 1994.

[App97] Sam S. Appleton. *Performance Directed Design of Asynchronous VLSI Systems*. PhD thesis, Department of Electrical and Electronic Engineering, The University of Adelaide, August 1997.

[AS72] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions*. Dover Press, ninth edition, 1972.

[Ash96] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan-Kaufmann Publishers, 1996.

[ASU86] A.V Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Bab88] R.G. Babb. *Programming Parallel Processors*. Addison-Wesley, 1988.

[Bak90] H.B. Bakoglu. *Circuits, Interconnections and Packaging for VLSI*. Addison-Wesley, 1990.

[Bar98] J.G.P. Barnes. *Progrmmaing in Ada 95.* Addison-Wesley, 1998.

[BBC88] E. Biglieri, S. Barberis, and M. Catena. "Analysis and Compensation of Nonlinearities in Digital Transmission Systems". *IEEE Journal on Selected Areas of Comms.*, COM-6(1):42–51, January 1988.

[BBD79] S. Benedetto, E. Biglieri, and R. Daffara. "Modeling and Performance Evaluation of Nonlinear Satellite Links-A Volterra Series Approach". *IEEE Trans. on Aerospace and Electronic Systems*, 15(4):494–507, July 1979.

[BCC+88] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb. "iWarp: an integrated solution to high speed parallel computing". In *Proc. Supercomputing 88*, Kissimmee, Florida, USA, November 1988.

[Bet00] Nicholas M. Betts. *Design and Evaluation of a Memory Architecture for a Parallel Matrix Processor Array.* PhD thesis, Department of Electrical and Electronic Engineering, The University of Adelaide, 2000.

[BK82] R.P. Brent and H.T. Kung. "A Regular Layout for Parallel Adders". *IEEE Transactions on Computers*, 31:260–264, March 1982.

[Boo51] D.A. Booth. "A Signed Binary Multiplication Technique". *Q.J. Mech. Appl. Math.*, 4:236–240, 1951.

[BSB96] Andrew J. Beaumont-Smith and Neil Burgess. "Modified Kogge-Stone VLSI adder architecture for GaAs technology". In *Proc. European GaAs and Related III-V Compounds Application Symposium*, page 2A1, Paris, France, June 1996.

[BSB97] Andrew J. Beaumont-Smith and Neil Burgess. "A GaAs 32-bit adder". In *Proc.* 13^{th} *IEEE Symposium on Computer Arithmetic*, pages 10–17, Asilomar, California, USA, July 1997.

[BSBCL97] Andrew J. Beaumont-Smith, Neil Burgess, Song Cui, and Michael J. Liebelt. "GaAs multiplier and adder designs for high-speed DSP applications". In *Proc.* 31^{st} *Asilomar Conference on Signals Systems and Computers*, pages 1517–1521, Asilomar, California, USA, November 1997.

[BSBLL99] Andrew J. Beaumont-Smith, Neil Burgess, Stephane Lefreré, and Cheng-Chew Lim. "Reduced Latency IEEE Floating-Point Standard Adder Architectures". In *Proc.* 14^{th} *IEEE Symposium on Computer Arithmetic*, pages 35–42, Adelaide, Australia, April 1999.

[BSL01] Andrew J. Beaumont-Smith and Cheng-Chew Lim. "Parallel Prefix Adder Design". In *Proc. 15th IEEE Symposium on Computer Arithmetic*, pages 218–225, Vail, Colorado, USA, June 2001.

[BSLL+97] Andrew J. Beaumont-Smith, Michael J. Liebelt, Cheng-Chew Lim, Kiet N. To, and Warren Marwood. "A Digital Signal Multiprocessor for Matrix Applications". In *Proc. 14th Australian Microelectronics Conference*, pages 245–250, Melbourne, Australia, September 1997.

[BSLTM99] Andrew J. Beaumont-Smith, Cheng-Chew Lim, John E. Tsimbinos, and Warren Marwood. "A VLSI chip Implementation of an A/D Converter Error Table Compensator". In *Proc. 3rd IEE International Conference on Advanced A/D and D/A Conversion Techniques and their Applications (ADDA'99)*, pages 122–125, University of Strathclyde, Glasgow, UK, July 1999.

