No Excess Babbage - Design Considerations for the Interface to a Systolic Matrix Processor

T. Shaw
B.Sc., B.E. (Hons)

A thesis submitted to the Department of Electrical and Electronic Engineering, The University of Adelaide, to meet the requirements for award of the degree of Master of Engineering Science by research.

May 1995

Awarded 1995

Contents

List of Figures iv

List of Tables viii

Abstract ix

Statement of Originality x

Acknowledgements xi

1 Introduction 1
- 1.1 Systolic Arrays . . . 2
- 1.2 Wavefront Arrays . . . 2
- 1.3 Standard Systolic/Wavefront Arrays . . . 3
  - 1.3.1 Systolic and Wavefront SIMD Array Processors . . . 5
  - 1.3.2 Multi-bit SIMD Processing Arrays . . . 7
  - 1.3.3 Configurable Processing Arrays . . . 8
- 1.4 Conclusion . . . 9

2 The System 10
- 2.1 The University of Adelaide Systolic Processing Array . . . 10
  - 2.1.1 The Basic Architecture . . . 10
  - 2.1.2 The Outer Product . . . 12
  - 2.1.3 Interfacing to the Processing Array . . . 13
  - 2.1.4 Processing Element Redundancy . . . 14
- 2.2 Evolution of the System . . . 16
- 2.3 Conclusion . . . 17

3 Algorithms 20
- 3.1 Memory Subsystem: An Overview . . . 20
- 3.2 Matrix Multiplication . . . 20
  - 3.2.1 Partitioning . . . 21
- 3.3 Solution of Sets of Linear Equations . . . 24
- 3.4 Gauss-Jordan Elimination . . . 24
  - 3.4.1 Block Version . . . 24
  - 3.4.2 Determining the Inverse of the Pivot Block . . . 25
  - 3.4.3 Implementation on the Proposed Memory Architecture . . . 26
  - 3.4.4 Inverting Rather Than Solving a Set of Matrix Equations . . . 28
  - 3.4.5 Solving Sets of Equations That Don't Fit Into the Cache . . . 29
  - 3.4.6 Iteratively Improving a Set of Solutions . . . 30
- 3.5 The Discrete Fourier Transform . . . 31
  - 3.5.1 Implementation . . . 33
- 3.6 The Kalman Filter . . . 34
- 3.7 Conclusion . . . 37

4 Arithmetic **38**
4.1 Signed-Digit, or Carry-Propagation Free, Arithmetic 38
4.1.1 Why SD Arithmetic 39
4.1.2 Sign Magnitude Specifics 40
4.1.3 Conversion To and From Signed Digit and Two's Complement 41
4.1.4 Signed Digit Implementation 43
4.1.5 VLSI Layout and Implementation 48
4.2 Byte Divider 56
4.2.1 Newton-Raphson Iterative Divider 58
4.2.2 Byte Divider 63
4.3 8 × 8 Multiplier 66
4.3.1 Pipelined Multiplier 70
4.4 Conclusion 76

5 Address Generator **77**
5.0.1 Matrix Addressing 78
5.0.2 Parallelizing and Expanding the Difference Engine 80
5.0.3 Example Mappings on the Difference Engine 81
5.1 Address Generator Components 81
5.1.1 Sign Detector 81
5.1.2 Initialisation 84
5.1.3 Dimension Counter 84
5.1.4 Complete Decrement Units 89
5.1.5 Multiplexer Selection 91
5.2 Implementation 93
5.3 After Calculating the Offset 101
5.4 Conclusion 102

6 Memory Interface **103**
6.1 Introduction 103
6.2 Applying and Extracting the Outer Product 103
6.2.1 Extracting the Outer Product 103
6.3 Memory Overview 104
6.3.1 Difference From Vector Memory Systems 104
6.4 Cache 106
6.4.1 Cache Size 106
6.4.2 Block Size 106
6.4.3 Associativity 107
6.4.4 Latency After a Miss 108
6.4.5 Write Policy 109
6.4.6 Implementation 109
6.5 Main Memory 111
6.5.1 Implementation 111
6.5.2 Synchronous DRAMs 114
6.5.3 Bank Size 115
6.6 Bus Exchangers 116
6.6.1 Control 117
6.6.2 Read/Write Path 118
6.6.3 The Complete Bus Exchanger 119
6.7 Loading Double-length Words 123

7 Performance Estimates **125**
7.1 Matrix Multiplication 125
7.2 Gauss Jordan Elimination and Inversion 130
7.2.1 Gaussian Elimination for Matrices Smaller than the Cache Size 130
7.2.2 Gaussian Elimination for Matrices Larger than the Cache Size 134
7.2.3 Conclusion 136
7.3 Discrete Fourier Transform 136
7.3.1 Discrete Fourier Transform Without Multi-level Memory System 138
7.4 The Kalman Filter 142
7.4.1 Small Problem Size Running From SRAM 145
7.4.2 Large Problem Size Running From SRAM 146
7.4.3 Kalman Filter With a Multi-level Memory Sub-system 148
7.5 Conclusion 152

8 Future Projects **153**
8.1 A Multi-Processor Teraflop Engine 153
8.1.1 The Hypercube 153
8.1.2 Fibre-Optical Interconnection 154
8.1.3 Utilising the Hypercube - the Proposed Model 155
8.1.4 Algorithms 156
8.2 Wavelet Processor 163
8.2.1 Still Picture Compression 169
8.2.2 Moving Picture Compression 169
8.3 Conclusion 169

9 Summary and Conclusion **171**

Appendices

A SCAP Data Sheets **173**

B Proof of Convergence for Iterative Matrix Inversion **179**
B.1 Initialising the Pivot Inverse Iteration 179

C Broadcasting Data on a Hypercube **182**
C.1 Simple Broadcast 182
C.2 Pipelined Broadcast 183
C.3 Parallel or Rotated Broadcast 184
C.4 Pipelined and Rotated Broadcast 185

Bibliography **187**

List of Figures

1.1 Basic Systolic Array 2
1.2 Timing Ring Diagram for a) Systolic Array b) Wavefront Array 3
1.3 Linear Systolic Array for Convolution 4
1.4 Array for the Direct Solution of a Set of Linear Equations 4
1.5 MPP Architecture 5
1.6 DAP Architecture 6
1.7 Saxpy-1 Block Structure 7
1.8 Processing Element Structure 8
1.9 Warp Architecture 9

2.1 Processing Element Structure 11
2.2 Nibble (Digit) Skewing Input Data 12
2.3 a) Inner Product b) Outer Product 12
2.4 Block Diagram of Data Controller 14
2.5 Block Diagram of System 15
2.6 Redundant Array a) Before Failure b) After Failure 15
2.7 Redundancy Using Data Controller 16

3.1 Block Diagram of System 21
3.2 Four Regions of a Partitioned Matrix Product 23
3.3 Memory Access for Matrix Product 23
3.4 Row Oriented Gauss-Jordan Elimination 25
3.5 Extracting a) Standard b) Transpose 26
3.6 Column Oriented Gauss-Jordan Elimination 28
3.7 Dividing Sets of Equations Into Blocks 30
3.8 3 Dimensional Representation of Prime Factor DFT 34

4.1 Sign-Digit Addition Cell. 39
4.2 Sign-Digit Propagation Path. 40
4.3 Converting Positive 2's Complement to Signed Digit. 42
4.4 Converting Negative 2's Complement to Signed Digit. 42
4.5 Converting Signed Digit to 2's Complement. 43
4.6 a) b) 46
4.6 c) d) 46
4.6 Karnaugh Maps for First Digit Cell of an Adder 46
4.7 a) b) 47
4.7 c) 47
4.7 d) 47
4.7 Karnaugh Maps for First Digit Cell of an Adder 47
4.8 a) b) 47
4.8 c) d) 47
4.8 Karnaugh Maps for Last Digit Cell of an Adder with Single Input 47
4.9 Unlatched Signed-Digit Adder Cell 49
4.10 HSpice Plots for Unlatched Adder Cell 49
4.11 Layout of Latch Cell 50

4.12 Latched Signed-Digit Adder Cell 50
4.13 HSpice Simulation for Latched Adder Cell 51
4.14 Latched Adder Cell With Multiplexer 51
4.15 Layout of Unlatched Beginning Cell 52
4.16 HSpice Simulation Results for Unlatched Beginning Cell 52
4.17 Layout of Latched Beginning Cell 53
4.18 Layout of Latch & Multiplexed Input Beginning Cell 53
4.19 a) Unlatched End Cell With Standard Propagated Outputs 54
4.19 b) Unlatched End Cell With Sign Magnitude Progopated Outputs 54
4.20 a) Unlatched End Cell With Standard Propagated Outputs Simulation Results 55
4.20 b) Unlatched End Cell With Sign Magnitude Progopated Outputs Simulation Results 55
4.21 a) Latched End Cell With Standard Propagated Outputs 56
4.21 b) Latched End Cell With Sign Magnitude Progopated Outputs 56
4.22 a) Latched End Cell With Standard Propagated Outputs Simulation Results . 57
4.22 b) Latched End Cell With Sign Magnitude Progopated Outputs Simulation Results 57
4.23 Error graphs for a) $0.5 < Q \leq \frac{2}{3}$ b) $\frac{2}{3} < Q \leq 1$ 60
4.24 Error graph over full range of Q 61
4.25 Difference in Errors for Two Schemes 61
4.26 Wallace-tree Multiplier. 67
4.27 Signed Digit Multiplier. 67
4.28 VLSI Layout of Digit Multiply Cell 69
4.29 VLSI Layout of 8×8 Non-pipelined Multiplier 69
4.30 a) b = 1 b) b = 64 70
4.30 ci) b = 141 cii) b = 141 70
4.30 ciii) b = 141 d) b = 255 71
4.30 dii) b = 255 diii) b = 255 71
4.31 Processing Cell Using Pipelined Multiplier 74
4.32 VLSI Layout of Pipelined 8×8 Multiplier 75
4.33 HSpice Simulation Plots a) Input b) Output 75

5.1 Lattice Representation of a Matrix Product 79
5.2 2D Addr Gen. 79
5.3 4D Addr Gen. 81
5.4 Sign Detect Unit Cell 82
5.5 Sign Detect Unit 84
5.6 Domino Logic Exclusive-OR Gate 85
5.7 Flow Diagram for Decrement Cell 87
5.8 Decrement Cell 87
5.9 Full Decrement Cell 88
5.10 Four Input Zero Detection Cell 89
5.11 Four-Bit Decrement Unit 90
5.12 Simulation Results of Four Bit Decrement Cell 90
5.13 Sixteen-Bit Decrement Unit 91
5.14 HSpice Simulation Results for Sixteen Bit Decrementer 92
5.15 Delta Register Multiplexers 92
5.16 VLSI Layout of an Eight-bit Address Generator 93
5.17 a) D0 to D3 b) D4 to D7 95
5.17 c) Q0 to Q3 d) Q4 to Q7 & Neg 95
5.17 Simulation of Normal Address Generation 95
5.18 a) D0 to D3 b) D4 to D7 97
5.18 c) Q0 to Q3 d) Q4 to Q7 and Neg Out 97
5.18 Simulation of Transposed Address Generation 97
5.19 a) D0 to D3 b) D4 to D7 99

5.19 c) **Q0 to Q3** d) **Q4 to Q7 and Neg Out** 99
5.19 **Simulation of Prime Factored Address Generation** 99
5.20 Extended Address Generator 101