[BSMT+98] Andrew J. Beaumont-Smith, Warren Marwood, Kiet N. To, Cheng-Chew Lim, and Michael J. Liebelt. "MatRISC-1: A RISC Multiprocessor for Matrix-Based Real Time Applications". In *Proc. 5th Australasian Conference on Parallel and Real-Time Systems (PART'98)*, pages 320–331, Adelaide, Australia, September 1998.

[BSTL+99] Andrew J. Beaumont-Smith, John E. Tsimbinos, Cheng-Chew Lim, Warren Marwood, and Neil Burgess. "A 10GOPS Transversal Filter and Error Table Compensator". In *Proc. SPIE '99*, pages 157–163, Denver, Colorado, USA, July 1999.

[BSTLM01] Andrew J. Beaumont-Smith, John E. Tsimbinos, Cheng-Chew Lim, and Warren Marwood. "A VLSI Chip Implementation of an A/D Converter Error Table Compensator". *Computer Standards and Interfaces*, 23:111–122, 2001.

[BSTM+97] Andrew J. Beaumont-Smith, John E. Tsimbinos, Warren Marwood, Cheng-Chew Lim, and Michael J. Liebelt. "A Matrix Processor Implementation of the Volterra Model". In *Proc. 2nd Australian Workshop on Signal Processing Applications*, pages 195–198, Brisbane, Australia, December 1997.

[Bur98] Neil Burgess. "The Flagged Prefix Adder for Dual Additions". In *Proc. SPIE ASPAAI-7*, volume 3461, pages 567–575, San Diego, USA, July 1998.

[CBF01] Anantha Chandrakasan, William J. Bowhill, and Frank Fox. *Design of High Performance Microprocessor Circuits*. IEEE Press, 2001.

[CCMC92] R.J. Clarke, I.A. Curtis, W. Marwood, and A.P. Clarke. "A Floating Point Matrix Arithmetic Processor: An Implementation of the SCAP Concept". In *Proc. Asia-Pacific Conference on Circuits and Systems*, pages 519–524, Australia, December 1992.

[CDW94] J. Choi, J.J. Dongarra, and D.W. Walker. "PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers". *"Concurrency: Practice and Experience"*, 6(7):543–570, 1994.

[CKHP90] Y. S. Cho, S. B. Kim, E. L. Hixson, and E. J. Powers. "Nonlinear Distortion Analysis Using Digital Higher-order Coherence Spectra". In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pages 1165–1168, 1990.

[CMB+99] J. Clouser, M. Matson, R. Badeau, R. Dupcak, S. Samudrala, R. Allmon, and N. Fairbanks. "A 600-MHz Superscalar Floating-Point Processor". *IEEE Journal of Solid-State Circuits*, 34(7):1026–1029, July 1999.

[Com] Scyld Computing. The Beowulf Project. Web Reference *www.beowulf.org*.

[CS94] R. Cypher and J.L.C. Sanz. *The SIMD model of Parallel Computation.* Springer-Verlag, 1994.

[CS96] J. Craninckx and M. Steyaert. *Analog Circuit Design*, chapter Low-Phase-Noise GigaHertz Voltage-Controlled Oscillators in CMOS. Kluwer Academic Press, 1996.

[CTP96] T.L. Casavant, P. Tvrdik, and F. Plasil. *Parallel Computers - Theory and Practice.* IEEE Press, 1996.

[DAC+92] Daniel W. Dopperpuhl, Robert Anglin, Linda Chao, Bruce Gieseke, Kathryn Kuchler, Liam Madden, James Montanaro, Sridhar Samudrala, Richard T. Witekn, Davoid Bertucci, Robert A. Conrad, Soha M.N. Hassoun, Maureen Ladd, Edward J. McLellan, Donald A. Priore, Sribalan Santhanam, Randy Allmon, Sharon Britton, Daniel E. Dever, Gregory W. Hoeppner, Burton M. Leary, Derrick R. Meyer, and Vidya Rajagopalan. "A 200-MHz 64-bit Dual-Issue CMOS Microprocessor". *Digital Technical Journal*, 4(4), 1992.