6.1 **Extraction Hardware** 104
6.2 **Vector Memory Architecture.** 105
6.3 **Non-uniform Stride Conflict.** 106
6.4 **Data Controller / Cache Memory pair** 109
6.5 **Internal Structure of** $1M \times 4$ **Static Column DRAM** 112
6.6 **Block Diagram of Four Way Interleaved DRAM Subsystem** 113
6.7 **Initial Timing of Burst Read** 113
6.8 **Synchronous DRAM Internal Block Diagram** 114
6.9 **Memory Bank Configuration using a) Thin & b) Wide DRAMS** 115
6.10 **Initiating a Read Simultaneously to Completing a Write** 116
6.11 **a) Latch Cell** **b) Driver Cell** 116
6.12 **VLSI Layout of the Driver/Transmission Cell** 117
6.13 **Four-to-One Multiplexer/Demultiplexer** 119
6.14 **VLSI Section of Four-to-One Mux/DeMux** 119
6.15 **Bus Exchanger VLSI Layout** 120
6.16 **Latched Read With Constant Select Signals** 121
6.17 **Latched Read With Varying Select Signals** 122
6.18 **Latched Write With Varying Select Signals** 122
6.19 **Transparent Read With Varying Select Signals** 123
6.20 **Transparent Write With Varying Select Signals** 123

7.1 **Performance Figures for Simple Multiplication Model (MFlops)** 126
7.2 **Performance for a)Array Dimension = 64 b) Load/Store Time = 5ns (MFlops)** 126
7.3 **Performance for Array Dimension = 64 and Load/Store Time = 5ns (MFlops)** 127
7.4 **a) Block Size = 4** **b) Block Size = 16** 129
7.4 **c) Block Size = 64** **d) Block Size = 512** 129
7.4 **Performance Estimates for Matrix Multiplication Without Streaming** 129
7.5 **a) Block Size = 4** **b) Block Size = 16** 131
7.5 **c) Block Size = 64** **d) Block Size = 512** 131
7.5 **Performance Estimates for Matrix Multiplication With Streaming** 131
7.6 **a) Block Size = 4** **b) Block Size = 16** 133
7.6 **c) Block Size = 64** **d) Block Size = 512** 133
7.6 **Performance for Gauss-Jordan Elimination on Matrices that Fit Into Cache** . 133
7.7 **a) Block Size = 4** **b) Block Size = 16** 135
7.7 **c) Block Size = 64** **d) Block Size = 512** 135
7.7 **Performance for Gauss-Jordan Inversion on Matrices that Fit Into Cache** . . . 135
7.8 **a) Block Size = 4** **b) Block Size = 16** 137
7.8 **c) Block Size = 64** **d) Block Size = 512** 137
7.8 **Performance for Gauss-Jordan Inversion on Matrices that Do Not Fit Into Cache**137
7.9 **Performance of Two Dimension DFT** - FLOPs vs DFT length 140
7.10 **Performance of Partitioned Two Dimension DFT** - FLOPs vs DFT length 140
7.11 **Performance of Partitioned Two Dimension DFT** - FLOPs vs DFT length 142
7.12 **Performance of Three Dimension DFT** - FLOPs vs DFT length 143
7.13 **Performance of Four Dimensional DFT** - FLOPs vs DFT length 144
7.14 **10ns Cycle Time** a) **Array Size = 32** b) **Array Size = 64** 146
7.14 **20ns Cycle Time** c) **Array Size = 32** d) **Array Size = 64** 146
7.14 **Simple Kalman Filter for State Vector Smaller Than Array Dimension** 146
7.15 **10ns Cycle Time** a) **Array Size = 32** b) **Array Size = 64** 149
7.15 **20ns Cycle Time** c) **Array Size = 32** d) **Array Size = 64** 149
7.15 **Simple Kalman Filter for State Vector Larger Than Array Dimension** 149
7.16 a) **Block Size = 4** b) **Block Size = 16** 151
7.16 a) **Block Size = 64** b) **Block Size = 512** 151

7.16 **Lower Bound on Kalman Filter With Multilevel Memory** 151

8.1 **Node Labelling for a 3-Cube** 154
8.2 **a) Extended Simple Memory Structure b) X-Connect Memory Structure** . . . 156
8.3 **Processing/transmission Diagram for 1st Partitioning** 157
8.4 a) **First Multiplication Phase** 158
8.4 b) **Second Multiplication Phase After First 'Roll'** 158
8.5 a) **100ns Internode Delay** b) **1000ns Internode Delay** 161
8.5 c) **10000ns Internode Delay** 161
8.5 **Performance Estimates for Multiprocessor Multiplication** 161
8.6 **Distributed Row-wise Gauss-Jordan Elimination** 161
8.7 a) **Latency = 100ns** b) **Latency = 1000ns** 164
8.7 c) **Latency = 10000ns** 164
8.7 **Performance Estimates for Multiprocessor Gauss-Jordan Elimination** 164
8.8 **Four Coefficient Compact Wavelet Transformation Matrix** 165
8.9 **Pyramidical Transform and Permute Procedure** 166
8.10 **Cell from a Linear Array for the Wavelet Transform** 166
8.11 **Linear Array for n-dimensional Wavelet Transform** 167
8.12 **Single Cell for Wavelet Implementation** 168

9.1 **Complete System Block Diagram** 172

C.1 **Spanning Tree for a 4 Dimensional Hypercube** 183
C.2 **4 Rotated Spanning Trees for 4-Dimensional Hypercube** 186
C.3 **3-Dimensional Edge-disjoint Spanning Binary Tree** 186

List of Tables

2.1 Contemporary Systolic Arrays in SIMD Configuration 17
2.2 Contemporary Systolic Arrays in MIMD Configuration 18
2.3 Comparison of Vector Computers in the 1980's 18
2.4 Comparison of Vector Computers in the 1990's 19
2.5 Comparison of Massively Parallel Computers 19

3.1 Ordering when Number of State Vectors is Small 36
3.2 Ordering when Number of State Vectors is Large 36

4.1 Table of SD representation 41
4.2 Table of SM representation 41
4.3 Digit Outputs for Unlatched Multiplier, b=1 72
4.4 Digit Outputs for Unlatched Multiplier, b=64 72
4.5 Digit Outputs for Unlatched Multiplier, b=141 73
4.6 Digit Outputs for Unlatched Multiplier, b=255 73

5.1 2 Dimensional Mappings on Difference Engine 82
5.2 4 Dimensional Mappings on Difference Engine 83
5.3 Decrement Cell Combinations 85
5.4 a) S_{out} & M_{out} b) *Neg_Out* 86
5.5 Resultant Addresses for Normal Addressing Data 96
5.6 Result of Simulation of Addressing for Transposed Access 98
5.7 Result of Simulation of Addressing for Prime Factored Access 100

6.1 Tag Requirements 110

7.1 2 Dimensional DFT Performance 139
7.2 Partitioned 2 Dimensional DFT Performance 141
7.3 Partitioned 2 Dimensional DFT Performance 141
7.4 Three Dimensional DFT Performance 143
7.5 Four Dimensional DFT Performance 144

8.1 Parallel Multiplication Allocation Table 159
8.2 Wavelet Products versus Time 168

Abstract

Computational systems used for the solution of large matrix-based numerical problems are rapidly converging on the limits of technology, and novel architectures are now being sought to improve performance. In signal processing and control theory, high performance systems are required which do not contain the physical size penalty of current supercomputers. To achieve performance comparable with current supercomputers, a systolic processing array specifically targeted at matrix applications has been developed at the University of Adelaide.

The work of this thesis involves the problem of delivering and receiving the data moving between the processing array and the memory subsystem. This involves the reformatting of existing algorithms to map efficiently onto the matrix array, the design and VLSI layout of a matrix address generation unit using signed digit arithmetic for enhanced performance, and the block level description of a multiport cached memory system. Performance estimates predict a modest configuration will perform selected matrix routines in excess of three GigaFLOPs.

Statement of Originality

Acknowledgements

I would especially like to thank my supervisor Mr Michael Liebelt for his direction, advice and support throughout my work.

The work undertaken for the degree is based around a project at the Univerisity of Adelaide which has been largely carried out by Dr Warren Marwood. Dr Marwood has provided a great deal of knowledge, enthusiasm and encouragement to me over the past two years, and I extend my grateful thanks to him as well.

The advice of Dr C.C. Lim and Dr N. Burgess has been of great benefit, and led me into areas of great interest.

Thanks also go to the group of other postgraduate students with whom I have worked, for their comments and thoughts. Particularly to David Standingford, Richard Beare, Ali Moini, Andrew Blanksby and the HiPCAT team.

Finally, I would like to thank my parents, for the support they have always given me in whatever I wished to do, and to Alison, for all her enthusiasm.

Chapter 1

Introduction

This thesis describes some design considerations made during a project at the University of Adelaide involving the design of a Systolic Processor and Interface. The processor is a specialized architecture intended for use as a generic matrix processing engine, comprising a large number of processing elements configured in a two-dimensional array, and some support hardware. The processor elements themselves are kept very simple, with more complex operations taking place in the support hardware. On its own, the processing array performs the simple operation of a *matrix outer-product*, which can be used to perform other algorithms by clever addressing and application of data to the array.

The work presented in this thesis is the culmination of several design 'tasks'. Briefly, they are:

1. the development and redesign of various matrix algorithms to map efficiently to the defined matrix array structure. This process indicated what hardware features would significantly enhance or reduce the system performance.

2. the VLSI design of the address generation hardware, which is the critical path in the processing-array/memory interface. To maximize the performance of this critical component, non-conventional arithmetic was used to implement an expanded Marwood Difference Engine.

3. the specification of a memory subsystem. The organisation of the memory system and its inter-relationships with the algorithms significantly affects the overall system performance. Therefore, a memory system that can support the matrix processing array at nearly peak speed was designed to extract maximum performance while still retaining relatively moderate cost.

Additionally, a multiprocessor architecture with a matrix processor as each node is proposed and its feasibility investigated.

The thesis is organised around the work in such a way that each distinct task has its own chapter. However, as the tasks are heavily inter-related, there is some overlap among the chapters. The tasks occupy Chapters 3 to 8. Chapter 2 introduces the architecture of the system design to the time of the start of this project, while conclusions of this project are made in Chapter 9. The rest of this chapter is devoted to an brief introduction to systolic and wavefront systems.

1.1 Systolic Arrays

Originally proposed by H.T.Kung and C.E.Leiserson in 1978 and described in the landmark text "Introduction to VLSI Systems" [66], the term 'systolic' refers to the systolic cycle of the heart. The systole is the contraction of the heart and arteries to expell blood and pass it throughout the body. Similarly, a systolic computer system is one in which data is 'pumped' through the systolic hardware or array, and is operated on by several sections of the hardware in turn. A basic system of this form is shown in Figure 1.1. Data moves from the memory and is passed to each processing cell in turn until it is finally returned to memory. As can be seen from the diagram, data is utilised by each cell (in this case six times) for the cost of only one memory load and one memory store operation. Obviously, for a memory bandwidth limited system, the speed-up of a linear systolic system is equal to the number of cells in the array.

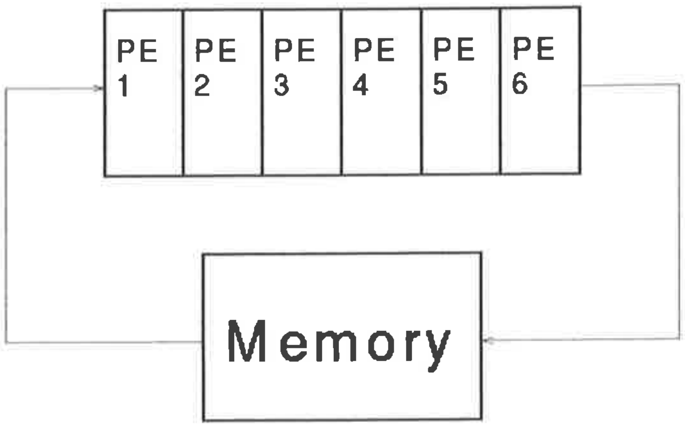


Figure 1.1: **Basic Systolic Array**

Systolic arrays have been constructed in linear, rectangular, triangular and hexagonal configurations, to name a few, to solve various specific problems, such as Gaussian analysis, LU decomposition radar adaptive beam-forming and data formatting.

The properties of a systolic array are that the array is constructed of cellular processing units, that can be theoretically repeated indefinitely in each dimension. The cells contain only *local, homogeneous* interconnections, and operate such that the array exhibits linear pipelinability, ie that the array can be pipelined into as many stages as there are processing cells.

1.2 Wavefront Arrays

S.Y. Kung [49] distinguishes between systolic arrays and wavefront arrays by the single feature of a global clock. The global clock is required for the pumping action of data through a pipeline, as each cell passes data on to the next at cycle intervals. However, if there is no global clock, and the data operations are locally timed, data flow techniques become applicable, and the array takes on a *wavefront* approach, ie data is processed in waves that propagate through the array using local synchronization. Kung reproduces diagrams from a report by Q.E. Dolecek entitled "Parallel processing systems for VHSIC." that show the difference [49, 22]. These are shown in Figure 1.2a) for a systolic processor and Figure 1.2b) for a wavefront processor.

Kung goes on to compare the wavefront and systolic structures and draws the conclusion

$$\textit{Wavefront array} = \textit{Systolic array} + \textit{data flow computing}$$

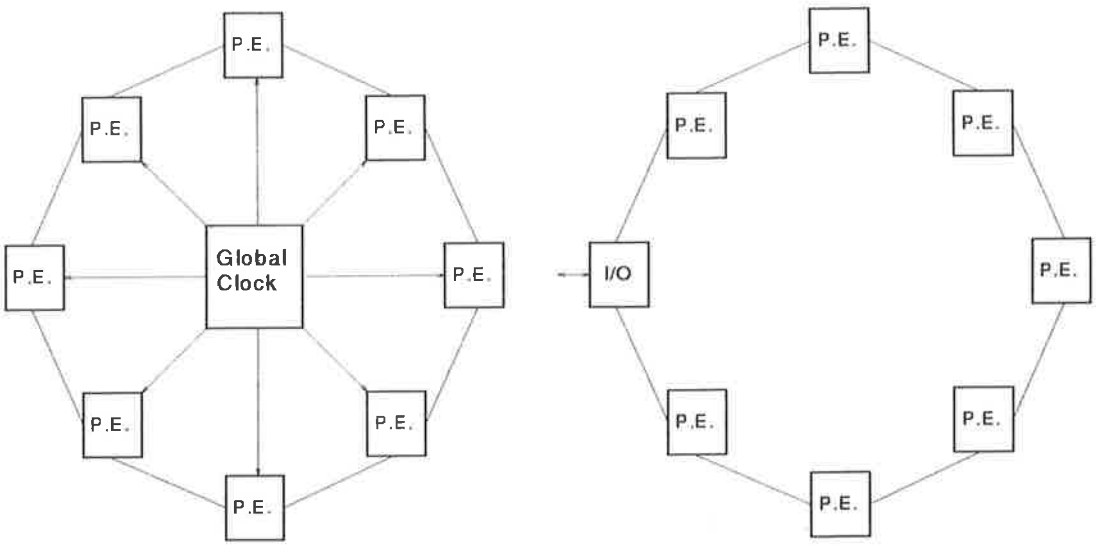


Figure 1.2: **Timing Ring Diagram for** a) **Systolic Array** b) **Wavefront Array**

Note that a wavefront array that uses a local clock synchronised to an adjacent processor could be considered to be either a wavefront or a systolic array, depending on the interpretation of the clock signal. For example, the local clock could be synchronised with and derived from an input clock, and an output clock could be the local clock delayed by one cycle. Thus, the output clock could be considered a data-flow timing parameter, as it is derived locally and there is no global propagation parameter. However, as the output clock is derived from the input clock which is ultimately applied to all the edge cells of the array, the output clock could be considered to be a global clock. It is just a matter of interpretation as to which class some processing arrays should be classed.

1.3 Standard Systolic/Wavefront Arrays

Initially, systolic arrays implemented structures designed to solve specific algorithms. Dedicated to their task, they were of little use for anything but the task they were designed to achieve. Some examples are shown below.

1. Convolution
   Given two sequences of numbers u_j and w_j for $j = 0, 1, \ldots, N-1$, then the linear convolution of the two sequences is defined by [49]

   $$y_j = \sum_{k=0}^{j} u_k w_{j-k} \quad j = 0, 1, \ldots, N-1 \qquad (1.1)$$

   A linear systolic array that implements this algorithm is shown in Figure 1.3.

2. Direct Solution of the Linear System $Ax = b$
   Using the Gauss-Jordan elimination scheme, the direct solution of a set of linear equations can be formed. Figure 1.4 shows an array that implements this algorithm [76]. Data inputs are propagated through the array together with a 'control' word that determines the operation to be undertaken at the processing cell. The cells can become quite complex because of the control word and the various operations that are available, although the general interconnection scheme is only suitable for a very limited set of algorithms.

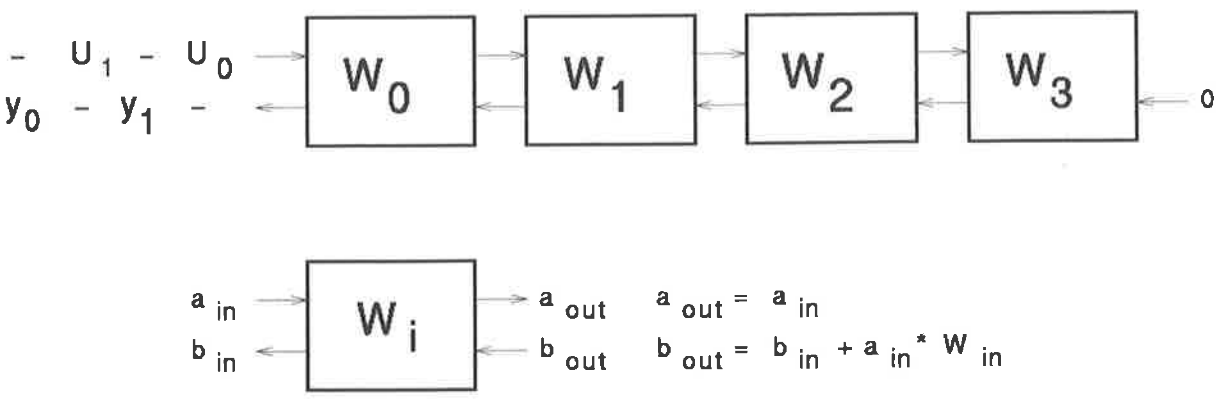


Figure 1.3: **Linear Systolic Array for Convolution**

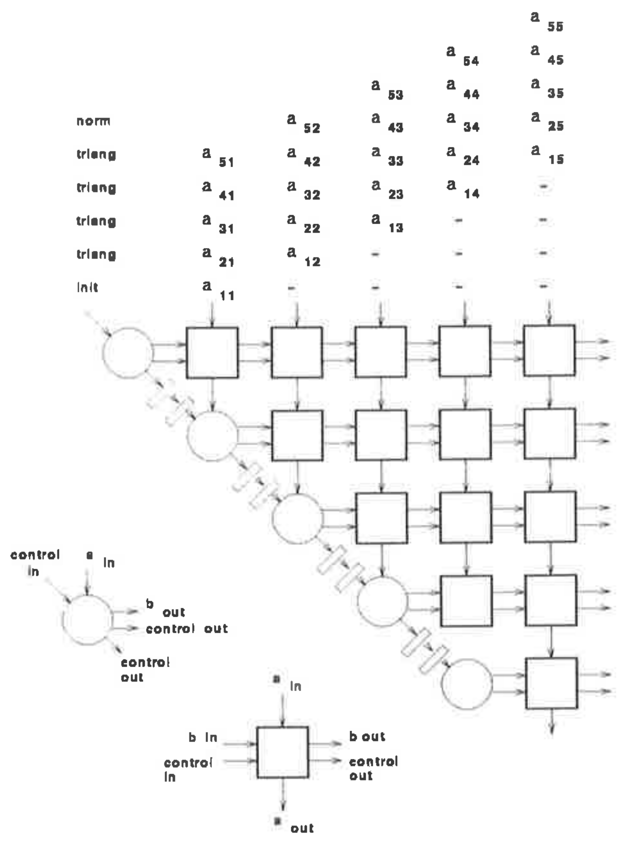


Figure 1.4: **Array for the Direct Solution of a Set of Linear Equations (from [76])**

1.3.1 Systolic and Wavefront SIMD Array Processors

The early systolic and wavefront machines were often modelled on the Single-Instruction-Multiple-Data (SIMD) model [49]. In these architectures, there is local interconnection between the processors, and between a processor and its local memory. Each instruction is globally broadcast by a host controller, and a processing cell has the choice of either executing the instruction or masking the instruction. Thus, two processing elements can not be executing different instructions during the same instruction cycle.

Initially, many of the early systolic SIMD arrays operated in bit-serial form, and were therefore referred to as Binary Array Processors (BAP) [49]. The bit-serial architecture provides for flexible data formatting, and is efficient in terms of the memory resources and chip interconnections, as a large number of processing elements fabricated on a chip will require minimal I/O interconnections. However, as available chip resources have increased with time and the memory bandwidth is now often the bottleneck of a system, there has been a decline in bit-serial processing architectures. Example BAPs include the Massively Parallel Processor (MPP), from Goodyear Aerospace, the Distributed Array Processor (DAP), from International Computer Limited, and the Geometric-Arithmetic Parallel Processor (GAPP), from NCR [38, 49, 24].

The Massively Parallel Processor

Commissioned by NASA for the processing of satellite images, and running considerably over budget in the process, the MPP comprises 16,384 bit-serial processing elements, configured as a 128 x 128 array. An additional four rows of processing elements are included for redundancy in the event of failure, so the actual array is in fact 128 x 132, although the user only sees the square 128 x 128 array. The PEs in the MPP can accept inputs from one of several nearest neighbours and allows overlapping data I/O and array operations. The PEs are custom designs, with eight PEs on a chip. Each chip has access to fast local RAM, although the memory interface still represents the performance bottleneck due to the speed of the custom circuits. A block diagram of the MPP is shown in Figure 1.5, from [80].