[Dal89] I.W. Dall. "Modelling Nonlinear Distortion in Mixers". In *Proc. of Australian Symp. on Signal Processing and Applications (ASSPA 89)*, pages 118–122, Adelaide, Australia, April 1989.

[DCDH98] J. Dongarra, J. Du Cruz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. Technical report, "Mathematics and Computer Science Division Argonne National Laboratory", August 1998.

[Dev96] Analog Devices. SHARCPAC Module Specification. Technical report, Analog Devices, 1996.

[Far81] M.P. Farmwald. *On the Design of High Performance Digital Arithmetic Units.* PhD thesis, Stanford University, August 1981.

[Fis83] J. Fisher. "Very Long Instruction Word Architectures and the ELI-512". In *Proc. IEEE* 10^{th} *International Symposium on Computer Architecture*, pages 140–150, Los Alamitos, California, USA, 1983.

[FKM+83] A.L. Fisher, H.T. Kung, L.M. Monier, H. Walker, and Y. Dohi. "Design of the PSC: A Programmable Systolic Chip". In *Proc.* 3^{rd} *Caltech Conference on VLSI*, pages 287–302, Pasadena, California, USA, March 1983.

[Fly72] M.J. Flynn. "Some Computer Organizations and Their Effectiveness". *IEEE Transactions on Computers*, c–21(9):948–960, September 1972.

[Fra94] W. A. Frank. "MMD - An Efficient Approximation to the 2nd Order Volterra Filter". In *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing, (ICASSP'94)*, pages 517–520, Adelaide, Australia, April 1994.

[GD85] Lance A. Glasser and Daniel W. Dobberpuhl. *The Design and Analysis of VLSI Circuits.* Addison-Wesley, 1985.

[Gep99] Linda Geppert. "Approaching the 100M transistor IC". *IEEE Spectrum*, 36(7):22–24, July 1999.

[GL96] G.H.Golub and C.R.Van Loan. *Matrix Computations.* John Hopkins University Press, third edition, 1996.

[GMS+94] A. Gunzinger, U.A. Muller, W. Scott, B. Baumle, P. Kohler, H.R. vonder Muhll, F. Muller-Plathe, W.F. van Gunsteren, and W. Guggenbuhl. Achieving Super Computer Performance with a DSP Array Processor. Technical report, Swiss Federal Institute of Technology, Switzerland, 1994.

[Gol90] David Goldberg. *Computer Architecture: A Quantitative Approach*, chapter Appendix A. Morgan Kaufmann, 1990.

[GP00] Linda Geppert and Tekla S. Perry. "Transmeta's new Chip". *IEEE Spectrum*, 37(5):26–33, May 2000.

[Gre95] D. Greenley et al. "UltraSPARC: the next generation superscalar 64-bit SPARC". In *Digest of Papers, COMPCON 95*, pages 442–451, March 1995.

[GSvdG95] B. Grayson, A.P. Shah, and R.A. van de Geijn. A High Performance Strassen Implementation. Technical report, Dept. of Electrical and Computer Engineering, The University of Texas at Austin, June 1995.

[HC87] T. Han and D.A. Carlson. "Fast area-efficient VLSI adders". In *Proc. 8th Symposium on Computer Arithmetic*, pages 49–56, Como, USA, September 1987.

[HCB96] C. Heikes and G. Colon-Bonet. "A dual floating point coprocessor with an FMAC architecture". In *ISSCC Dig. Tech. Papers*, pages 354–355, February 1996.

[HF84] Kai Hwang and Fayé. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[Hil85] Daniel W. Hillis. *The connection machine.* MIT Press, 1985.

[HIO+97] Y. Hagihara, S. Inui, F. Okamoto, M. Nishida, T. Nakamura, and H. Yamada. "Floating-Point Datapaths with Online Built-In Self Speed Test". *IEEE Journal of Solid-State Circuits*, 32(3):444–449, March 1997.

[HJS99] M.D. Hill, N.P. Jouppi, and G.S. Sohi. *Readings in Computer Architecture.* Morgan-Kaufmann Publishers, 1999.

[HM90] E. Hokenek and R.K. Montoye. "Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit". *IBM Journal of Research and Development*, 34(1):71–77, January 1990.

[Hod98a] M. Hodson. MatRISC Processor Core Assembler User Manual. Technical Report CHiPTec, Department of Electrical and Electronic Engineering, The University of Adelaide, 1998.

[Hod98b] M. Hodson. MatRISC Processor Core Debugger User Manual. Technical Report CHiPTec, Department of Electrical and Electronic Engineering, The University of Adelaide, 1998.

[HSS90] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, Second edition, 1990.

[Hwa79] K. Hwang. *Computer Arithmetic: Principles, Architecture and Design.* John Wiley & Sons, 1979.

[HX96] K. Hwang and Z. Xu. "Scalable Parallel Computers for Real Time Signal Processing". *IEEE Signal Processing Magazine*, 13(4):50–66, July 1996.

[IT89] R.N. Ibbett and N.P. Topham. *Architecture of High Performance Computers, Volumes 1 & 2.* MacMillan, 1989.

[JAB+01] A. Jain, W. Anderson, T. Benninghoff, D. Berucci, M. Braganza, J. Burnetie, T. Chang, J. Eble, R. Faber, O. Gowda, J. Grodstein, G. Hess, J. Kowaleski, A. Kumar, B. Miller, R. Mueller, P. Paul, J. Pickholtz, S. Russell, M. Shen, T. Truex, A. Vardharajan, D. Xanthopulos, and T. Zou. "A 1.2GHz Alpha Microprocessor with 44.8GB/s Chip Pin Bandwidth". In *Digest of Technical papers, 2001 IEEE International Solid State Circuits Conference (ISSCC)*, pages 240–1, San Francisco, California, USA, February 2001.

[Jai94] Y. Jain. Parallel Processing with the TMS320C40 Parallel Digital Signal Processor. Technical Report SPRA053, Texas Instruments, February 1994.

[JBD+01] N.A. Jurd, J.S. Barkatullah, R.O. Dizon, T.D. Fletcher, and P.D. Madland. "A Multi-GHz Clocking Scheme for the Pentium 4(R) Microprocessor". In *Digest of Technical papers, 2001 IEEE International Solid State Circuits Conference (ISSCC)*, pages 404–5, San Francisco, California, USA, February 2001.

[JH88] Mark G. Johnson and Edwin L. Hudson. A Variable Delay Line PLL for CPU-Coprocessor Synchronisation. *IEEE Journal of Solid-State Circuits*, 23(5), October 1988.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design.* Prentice-Hall, 1991.

[KF89] L. Kohn and S-W. Fu. "A 1,000,000 transistor microprocessor". In *IEEE International Solid-State Circuits Conf. Dig. Tech. papers*, pages 54–55, 1989.

[KKA+96] Y. Kondo, Y. Koshiba, Y. Arima, M. Murasaki, T. Yamada, H. Amishiro, H. Mori, and K. Kyuma. "A 1.2GFLOPS Neural Network Chip for High-Speed Neural Network Servers". *IEEE Jounal of Solid-State Circuits*, 31(6):860–864, June 1996.

[KL78] H.T. Kung and C.E. Leiserson. "Systolic Arrays (for VLSI)". In Duff and Stewart, editors, *Proc. Symposium on Sparse Matrix Computations and their Applications*, 1978.

[Kno91] Simon Knowles. "Arithmetic Processor Design for the T9000 Transputer". In *Proc. SPIE ASPAAI-2*, volume 1566, pages 230–243, San Diego, USA, July 1991.

[Kno99] Simon Knowles. "A Family of Adders". In *Proc. 14th IEEE Symposium on Computer Arithmetic*, pages 30–34, Adelaide, Australia, April 1999.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Seminumerical Algorithms*, volume 2. Addison-Wesley, third edition, 1998.

[Kog81] P.M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, 1981.