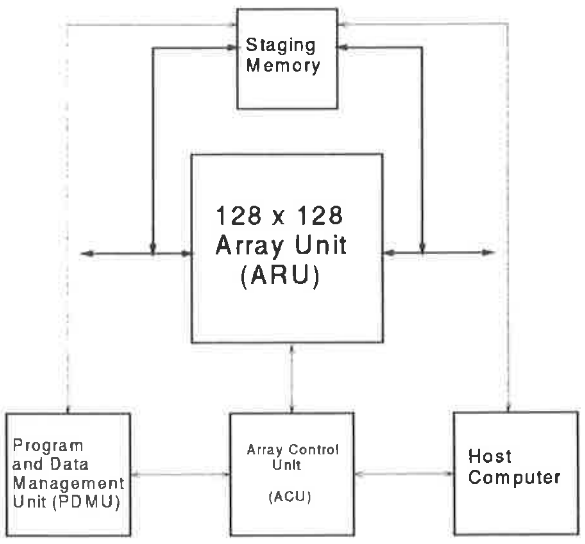

Staging Memory
128 x 128 Array Unit (ARU)
Program and Data Management Unit (PDMU)
Array Control Unit (ACU)
Host Computer

Figure 1.5: **MPP Architecture (from [80])**

The processing elements have no built in fixed or floating point arithmetic, only a full adder. Thus, the range of applications that can be run on the MPP is limited to those that do not require complex arithmetic to run efficiently. The MPP provides working performance of up to seven billion additions per second [80].

Distributed Array Processor

The DAP is constructed of blocks of 16 PEs, in various two dimensional sizes, such as 32 × 32, 64 × 64, etc. A DAP PE combines 4096 bits of RAM with a bit-serial adder, and local interconnections to the nearest four neighbours. The local interconnections combine to form a grid, along which arbitrary data movement is possible (within the confines of the SIMD array architecture). Therefore, data shifting becomes very easy. A diagram adapted from Hockney and Jesshope [37] is shown in Figure 1.6.

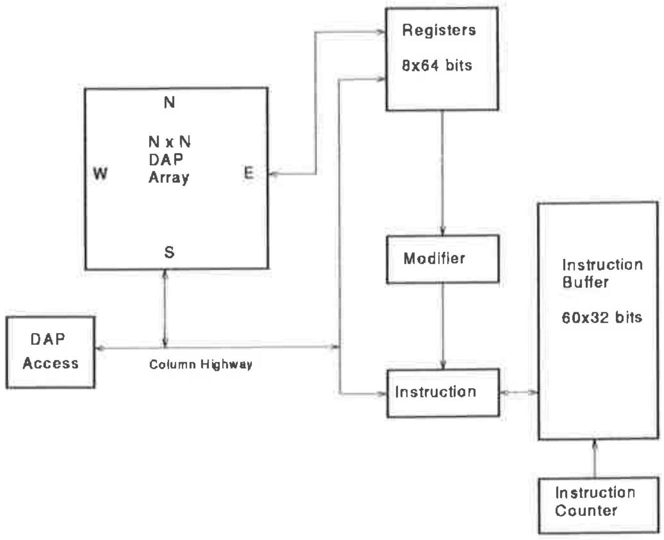


Figure 1.6: **DAP Architecture (from [37])**

Geometric Arithmetic Parallel Processor

The Geometric Arithmetic Parallel Processor (GAPP) is aimed at low precision image processing, and as such uses very simple processing primitives. The GAPP processor cascades an arbitrary number of chips, each containing an array of 6×12 processing elements (PEs). A commercial array typically will contain approximately 2304 PEs, configured as an array of 48 × 48 PEs. Each PE contains an ALU, 128 bits of RAM, and local interconnection to each of the PE's four nearest neighbours. Each instruction is broadcast to the complete array, in the standard SIMD form. The ALUs are single-bit processing cells.

The GAPP structure is very similar to that of the DAP, although there is much less memory made available to each PE (128 bits compared to 4096 bits). This results in a very high chip density, and so a large number of chips can be packed into a small area. However, as the GAPP PEs were designed for use in image processing and low precision applications, the PE's resources would rapidly be exhausted in the event that a numerically intensive algorithm, such as boundary integral evaluation, were attempted. The problem could still be feasibly run on a DAP system with greater system resources.

1.3.2 Multi-bit SIMD Processing Arrays

The Cellular Array Processor

The Cellular Array Processor (CAP) [49] was developed by Fujitsu, and includes a multibit ALU that can perform block floating point operations on the applied data. Each PE is more complex than those of the DAP and GAPP and only twenty will fit on a chip, even though the CAP was implemented using a $1.25\mu m$ process instead of the $3\mu m$ process used for the GAPP. Of the twenty PEs, only sixteen are used at any one time. The remaining four are used to provide fault-tolerance in the system, so that up to 25% of the PEs can fail on a chip and the operation of the system will show no apparent change. Each PE incorporates 4000 bits of memory, and more can be added externally if required.

Not only are the arithmetic units of the CAP processor more complex than those of the DAP and GAPP systems, the control is more complex as well. The CAP processing elements can control three independent functions of the PE, which are the arithmetic/logic function, the memory/common bus function and the I/O function. Thus, the CAP is aimed at more computationally- and I/O- sophisticated problems that exhibit a large degree of parallelism, such as parallel searches and Kalman filtering.

The Saxpy-1

The Saxpy-1 is a vector architecture that uses systolic principles to obtain fast inter-processor communication. It was developed by Saxpy Computer Corp in the mid to late eighties. The system is composed of five major segments, shown in Figure 1.7. The peak performance is estimated at approximately 1000 MFLOPs.

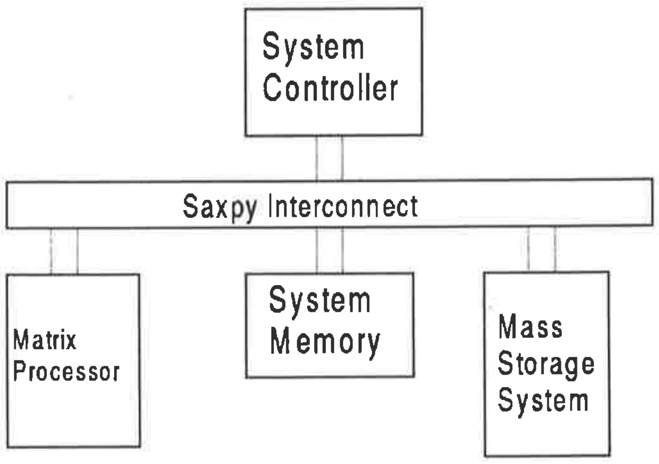


Figure 1.7: **Saxpy-1 Block Structure**

The five segments are:

- The System Controller. This a general purpose computer, such as a DEC VAX, running VMS. The System Controller is used to compile and link the application program, and coordinate the allocation of resources.
- Matrix Processor. This is a linear array of up to 32 pipelined, floating-point processors that have systolic and global interconnections. Although global interconnections are against the philosophies that drove initial systolic designs, the practicalities of the Saxpy-1

implementation justify this departure [27]. A diagram of the Matrix Processor segment and the arrangement of the 32 processing elements is shown in Figure 1.8

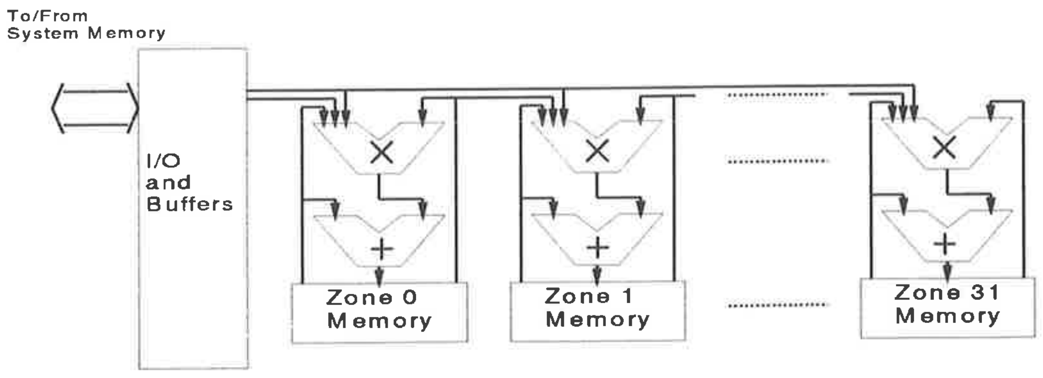


Figure 1.8: **Processing Element Structure**

- System Memory, which stores all the data arrays for use by the matrix processor.
- Mass Storage System, an I/O interface that provides access to high speed data-storage peripherals
- Saxpy Interconnect, a bus containing both data and control that links the other four units of the system.

The only part of the Saxpy-1 that uses systolic techniques is the Matrix Processor, which operates as a SIMD systolic machine. None of the processing elements possess independent program code, and program control ultimately resides with the host computer or workstation.

1.3.3 Configurable Processing Arrays

With the very large range of systolic algorithms that have been developed, attempts have been made to design systolic systems that can be reconfigured to optimally suit several algorithms. Examples of these include the 'Configurable Highly Parallel Computer' (CHiP)[87] and the 'Programmable Systolic Chip' (PSC) (from Marwood[58], [26]), although probably the best known architecture of this class is the (i)Warp machine, which was designed and developed by Carnegie-Mellon University and various industry partners [49].

The Warp system is composed of an array of ten or more programmable processing cells, each capable of 10 MFLOPS performance, together with a host MC68020-based machine and a host/array interface unit. A block diagram of the Warp is shown in Figure 1.9. Data flows through the array onpaths labelled X & Y, with a third path allocated to address and control signals.

Each processing cell contains its own program store and sequencer, a 5MFLOP multiplier, a 5MFLOP ALU, a 64 Kword memory, communication interface and a register file. The cells can operate on the data with IF-THEN-ELSE and DO-WHILE structures, and must each be programmed separately. More complex operations such as divide and square root are left to specialised 'Boundary Processors' (BP), which can be attached to one end of the array. A 10-cell Warp machine is capable of computing a 1024-point complex FFT every 0.6 ms, and a 2-D cosine transform on a 256×256 image in 13mS.

The Warp system was followed by the iWarp architecture, a partnership between Carnegie-Mellon University and Intel Corp. The cell in the iWarp became a custom VLSI processor

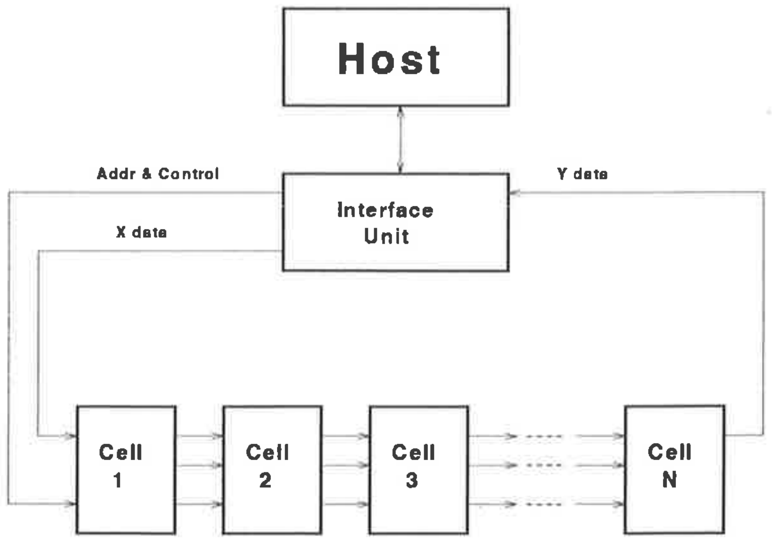


Figure 1.9: **Warp Architecture (from [49])**

consuming some 600,000 transistors for a computational output of 20 MFLOPS per processor, together with up to 64 Mbytes of memory. It can be seen, therefore, that the transistor count becomes very high for systems of any significant number of cells if the cells themselves become complex. Indeed, the increase in complexity is contrary to one of the initial aims of systolic processing as defined by H.Kung, namely the construction of arrays of small, simple cells that can be repeated across a VLSI structure.