[Kor93] Israel Koren. *Computer Arithmetic Algorithms.* Prentice-Hall, 1993.

[KOS+93] Natsuki Kushiyama, Shigeo Ohshima, Don Stark, Hiroyuki Noji, Kiyofumi Sakurai, Satoru Takase, Tohru Furuyama, Richard M. Barth, Andy Chan, John Dillon, James A. Gasbarro, Matthew M. Griffin, Mark Horowitz, Thomas H. Lee, and Victor Lee. "A 500-Megabyte/s Data-Rate 4.5M DRAM". *IEEE Journal of Solid-State Circuits*, 28(4), April 1993.

[KP79] Y.C. Kim and E.J. Powers. "Digital Bispectrum Analysis and its Application to Nonlinear Wave Interactions". *IEEE Trans. on Plasma Science*, PS-7(2):120–131, June 1979.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, second edition, 1988.

[Kri89] R.V. Krishnamurthy. *Parallel Processing, principles and practice.* Addison-Wesley, 1989.

[KS73] P.M. Kogge and H.S. Stone. "A parallel algorithm for the efficient solution of a general class of recurrence relations". *IEEE Transactions on Computers*, 22:786–793, August 1973.

[KTM91] J. Kowalczuk, S. Tudor, and D. Mlynek. "A new architecture for an automatic generation of fast pipeline adders". In *Proc. ESSCIRC*, pages 101–104, Milan, Italy, September 1991.

[Kun88] S.Y. Kung. *VLSI Array Processors.* Prentice-Hall, 1988.

[KWF+96] J.A. Kowaleski Jr., G.M. Wolrich, T.C. Fischer, R.J. Dupcak, P.L. Kroesen, T Pham, and A. Olesin. "A dual-execution pipelined floating point CMOS processor". In *ISSCC Dig. Tech. Papers*, pages 358–359, February 1996.

[LB90] Stephen I. Long and Steven E. Butner. *Gallium Arsenide Digital Integrated Circuit Design.* McGraw-Hill, 1990.

[LDH+94] T.H. Lee, K.S. Donnelly, J.T.C Ho, J. Zerbe, M.G. Johnson, and T. Ishikawa. "A 2.5V CMOS Delay-Locked Loop ofr an 18Mbit, 500Megabyte/s DRAM". *IEEE Journal of Solid State Circuits*, 29(12):1491–1496, December 1994.

[LF80] R.E. Ladner and M.J. Fischer. "Parallel Prefix Computation". *Journal of the ACM*, 27(4):831–838, October 1980.

[Lin81] Huey Ling. "High-Speed Binary Adder". *IBM Journal of Research and Development*, 25(3):156–160, May 1981.

[LPS94] G. Lazzarin, S. Pupolin, and A. Sarti. "Nonlinearity Compensation in Digital Radio Systems". *IEEE Trans. on Comms.*, 42(2/3/4):988–999, February/March/April 1994.

[LSU89] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. *VHDL, hardware description and design.* Kluwer Academic Publishers, 1989.

[LWMM96] S. Lakhani, Y. Wang, A. Milenkovic, and V. Milutinovic. "2d matrix multiplication on a 3d systolic array". *"Microelectronics Journal"*, 27(1):11–22, 1996.

[Mad95] V.K. Madisetti. *VLSI Digital Signal Processors- An Introduction to Rapid Prototyping and Design Synthesis.* IEEE Press, 1995.

[Mar90] A.J. Martin. *Developments in Concurrency and Communication*, chapter Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. Addison-Wesley, 1990.

[Mar94] Warren Marwood. *An Integrated Multiprocessor for Matrix Algorithms.* PhD thesis, Department of Electrical and Electronic Engineering, The University of Adelaide, June 1994.

[Mat99] *Matlab User's Guide, Version 6.0*, 1999. Natick, Massachusetts, USA.

[McC95] J.D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[Mer94] B.G. Mertzios. "Parallel Modeling and Structure of Nonlinear Volterra Discrete Systems". *IEEE Trans. on Circuits and Systems-I: Fundamental Theory and Applications*, 41(5):359–371, May 1994.

[Met96] Meta-Software Inc. *HSPICE Users Manual*, v.96-1 edition, 1996.

[MHR90] R.K. Montoye, E. Hokenek, and S.L. Runyon. "Design of the IBM RISC System/6000 floating-point execution unit". *IBM Journal of Research and Development*, 34(1):59–70, January 1990.

[MKA+01] Sanu Mathew, Ram Krishnamurthy, Mark Anders, Rafael Rios, Kaizad Mistry, and K. Soumyanath. "Sub-500ps 64b ALUs in 0.18μm SOI/Bulk CMOS: Design and Scalaing Trends". In *Proc. ISSCC Digest of Technical Papers*, pages 318–319, February 2001.

[MM78] P.Z. Marmarelis and V.Z. Marmarelis. *Analysis of Physiological Systems-The White Noise Approach.* Plenum Press, 1978. Chapter 6 and 7.

[MM96] J. Marsh and J. McCaskill. "The TM-66 swiFFT DSP Chip - Architecture, Algorithms and Applications". In *Proc. SPIE AeroSense Symposium*, April 1996.

[MMU86] W. Moore, A. McCabe, and R. Urquhart. *Systolic Arrays.* Adam Hilger, 1986.

[Mod88] J.J. Modi. *Parallel Algorithms and Matrix Computation.* Oxford Press, 1988.

[Moo65] G.E. Moore. "Cramming more components onto integrated circuits". *Electronics*, pages 114–117, April 1965.

[Mur90] William D. Murray. *Computer and Digital System Architecture.* Prentice-Hall, 1990.

[Naf96] Sam Naffziger. "A Sub-nanosecond 0.5μm 64b Adder Design". In *Proc. International Solid-State Circuits Conference*, pages 362–363, February 1996.

[Nat01] Nature. "The Human Genome". *Nature*, 409(6822), February 2001.

[NMLE97] A.M. Nielsen, D.W. Matula, C.N. Lyu, and G. Even. "Pipelined Packet-Forwarding Floating Point: II. An Adder.". In *Proc. IEEE* 13^{th} *International Symposium on Computer Arithmetic*, pages 148–155, Asilomar, California, USA, July 1997.

[NV95] R.D. Nowak and B.D. Van Veen. "Efficient Methods for Identification of Volterra Filter Models". *Signal Processing*, 38(3):417–428, August 1995.

[OATF97] S.F. Oberman, H. Al-Twaijry, and M.J. Flynn. "The SNAP Project: Design of Floating Point Arithmetic Units". In *Proc. IEEE* 13^{th} *International Symposium on Computer Arithmetic*, pages 156–165, 1997.

[Occ88] InMOS Ltd. *OCCAM 2 Reference Manual*, 1988.

[Okl94] V. Oklobdzija. "An Algorithmic and novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis". *IEEE Transactions on VLSI Systems*, 2(1):124–128, March 1994.

[Omo94] Amos R. Omondi. *Computer Arithmetic Systems, Algorithms, Architecture and Implementations.* Prentice-Hall, 1994.

[Pap96] David B. Papworth. "Tuning the Pentium Pro Microarchitecture". *IEEE Micro*, April 1996.

[PE88] Douglas A. Pucknell and Kamran Eshraghian. *Basic VLSI Design : Systems and Circuits.* Prentice-Hall, second edition, 1988.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture : A Quantitative Approach.* Morgan-Kaufmann Publishers, Second edition, 1996.

[PH98] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface.* Morgan-Kaufmann Publishers, Second edition, 1998.

[PLK+96] W-C. Park, S-W. Lee, O-Y. Kown, T-D. Han, and S-D. Kim. "Floating Point Adder/Subtractor Performing IEEE Rounding and Addition/Subtraction in Parallel". *IEICE Transactions on Information and Systems*, E79-D(4):297–305, April 1996.

[Pol88] C.D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[PR98] J.F. Parker and D. Ray. "A 1.6GHz CMOS PLL with On-Chip Loop Filter". *IEEE Journal of Solid State Circuits*, 33(3):337–343, March 1998.