1.4 Conclusion

The one point that becomes apparent from a study of the existing architectures is a simple conundrum involving the complexity of the cells (processing elements):

1. If the processing elements are very simple, the versatility of the processing array is greatly restricted, and so the potential market and range of applications is limited.

2. If the processing elements are too complex, then systems involving large numbers of processing elements become unwieldy, and the advantage of systolic processing is lost.

This led the designers of the processing element structure to the conclusion of keeping the processing element structure simple, while increasing the versatility of the array by using an intelligent interface between the array and the data storage to perform the more complex data and address operations at the boundary of the array.

Chapter 2

The System

2.1 The University of Adelaide Systolic Processing Array

Over the past few years, the Department of Electrical & Electronic Engineering and the Centre for Gallium Arsenide VLSI Technology, at the University of Adelaide, have been engaged in a project to implement a systolic processor for matrix computation. The array can perform the basic matrix operations of *addition, subtraction, multiplication and Hadamard multiplication*, with more complex operations provided in firmware in the array controllers [54, 7, 62, 59, 63, 55, 56, 65, 57, 5, 18]. In this way, Marwood *et al.* attempt to produce the generality desired to make a systolic architecture useful for a large range of applications, while maintaining the complexity at a level simple enough to minimise overheads and contain the cost of the large number of computational units necessary. By using novel address generation hardware, Marwood has created the generality in address mappings that are required for various systolic arrays, while the integration of computational units with more complex address generation/boundary interface data controller units ensures that the simplicity of the repeated cells is retained.

2.1.1 The Basic Architecture

The systolic processor developed by the University of Adelaide is composed of a two dimensional square array of simple processing elements. Each processing element (P.E.) is capable of performing the operations *multiply, addition & multiply-accumulate*, and contains a simple multiplier, accumulator, output register and control structure. The structure of a P.E. is as shown in Figure 2.1.

Before the data is supplied to the multiplier cell, a control word of four bits is applied to the input of the processing element. This control word defines the cell operation for the duration of the next operation, and includes information including:

- which operation to apply to the operands
- whether to reset the accumulators (perform multiply), or to leave the data in the accumulators (multiply-accumulate)
- whether to unload the accumulators, and, if so, in which direction to move the output data.

The operands supplied to the multiplier cell are broken into one nibble (four bits) digits, and supplied one at a time from the '*A in*' & '*B in*' inputs, which progressively builds the full product from the successive partial products. The input data is also passed on to the next cell in the direction of propagation. As there will be some propagation delay within the cell, the

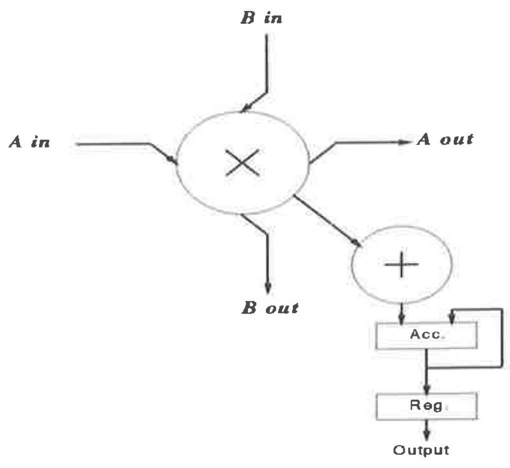


Figure 2.1: **Processing Element Structure**

data is latched before being retransmitted, thus ensuring a fixed propagation delay through a cell. As the data delayed by this amount is one nibble in size, the delay is referred to a '*nibble propagation delay*', or nibble delay.

As data is coming from two directions, the propagation delay in one dimension must be compensated for when data is applied in the other dimension. For example, if all cells on an array boundary were to receive data at the same time, then a processing element located at co-ordinate (1,4) (first row, fourth column) will receive its row data immediately and its column data after three nibble delays[1]. Therefore, the row data must be delayed by four nibble delays for proper coordination to occur. Of course, this applies equally for the processing element (4,1), in which the column data must be delayed by three nibble delays, and similarly the row data must delay applying data to the second column by one nibble delay and to the third column by two nibble delays, etc. These delays result in a skewing of the input wavefronts, by one nibble delay period, and so the inputs are said to be '*nibble skewed*'. The nibble skewing is shown in Figure 2.2. Note that the loading of adjacent words in a column for the vertical dimension or in a row for the horizontal dimension is initiated before the completion of the loading of the previous word.

By nibble skewing, the operation of the array can be approximated to be that of the calculation of an outer product. A matrix product is then a series of outer products that are accumulated within the array, and then unloaded as a batch at the end of computations. This approach requires no long bus drivers (and the inherent delay in such devices) that are needed for broadcast strategies, and hence can be scaled to an arbitrary size limited only by the speed data can be fetched from memory.

In practice, the 'arbitrary' size of the processing array is bounded on each side by practical considerations. If the processing array is too small, then the start-up overhead will be significant, and no speed improvement over a scalar processor is obtained (consider the limiting case of $n \rightarrow 1$ for an $n \times n$ array). The maximum size of the processing array is limited mainly by algorithm considerations. While a large array is very efficient if it is operating on data that 'fills' the array, the efficiency drops considerably if the problem dimension is significantly smaller than the processing array dimension (eg. a 10×10 matrix product on a 20×20 processing array only uses one quarter of the available processing elements). Simulations of algorithms have shown

[1]Not four nibble delays. Note that the first position must wait zero nibble delays

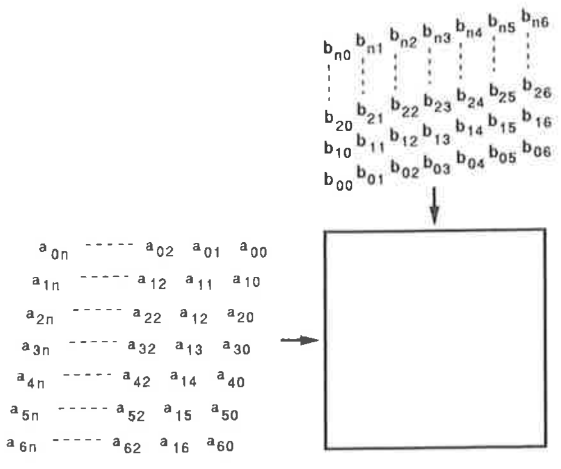


Figure 2.2: **Nibble (Digit) Skewing Input Data**

that a processing array of between 20 × 20 and 100 × 100 processing elements is practically feasible [58].

2.1.2 The Outer Product

Contrary to most matrix systolic arrays which perform matrix multiplication in the *inner-product*, our matrix processor calculates the answer in *outer-product* form. There are two reasons for this:

- Memory access patterns are simplified.This is a simple observation of the access patterns of the two configurations.

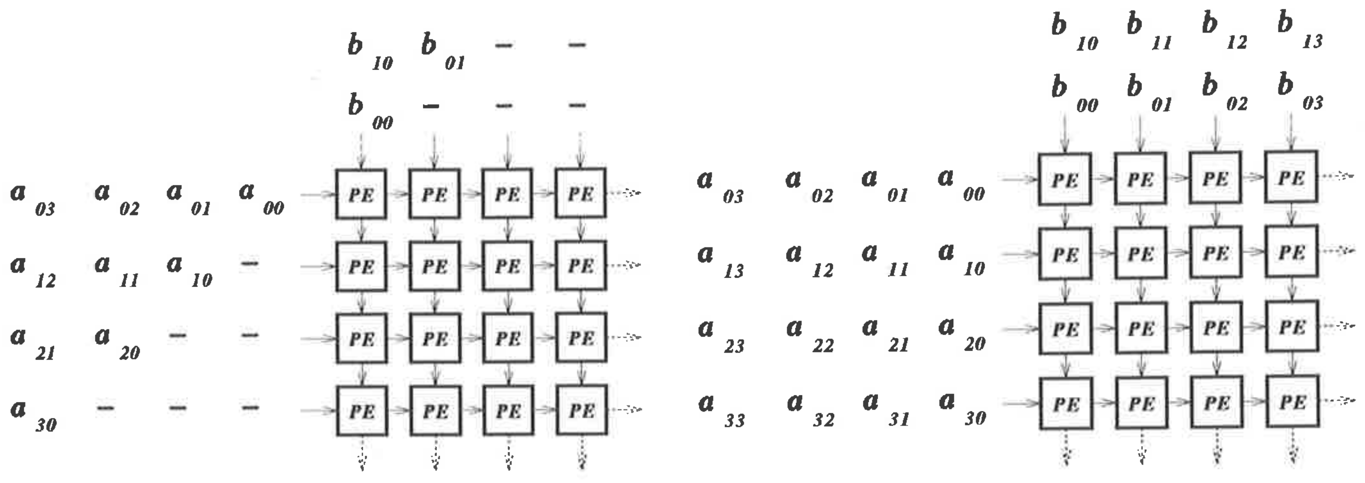


Figure 2.3: **a) Inner Product** **b) Outer Product**

Consider the inner product configuration in Figure 2.3a). If it is assumed that the matrix is stored in standard form in memory (ie a complete row or column is stored linearly in memory, followed by the next row or column) and the systolic array is of size $P \times Q$, then the access pattern to fetch a_{00}, then a_{01} & a_{10}, followed by a_{02}, a_{11} & a_{20}, as shown in the figure, will be

$0, (P, P{+}1), (2P, 2P{+}1, 2P{+}2), \ldots$

ie. an *out of order* access.

For the outer product array in Figure 2.3b), the same storage arrangement will result in the access pattern

$0, 1, 2, \ldots, P{-}1, P, P{+}1, \ldots$

which is an *in order* access. This results in fewer memory bank conflicts and a higher 'hit' rate for the caches.

- Operations on smaller matrices are more evenly balanced. This relates to the previously mentioned advantage, but is concerned more with the delay slots which must be inserted into the inner-product configuration.

  One of the motivations behind using systolic arrays is that the computational cells, or processing elements (*P.E.'s*), can be made simple and be replicated many times on a single chip. Therefore, it is not desirable to attempt to make the cells run at varying speeds.

  However, if the cells all run at the same speed all the time, only $\frac{1}{P}$ of the available bandwidth is utilised at start-up, then $\frac{2}{P}$, etc until the full bandwidth is utilized after $\frac{P^2}{2}$ accesses after the initial access. Although this may not be significant if the matrix being operated on is much larger than the array dimension P, the effect on matrices that are a similar order to the array is to double the time a computation takes. In Gaussian elimination (discussed later) operations on matrices of similar order to the processing array are common place. As the outer product configuration uses all the processing elements all the time, the full computational rate can be achieved.

2.1.3 Interfacing to the Processing Array

A complete system based around an array of processing elements will include a memory system for storing all the required operands. Therefore, a memory interface is required, that will provide the link between the array and the memory subsystem. The hardware that is used for the generation of the correct addresses, the receiving of the desired data, and other sundry functions can be incorporated into one unit, termed the '*Data Controllers*'. The functions these provide include:

- Address Generation
  The addresses of the next memory access must be calculated according to some formula. In this architecture, a difference engine is used, which calculates addresses according to the formula

  $$\textit{next addr.} = (\textit{prev addr.} + \textit{offset}) \bmod \textit{max. addr.}$$

  The address generator is dealt with in more detail in Chapter 5.

- Memory Interface
  The Data Controllers must supply the address calculated by the Address Generators to the correct memory unit, and then read the result back. In the event that a multi-level memory is implemented, the Data Controllers must also check cache memory tags before reading data to ensure that the correct data is read into the controllers. A proposed memory structure is presented in Chapter 6.