[PSG87] S. Peng, S. Samudrala, and M. Gavrielov. "On the Implementation of Shifters, Multipliers, and Dividers in VLSI Floating Point Units". In *Proc. IEEE Int'l Symp. on Computer Arithmetic*, pages 95–102, 1987.

[QF91] N. Quach and M. Flynn. "Design and Implementation of the SNAP Floating-Point Adder". Technical Report CSL-TR-91-501, Stanford University, December 1991.

[QTF91] N. Quach, N. Takagi, and M. Flynn. "On Fast IEEE Rounding". Technical Report CSL-TR-91-459, Stanford University, January 1991.

[Qui87] M.J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, 1987.

[RAM93a] RAMBUS Inc. *RAMBUS Architecture Overview*, 1993.

[RAM93b] RAMBUS Inc. *RDRAM Reference Manual*, 1993.

[RPA87] C.P. Ritz, E.J. Powers, and C.K. An. "Applications of Digital Bispectral Analysis to Nonlinear Wave Phenomena". In *Proc. Int. Symp. on Signal Processing and its Applications (ISSPA'87)*, pages 352–356, Brisbane, Australia, August 1987.

[Sch89] M. Schetzen. *The Volterra and Wiener Theories of Nonlinear Systems.* Krieger, Malabar, Florida, reprint edition, 1989.

[Sch90] E.E. Schwartzlander. *"Computer Arithmetic, Volumes 1 & 2".* IEEE Computer Society Press, 1990.

[SFK97] Dezso Sima, Terence Fountain, and Peter Kacsuk. *Advanced Computer Architectures - A Design Space Approach.* Addison-Wesley, 1997.

[Sho88] Masakazu Shoji. *CMOS Digital Circuit Technology.* Prentice-Hall, 1988.

[SIA99] SIA. *The National Technology Roadmap for Semiconductors.* Semiconductor Industry Association, 1999. Web Reference *www.semichips.org.*

[Sie85] H.J. Siegel. *Interconnection Networks for Large Scale Parallel Processing.* Lexington Books, 1985.

[Sim98] R. Simar Jr. "Codevelopment of the TMS320C6x VelociTI Architecture and Compiler". In *Proc. ICASSP'98*, Seattle Washington, May 1998.

[SK96] Phillip M. Sailer and David R. Kaeli. *The DLX Architecture Handbook.* Morgan-Kaufmann Publishers, 1996.

[Ski88] D.B. Skillicorn. "A Taxonomy for Computer Architectures". *IEEE Computer Magazine*, 21(11):46–57, November 1988.

[Sny82] L. Snyder. "Introduction to the Configurable, Highly Parallel Computer". *IEEE Computer*, pages 47–56, January 1982.

[Soc85] IEEE Computer Society. ANSI-IEEE Standard 754 - 1985 for Binary Floating-Point Arithmetic. Technical report, IEEE, Los Alamitos, California, USA, 1985.

[Soh98] G. Sohi. *25 years of the International Symposia on Computer Architecture: selected papers.* ACM Press, 1998.

[Sor85] Harold W. Sorenson. *Kalman Filtering: Theory and Application.* IEEE Press, 1985.

[Sze88] S.M. Sze. *VLSI Technology.* McGraw-Hill, second edition, 1988.

[Ter95] R.E. Terrill. "Aladdin: Packaging Lessons Learned". In *Proc. ICEMCM'95*, 1995.

[TL93] J. Tsimbinos and K. V. Lever. "Applications of Higher-order Statistics to Modelling, Identification and Cancellation of Nonlinear Distortion in High-speed Samplers and Analogue-to-digital Converters using the Volterra and Wiener Models". In *Proc. IEEE Signal Processing Workshop on Higher-Order Statistics*, pages 379–383, South Lake Tahoe, California, USA, June 1993.

[TL96] J. Tsimbinos and K.V. Lever. "The Computational Complexity of Nonlinear Compensators Based on the Volterra Inverse". In *Proc. of the* 8^{th} *IEEE Signal Processing Workshop on Statistical Signal and Array Processing (SSAP96)*, pages 387–390, Corfu, Greece, June 1996.