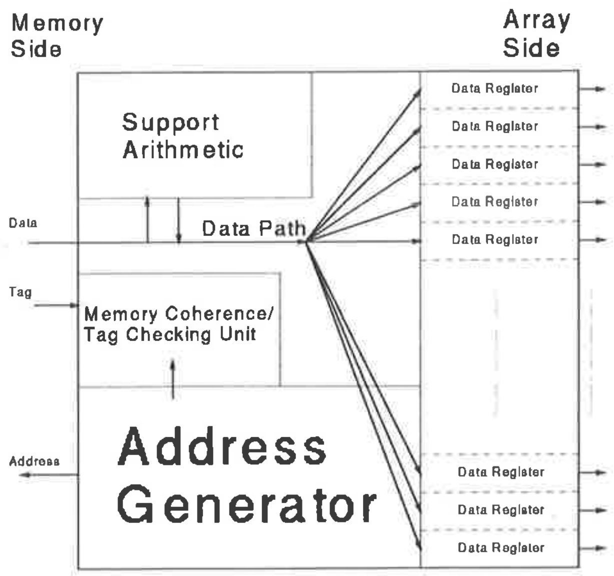


Figure 2.4: **Block Diagram of Data Controller**

- Convert and Supply Data to Array
  The data that is read from memory is supplied in 'word' form from the memory, and must be converted to nibble form and the rate of supply slowed down before it is passed to the array. This is done with a series of shift registers with parallel load and serial unload facilities.

- Support Arithmetic
  As the processing elements only perform the most basic of arithmetic operations (multiply and addition), any extra operations that are required in an algorithm must be implemented within the Data Controllers, either before the operands are applied to the array (such as row normalisation) or after they are extracted (such as trace determination).

A block diagram of a Data Controller is shown in Figure 2.4, while a diagram of the complete system is shown in Figure 2.5.

2.1.4 Processing Element Redundancy

As the typical size of a processing array will vary approximately between 20×20 to 100×100 processing elements, the total number of P.E.'s in a system will vary between approximately 400 and 10,000. With numbers this high, consideration needs to be given to the reliability of the processing elements, and hence of the complete system. Marwood and others [62, 75] have noted the importance of including redundancy into such large systems, and techniques have been devised that effectively reduce the defect rate to close to zero. These include concepts such as configuring the processing array with an extra row or column of processing elements and switching the data flow 'down' or 'across' a row or column in the event of processing element failure. Figure 2.6a) shows the a 5×4 processing array operating correctly, while Figure 2.6b) shows the same processing array reconfigured after the failure of processing element number 11. Using this approach, two extra multiplexers are required for each processing element for each dimension of redundancy, for a total of $4p^2$ multiplexers if the array is of dimension $p \times p$ and two dimensions of redundancy are implemented.

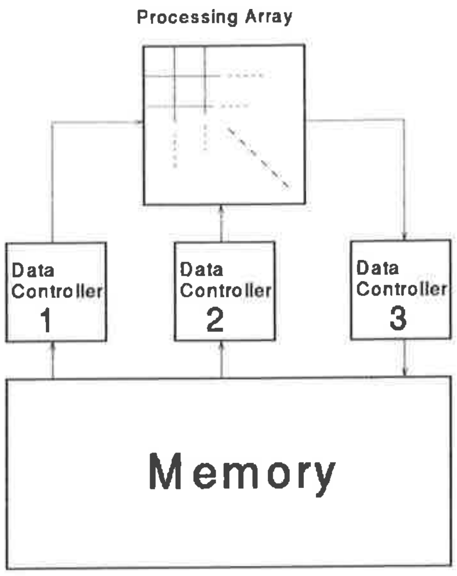


Figure 2.5: **Block Diagram of System**

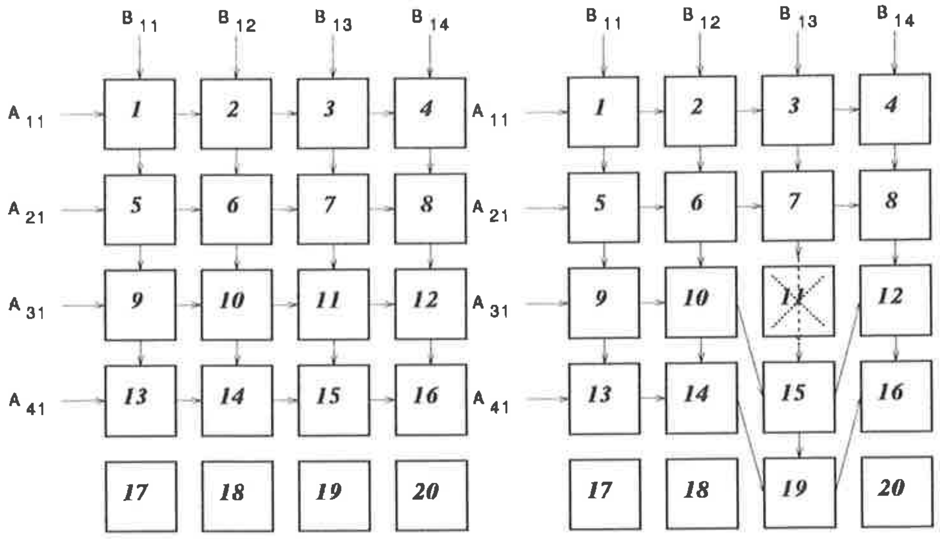


Figure 2.6: **Redundant Array** a) **Before Failure** b) **After Failure**

Similarly, failure among the processing elements can be dealt with by the data controllers, by switching in a complete new row or column fed by the controllers. This is illustrated in Figure 2.7. Although the versatility of this approach is reduced compared to placing the switching with the processing elements, the number of multiplexers is greatly reduced, to $2p$ from $4p^2$.

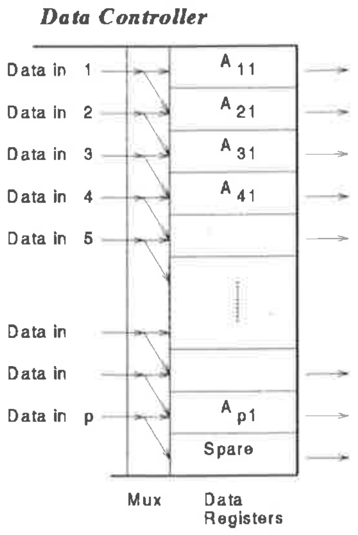


Figure 2.7: **Redundancy Using Data Controller**

2.2 Evolution of the System

The matrix engine designed at the University of Adelaide is very similar to one designed and implemented by Marwood and others at the Defence, Science & Technology Organisation at Salisbury, South Australia, called the Systolic Configurable Array Processor (SCAP) [18, 54, 55, 56, 58, 59, 61, 62, 63, 75]. Although the projects were initiated by Marwood at approximately the same time, several SCAP systems have already be constructed. The SCAP processing elements were constructed using CMOS technology, whereas the University of Adelaide design was in Gallium Arsenide, partly to aid development of that technology. The data sheets describing the SCAP architecture are included in Appendix A. The important comparison points from the data sheet of the SCAP A17502 Data Controller are reproduced below.

- Convenient interface between conventional memory architectures and a Processor Array ....
- Cascadable to service arbitrary sized processor arrays.
- Bus interface allows common or independent memory subsystems.
- Flexible matrix addressing modes are built in [to] provide zero cost matrix operations such as Transposition, Negation, Conjugation and Mapping for prime factor and other algorithms.
- Direct support for both real and complex matrices, submatrices and non-square matrices.
- Maps arbitrary sized problems onto a fixed-sized array. The problem is limited only by available host memory.

- Programmable modes of operation. The same device can be used to fetch operands or to store results.
- Executes its own instruction stream from host memory, allowing complicated algorithms to be performed without host intervention.
- Each Data Controller can fetch operands or store results at a rate of 6.25M floating point words per second.

These are the items that will be used for direct comparison with the system described in this thesis.

A range of systolic processors that are not as closely related to our system as SCAP was presented by Johnson, Hurson & Shirazi[41]. Their tables are reproduced in part in Tables 2.1 & 2.2. Some of these are briefly described in Chapter 1.

System Name, Developer	Development stage	Topology	Key Features
Brown Systolic Array per chip; Brown University	Prototype	Linear 470 cells	Very small VLSI footprint; 100s of cells ISA,SSR architecture; 8-bit ALU only; 157 MOPS
Micmacs IRISA, Campus de Beaulier	Prototype	Linear 18 cells	16-bit fixed point maths broadcast data; 90 MOPS
Geometric Arith. Par. Proc. NCR	Commercial	48 × 48 array 2304 cells	Bit-slice cellular arch.; global data; 900 MOPS
Saxpy-1M Computer Corp.	Commercial	Linear 32 cells	32-bit f.p. capability; broadcast and Saxpy global data with block processing; 1000 MFLOPs
Systolic/Cellular Arch. Hughes Research Lab.	Prototype	16 × 16 array 256 cells	32-bit fixed-function units
Cylindrical Banyan Multicomp. University of Texas at Austin	Research		Packet-switched programmable topology with programmable cells
Programmable Systolic Device ANU	Research		Instruction decoding occurs once per chip; each chip has many cells

Table 2.1: **Contemporary Systolic Arrays in SIMD Configuration**

For comparison purposes, the matrix engine can also be compared to high end 'add-on' array processors or, alternatively, with vector processors. In 1987, Dongarra produced a comparison of the characteristics of vector supercomputers in the 1980's, which was repeated in 1992 by Kahan in his examination of supercomputing in Japan in 1992 [43]. These were used by Marwood [58] as benchmarks against which he could compare the principles of systolic processing. The tables detailing the comparisons are reproduced below, as Table 2.3 and Table 2.4 respectively.

2.3 Conclusion

By performing an outer product on the array instead of an inner product, memory patterns are simplified and start-up delays are reduced. The simple processing elements can be replicated a large number of times on a single chip and many chips included on a single MultiChip-Module (MCM) to produce a large factor of parallelism while maintaining the simplicity of the simple architecture. More complex algorithms will be implemented by performing more complex tasks on an interface 'Data Controller' chip.