[TLBS+99] Kiet N. To, Cheng-Chew Lim, Andrew J. Beaumont-Smith, Michael J. Liebelt, and Warren Marwood. "An Array Processor Architecture for Support Vector Learning". In *Proc.* 3^{rd} *International Conference on Knowledge-based Intelligent Information Engineering Systems*, pages 377–380, Adelaide, Australia, August 1999.

[TMS96] TMS. GFLOPS for the masses. Technical report, Texas Memory Systems Inc., 1996. Web Reference *www.texmemsys.com/index.html.*

[To96] Kiet To. Memory-memory Architecture using RAMBUS Technology in Parallel. Technical Report CHiPTec 6/96, Department of Electrical and Electronic Engineering, The University of Adelaide, 1996.

[To97] Kiet To. Performing the Singular Valued Decomposition on the MDMA Architecture. Technical Report CHiPTec 2/97, Department of Electrical and Electronic Engineering, The University of Adelaide, 1997.

[To99] Kiet To. Implementation of a 200MHz CMOS 2k x16 dual read port SRAM. Technical Report CHiPTec 2/99, Department of Electrical and Electronic Engineering, The University of Adelaide, 1999.

[Tra88] InMOS Ltd. *Transputer Reference Manual*, 1988.

[Tse90] P-S. Tseng. *A Systolic Array Parallelizing Compiler.* Kluwer Academic Publishers, 1990.

[Tsi95] J. Tsimbinos. *Identification And Compensation of Nonlinear Distortion.* PhD thesis, School of Electronic Engineering, University of South Australia, Adelaide, February 1995.

[Tya93] A. Tyagi. "A reduced-area scheme for carry-select adders". *IEEE Transactions on Computers*, 42:1163–1179, 1993.

[USS+96] K. Ueda, H. Suzuki, K. Suda, H. Shinohara, and K. Mashiko. "A 64-bit Carry Look Ahead Adder Using Pass Transistor BiCMOS Gates". *IEEE Journal of Solid State Circuits*, 31(6):810–818, June 1996.

[vdGW97] R.A. van de Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, Dept. of Electrical and Computer Engineering, The University of Texas at Austin, 1997.

[vKAPD96] V. von Kaenel, D. Aebischer, C. Piquet, and E. Dijkstra. A 320MHz, 1.5mW at 1.35V CMOS PLL for Microprocessor Clock Generation. In *Proc. International Solid State Circuits Conference*, pages 132–133, 1996.

[Wal64] C.S. Wallace. "A suggestion for a fast multiplier". *IEEE Trans. Electron. Comput.*, EC-13:14–17, 1964.

[Wal90] David W. Wall. Limits of Instruction-Level Parallelism. Technical Note TN-15, Digital Western Research Laboratory, Palo Alto, California, USA, December 1990. Web Reference *www.research.digital.com/wrl/home.html.*

[WE95] Neil Weste and Kamran Eshragian. *Principles of CMOS VLSI Design.* Addison Wesley, second edition, 1995.

[WOK+97] J.P. Wittenburg, M. Ohmacht, J. Kneip, W. Hinrichs, and P. Pirsch. "HiPAR-DSP: A Parallel VLIW RISC Processor for Real Time Image Processing Applications". In *Proc. 3rd International Conference on Algorithms and Architectures for Parallel Processing*, pages 155–162, Melbourne, Australia, December 1997.

[WS81] H.J. Whitehouse and J.M. Speiser. "SONAR Applications of Systolic Array Technology". In *Conference Record, IEEE EASCON*, Washington D.C., USA, November 1981.

[Zim98] Reto Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis.* Hartung-Gorre, 1998.

[Zor92] G. Zorpette. "The Power of Parallelism". *IEEE Spectrum*, pages 28–33, September 1992.

Errata

1. Page 6, line 4: Replace "DSM machines are based on the Stanford DASH architecture where" by "In DSM machines".

2. Page 45, line 16: Replace "two" by "three",
   line 17: Add ", middle product" after "inner product".