System Name, Developer	Development stage	Topology	Key Features
PSC CMU	Prototype	Linear 9 cells	Early predecessor of Warp; 8-bit fixed-point ALUs
Warp CMU	Commercial	Linear 10 cells	32-bit f.p. multiplication; block processing; I/O queuing; 100 MFLOPs
iWarp CMU	Prototype	8×8 array	Warp cell minus on-chip memory expandable to 1024 cells
Computer for Experimental Synthetic Aperture Radar Norwegian Defense Res. Estab.	Prototype	Four 8×16 arrays 512 cells	Bit-serial cellular I/O; 32-bit f.p. multipliers in each cell 320 MFLOPs
Cellular Array Processor Fujitsu Lab., Japan	Commercial	16×16 array 256 cells	Block processing; f.p. maths; image oriented
Configurable Highly Parallel Comp. Purdue University	Research		Programmable cells embedded in switch lattice for programmable topology
Associative String Processor Brunel University	Research		Associative concepts; MIMD at high level; SIMD at low level
PICAP3 University of Paris	Prototype	8×8 array 64 cells	16-bit word length; image oriented

Table 2.2: **Contemporary Systolic Arrays in MIMD Configuration**

Machine	Cycle Time (ns)	Processors	Peak Mflops	Mflops per Proc
Amdahl 500	7.5	1	133	133
CRAY-1	12.5	1	160	160
CRAY X/MP-1	9.5	1	210	210
IBM 3090/VF-200	18.5	2	216	108
Amdahl 1100	7.5	1	267	267
NEC SX-1E	7	1	325	325
CDC CYBER 205	20	1	400	400
CRAY X/MP-2	9.5	2	420	210
IBM 3090/VF-400	18.5	4	432	108
Amdahl 1200	7.5	1	533	533
NEC SX-1	7	1	650	650
CRAY X-MP-4	9.5	4	840	210
Hitachi S-810/20	14	1	840	840
NEC SX-2	6	1	1300	1300
CRAY 2	4.1	4	2000	500

Table 2.3: **Comparison of Vector Computers in the 1980's**

Machine	Cycle Time (ns)	Processors	Peak Mflops	Mflops per Proc
HITAC S-3800	2	4	32000	8000
NEC SX-3R	4	4	25600	6400
Fujitsu VP2000	3.2	2	10000	5000
CRAY C90	4.2	16	16000	1000
IBM ES/9000	-	-	2670	-

Table 2.4: **Comparison of Vector Computers in the 1990's**

Machine	Architecture	Node Processor	Peak Node Mflops (D.P.)
Intel XP/S	MIMD 2d mesh	Intel i860	50
Kendell Square Research KSR1	MIMD rings	Custom RISC	40
MasPar MP-1	SIMD 2d mesh	32 4-bit custom	1.2
Meiko Sc. Corp	MIMD variable	i860 or SPARC	40 (i860)
nCube 2S	MIMD hypercube	64-bit custom	2.4
Parsytec GC	MIMD 3d mesh, variable	Transputer T9000	25
Thinking Machines CM-5	MIMD fat tree	SPARC	128 [a]
Wavetracer Data Transport	SIMD 3d mesh	Custom bit-serial	N/A

[a]with four vector units

Table 2.5: **Comparison of Massively Parallel Computers**

The matrix engine is aimed at a middle ground of competing technologies. As such, it should be faster than similar systems, such as those in Table 2.1 & 2.2. It should also compete on speed with the previous generation of vector machines (in Table 2.3), although the physical size and power consumption will be lower. It is these machines that many users are currently using, and making the matrix engine have similar performance will enable these users to get the same level of performance on the desktop, with lower initial and running costs. The latest vector machines, such as those in Table 2.4, can be faster than the matrix engine, but will be larger, more expensive, and consume more power. It is felt that a speed in excess of one GigaFLOP (1×10^9 floating point operations per second) is required on a matrix engine with array size of between 800 and 1600 elements to fulfill these conditions. This figure is the sustained or equivalent 'required' floating point performance, ie the total number of required floating point operations using a very efficient (the most efficient) algorithm divided by the total time, including start-up. Note that the performance figures quoted in Table 2.4 are the peak performances of the listed supercomputers, and that peak performance on these types of machine are typically three to ten times higher than the sustained or equivalent required performance [93].

The work undertaken for this thesis was to investigate the ability of algorithms to be modified to this simplified array, and to design the interface and memory specifications. This included an analysis of the algorithms to include any 'special' or 'sundry' functions that must be incorporated into the interface 'data controller' chips, and also the design specification of a memory subsystem that would efficiently support the algorithm. Additionally, the feasibility of incorporating several complete processing arrays, or 'matrix engines', into a multiprocessor architecture was investigated.

To this end, the remaining chapters have been divided largely according to these task requirements, although the actual design work of each part was largely done in parallel.

Chapter 3

Algorithms

In this chapter, four algorithms and their implementation on the matrix processing array will be presented. The algorithms are matrix multiplication, Gauss-Jordan elimination, the discrete Fourier transform and Kalman filtering, covering fields including numerical computation, signal processing and control systems. The expected performance of the algorithms when implemented on the matrix array with memory interface is presented in Chapter 7. A overview of the memory system, enough to explain the algorithm analysis, is presented here, while the memory subsystem is described in more detail in Chapter 6.

3.1 Memory Subsystem: An Overview

Most computations are of a form in which two operands are fetched from memory and a solution is stored to memory for each operation. Therefore, three ports to memory are ideally available if the matrix processing array (the computational engine) is to maintain its high computation rate. Unfortunately, several algorithms that will be implemented on the matrix engine have addressing patterns that cause conflicts when implemented on standard high performance three-port memories. Therefore, a bank-switched four-port memory was devised that allows two loads and one store to memory simultaneously. The bank-switching is achieved using a custom 4-1 bidirectional multiplexer, with caches attached to the memory banks that switch between the memory ports of the matrix engine. Additionally, to cope with data that must be recirculated from the output back to the inputs very quickly, two dual-ported static RAM chips are included, one for each of the output to input paths. A block diagram of the memory system is shown in Figure 3.1.

As shown in the figure, each memory bank has a cache system attached to it, on the memory side of the bank switches, unlike many other systems which attach the cache to the data port. This means that cache data will move with the main memory bank, easing the constraints caused by cache flushing and coherency considerations.

3.2 Matrix Multiplication

The most basic of operations to be run on the matrix engine is matrix multiplication. As mentioned in Chapter 2, the matrix array is configured to perform an outer product. Thus each of the data controllers applies a vector to the edge of the array, and the array produces the outer product of the two vectors. The outer product can thus be defined as

$$\mathbf{uv} = (v_1\mathbf{u}, v_2\mathbf{u}, \ldots, v_r\mathbf{u}) \tag{3.1}$$

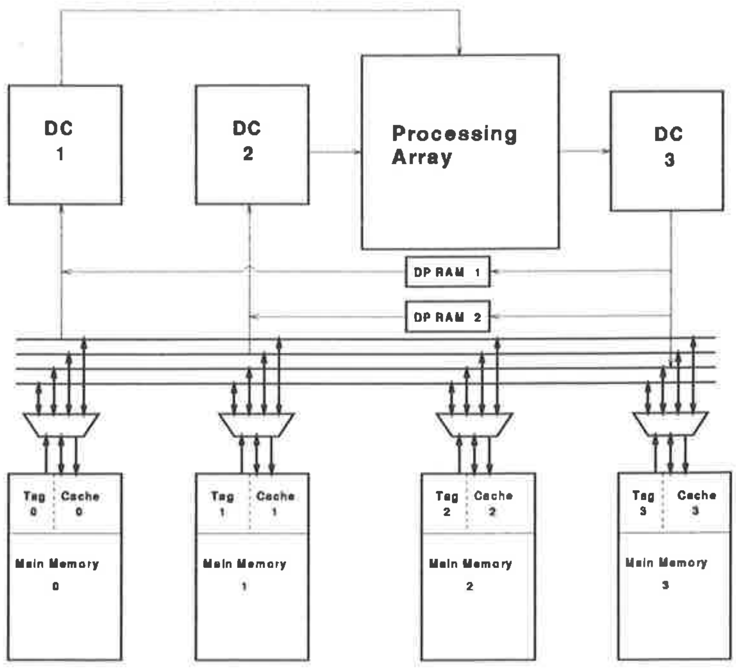


Figure 3.1: **Block Diagram of System**

where r is the dimension of the vectors $\mathbf{v}$.

The matrix product in terms of outer products is the in-place sum of the vector outer products [58, 70, 65], ie

$$\begin{aligned}
\mathbf{C} &= \mathbf{AB} \\
&= (\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_t) \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_n \end{pmatrix} \\
&= \sum_{i=1}^{t} \mathbf{a}_i \mathbf{b}_i \\
&= \sum_{i=1}^{t} (b_{i1}\mathbf{a}_i, b_{i2}\mathbf{a}_i, \ldots, b_{ir}\mathbf{a}_i) \qquad (3.2)
\end{aligned}$$

3.2.1 Partitioning

While Equation 3.2 is appropriate to describe the outer product of two matrices that are of a smaller size than the processing array of the matrix engine, in practice the matrix will generally be much larger than the processing array. Indeed, if the matrix were smaller, it may be faster to process such a small problem on a general processor, rather than going to the expense of configuring the matrix array. What is needed is a way of partitioning the matrix multiplication into segments that can be implemented on a fixed-size processing array that is smaller than the matrix multiplication problem.

The partitioning is relatively simple, and can be derived by dividing the vectors $\mathbf{a}_i$ in Equation 3.2 into several vectors with at most p non-zero elements, for a processing array size of

$p \times p$. The vectors become

$$\mathbf{a}_i = \begin{pmatrix} a_{1i} \\ a_{2i} \\ \vdots \\ a_{pi} \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ a_{(p+1)i} \\ \vdots \\ a_{(2p)i} \\ 0 \\ \vdots \end{pmatrix} + \cdots \tag{3.3}$$

and each is processed in turn. Of course, the zeros are only included for theoretical explanation, and in practice, only the non-zero elements are used. One way to consider the partitioning is to think of the partitioned outer product as an inner product of outer products, ie

$$\begin{aligned} \mathbf{C} &= \mathbf{AB} \\ &= \begin{pmatrix} \mathbf{A}_{1:}\mathbf{B}_{:1} & \mathbf{A}_{1:}\mathbf{B}_{:2} & \cdots & \mathbf{A}_{1:}\mathbf{B}_{:\lceil \frac{r}{p} \rceil} \\ \mathbf{A}_{2:}\mathbf{B}_{:1} & \mathbf{A}_{2:}\mathbf{B}_{:2} & \cdots & \mathbf{A}_{2:}\mathbf{B}_{:\lceil \frac{r}{p} \rceil} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{\lceil \frac{s}{p} \rceil:}\mathbf{B}_{:1} & \mathbf{A}_{\lceil \frac{s}{p} \rceil:}\mathbf{B}_{:2} & \cdots & \mathbf{A}_{\lceil \frac{s}{p} \rceil:}\mathbf{B}_{:\lceil \frac{r}{p} \rceil} \end{pmatrix} \end{aligned} \tag{3.4}$$

where the colon (:) is used to indicate a complete matrix dimension, in this case of a submatrix. Therefore, the notation $\mathbf{A}_{1:}$ indicates rows $1, \ldots, p$ and columns $1, \ldots, n$ of an $m \times n$ matrix $\mathbf{A}$, $\mathbf{A}_{2:}$ indicates rows $p+1, \ldots, 2p$ and columns $1, \ldots, n$, etc., and is referred to as the *row matrix* A_i. Similarly, the notation $\mathbf{B}_{:1}$ indicates rows $1, \ldots, m$ and columns $1, \ldots, p$ of an $m \times n$ matrix $\mathbf{B}$, $\mathbf{B}_{2:}$ indicates rows $1, \ldots, m$ and columns $p+1, \ldots, 2p$, etc., and is referred to as the *column matrix* B_i

Then each block of the matrix $\mathbf{C}$ can be calculated as:

$$\mathbf{C}_{ij} = \sum_t \mathbf{a}_i^t \mathbf{b}_j^t \tag{3.5}$$

where $\mathbf{a}_i^t$ is the t^{th} column of the row matrix $\mathbf{A}_i$, $\mathbf{b}_i^t$ is the t^{th} row of the column matrix $\mathbf{B}^j$, and $\mathbf{C}_{ij}$ is the $(i,j)^{th}$ partition of the matrix $\mathbf{C}$.

In effect, the partitioning can be viewed as a means of producing p^2 inner-products simultaneously.

Dividing the matrix in the general case without padding produces four regions, as defined in Figure 3.2.

The subscripts denote the coordinates of the block being calculated, while the superscript defines the region in which the block falls. This notation follows that defined in [65, 58].

A two dimensional address generator can extract each of the blocks one at a time. The overhead involved in accessing each partition independently is large, and an automated manner of determining each block in turn is necessary. A four-dimensional address generator will automatically calculate the partitions in region 1, a three- dimensional generator is needed for regions 2 and 3, while a two- dimensional generator is required for region 4. Together with the choice of four 'base-addresses', the partitions shown in Figure 3.2 can be automatically generated without any external intervention. A C++ class to implement this is found in [58].

A consideration to be made is one concerning the fetching of data from main memory into the cache. The memory architecture proposed in Chapter 6 allows two loads and one store

$$\left(\begin{array}{cccc|c}
\mathbf{C}^1_{11} & \mathbf{C}^1_{12} & \cdots & \mathbf{C}^1_{1\lfloor \frac{s}{p} \rfloor} & \mathbf{C}^2_{1\lceil \frac{s}{p} \rceil} \\
\mathbf{C}^1_{21} & \mathbf{C}^1_{22} & \cdots & \mathbf{C}^1_{2\lfloor \frac{s}{p} \rfloor} & \mathbf{C}^2_{2\lceil \frac{s}{p} \rceil} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\mathbf{C}^1_{\lfloor \frac{r}{p} \rfloor 1} & \mathbf{C}^1_{\lfloor \frac{r}{p} \rfloor 2} & \cdots & \mathbf{C}^1_{\lfloor \frac{r}{p} \rfloor \lfloor \frac{s}{p} \rfloor} & \mathbf{C}^2_{\lfloor \frac{r}{p} \rfloor \lceil \frac{s}{p} \rceil} \\
\hline
\mathbf{C}^3_{\lceil \frac{r}{p} \rceil 1} & \mathbf{C}^3_{\lceil \frac{r}{p} \rceil 2} & \cdots & \mathbf{C}^3_{\lceil \frac{r}{p} \rceil \lfloor \frac{s}{p} \rfloor} & \mathbf{C}^4_{\lceil \frac{r}{p} \rceil \lceil \frac{s}{p} \rceil}
\end{array}\right)$$

Figure 3.2: **Four Regions of a Partitioned Matrix Product**

$$\begin{pmatrix}
\mathbf{C}_{11} & \mathbf{C}_{12} & \cdots & \mathbf{C}_{1N} \\
\mathbf{C}_{21} & \mathbf{C}_{22} & \cdots & \mathbf{C}_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
\mathbf{C}_{N1} & \mathbf{C}_{N2} & \cdots & \mathbf{C}_{NN}
\end{pmatrix} = \begin{pmatrix}
\mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1N} \\
\mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
\mathbf{A}_{N1} & \mathbf{A}_{N2} & \cdots & \mathbf{A}_{NN}
\end{pmatrix} \begin{pmatrix}
\mathbf{B}_{11} & \mathbf{B}_{12} & \cdots & \mathbf{B}_{1N} \\
\mathbf{B}_{21} & \mathbf{B}_{22} & \cdots & \mathbf{B}_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
\mathbf{B}_{N1} & \mathbf{B}_{N2} & \cdots & \mathbf{B}_{NN}
\end{pmatrix}$$

Figure 3.3: **Memory Access for Matrix Product**

simultaneously to memory. The caches improve the average memory access speed significantly due to the large amount of cache reuse[1]. However, if one input controller is fetching data from main memory after a cache 'miss', the other must stall and wait if it has a cache 'hit'. Therefore, assuming that the complete matrices fit into the cache (up to 1000×1000 for a 1 MWord cache), there is a factor of approximately two introduced into the miss ratio and hence an increased total miss penalty.

Using the matrix product in Figure 3.3, in which two $n \times n$ matrices are multiplied together on a $p \times p$ array and $N = \lceil \frac{n}{p} \rceil$, the origin of the factor of two can be seen as follows[2].

If the first block of the solution to be determined is block $\mathbf{C}_{11}$, then blocks $(\mathbf{A}_{11}, \mathbf{A}_{12}, \ldots, \mathbf{A}_{1N})$ and $(\mathbf{B}_{11}, \mathbf{B}_{21}, \ldots, \mathbf{B}_{N1})$ need to be fetched from memory into the cache. The next logical block to calculate would be either block $\mathbf{C}_{12}$ or $\mathbf{C}_{21}$. However, whichever of these is chosen, either the controller accessing matrix $\mathbf{A}$ or the controller accessing matrix $\mathbf{B}$ respectively will have a cache hit while the other has a cache miss. This results in a memory access time mismatch and the data controller that had the hit would have to stall. Therefore, the standard (logical) access pattern of calculating the solution blocks in rows or columns will result in cache misses along two of the four edges, ie misses when calculating blocks $\mathbf{C}_{11}, \mathbf{C}_{12}, \ldots, \mathbf{C}_{1N}$, and $\mathbf{C}_{21}, \ldots, \mathbf{C}_{N1}$. The remaining blocks will all achieve cache hits. This results in $2N - 1$ misses and $(N-1)^2 = N^2 - 2N + 1$ hits for a hit/miss ratio of $\frac{N^2-2N+1}{2N-1} = \frac{N}{2} - \frac{3}{4} + \frac{1}{8N-4}$.

If the pattern of calculating blocks is changed to calculating the diagonal blocks first, both data controllers will have cache hits and misses at the same times. As the misses all occur when the diagonal blocks are being fetched, there will be N misses and $N(N-1) = N^2 - N$ hits, for a hit/miss ratio of $\frac{N(N-1)}{N} = N - 1$. As $\frac{1}{8N-4} \to 0$ as N gets large, it is obvious that the hit/miss ratio for calculating the diagonals first is approximately twice that for calculating by rows or columns.

[1]For an $n \times n$ matrix on a $p \times p$ array, the data is used $\lceil \frac{n}{p} \rceil$ times

[2]This of course extends to a more general case of two arbitrary matrices being multiplied together. For convenience, the cache access pattern is considered, without loss of generality, for the given size

3.3 Solution of Sets of Linear Equations

With a speed in excess of one Gigaflop for matrix operations, where a FLOP is a Floating point Operation Per Second, a method of solving large systems of equations on the matrix engine would be of considerable benefit. Of the several possible algorithms available, it is Gauss-Jordan which seems to hold the most promise, although others (eg. Gauss elimination, LU decomposition) can easily be implemented with only relatively minor modifications.

The set of equations is denoted as

$$\mathbf{A}.\mathbf{x} = \mathbf{b} \tag{3.6}$$

where $\mathbf{A}$ is an $m \times n$ matrix, $\mathbf{x}$ is an $n \times 1$ vector of unknowns and $\mathbf{b}$ is an $n \times 1$ coefficient vector. There can, of course, by several 'right-hand-side vectors' $\mathbf{b}_0$, $\mathbf{b}_1$, ..., $\mathbf{b}_r$, in which case the vectors $\mathbf{x}$ and $\mathbf{b}$ become $n \times r$ matrices, and can be solved simultaneously.

3.4 Gauss-Jordan Elimination

In engineering and mathematics, the solution to a very large number of equations is often sought. Common methods of solving such a system of equations include Gauss-Jordan elimination, Gauss elimination and back substitution and LU-decomposition and backsubstitution[3]

Of the three Gaussian approaches to solution of equations (Gauss elimination and back substitution, Gauss-Jordan elimination and LU-decomposition and backsubstitution), Gauss-Jordan elimination is generally not the recommended method due to its increased operation count [48, 73] and the fact that all the 'right-hand-sides' of a set of equations[4] need to be stored and manipulated at the same time as the elimination. However, Gauss-Jordan elimination has advantages which are specific to implementation on a matrix array. These include:

- Fixed-column Size
  Gaussian and LU decomposition both 'triangularise' the matrix of linear equations, and then backsubstitute. One consequence of this is that the number of elements operated on in sequence is continually diminishing until only a single element exists in a row.

- Determination of the Inverse

3.4.1 Block Version

It is now well documented how to obtain a block version of Gaussian based elimination [1, 58], although a brief overview follows.

Gaussian elimination works due to three premises of elementary row operations that leave the matrix system unaffected as a whole. These operations are[48]:

- Interchange of two rows

- Multiplication of a row by a non-zero constant

- Addition of a constant multiple of one row to another

Thus, a system of linear equations S_1 is *row equivalent* to another system S_2 if S_1 can be transformed into S_2 using the elementary row operations listed above.

[3]Others exist, such as QR decomposition, which use more complicated operations than the linear addition and scaling operations in Gaussian elimination

[4]There can be many

The row-oriented Gauss-Jordan elimination is

```
for $k = 1$ to $m$
    $\hat{a}_{kk} = a_{kk}^{-1}$
    for $i = k+1$ to $n$
        $\hat{a}_{ki} = a_{ki}\hat{a}_{kk}$
    for $j = 1$ to $m$, $j \neq k$
        for $i = k+1$ to $n$
            $\hat{a}_{ji} = a_{ji} - a_{jk}\hat{a}_{ki}$
```

Figure 3.4: **Row Oriented Gauss-Jordan Elimination**

To transform this into a block form, simply let the scalars a_{ij} become block matrices $\mathbf{A}_{ij}$, where each $\mathbf{A}_{ij}$ is of dimensions $p \times p$ and p is the size of the matrix array. A premultiplication of p rows by a $p \times p$ block can be shown to be simply p elementary row operations, as shown below for the first row of just such an operation.

$$\begin{aligned} \mathbf{A}_{1:} &= \text{row1}\,\{\mathbf{BC}\} \\ &= (b_{11}c_{11} + b_{12}c_{21} + \ldots + b_{1p}c_{p1}, b_{11}c_{12} + b_{12}c_{22} + \ldots + b_{1p}c_{p2}, \ldots, \\ &\quad b_{11}c_{1p} + b_{12}c_{2p} + \ldots + b_{1p}c_{pp}) \end{aligned} \tag{3.7}$$

ie row one of $\mathbf{A}$ is a linear combination of the rows of $\mathbf{C}$, with the scalar coefficients being elements from the matrix $\mathbf{B}$.

3.4.2 Determining the Inverse of the Pivot Block

Although we now have an algorithm to calculate the solution of a set of linear equations in block form using only matrix outer products, there is still the requirement of calculating the inverse of the pivot blocks. Logically, Gauss-Jordan elimination could again be used for this calculation. However, as the systolic array would need to be unloaded after each *vector* of the matrix is applied, this would be very inefficient, and an alternative was sought.

Such an alternative is to use an iterative algorithm. One that is well known goes under the various names of *Hotelling's Method* and *Schultz's Method* and others, although the algorithm basically reduces to a matrix version of Newton's method of root finding [73, 70]. Newton's method applied to finding the inverse of a scalar is considered in more depth in Section 4.2.1. Suffice to say that it produces the result

$$\mathbf{Y}_{2n+1} = 2\mathbf{Y}_n - \mathbf{Y}_n.\mathbf{A}.\mathbf{Y}_n \tag{3.8}$$

which is an iterative matrix equation for forming the matrix inverse $\mathbf{Y}$ of the square matrix $\mathbf{A}$. Note that the precision of convergence n more than doubles at each iteration. In fact, the convergence is *quadratic* if $\mathbf{Y}_0$ is sufficiently close to $\mathbf{A}^{-1}$. One choice of the initial estimate $\mathbf{Y}_0$ that satisfies the convergence condition is

$$\mathbf{Y}_0 = diag(\tfrac{1}{a_{11}}, \tfrac{1}{a_{22}}, \ldots, \tfrac{1}{a_{nn}}) \tag{3.9}$$

Appendix B contains an explanation as to why this choice converges quadratically to the correct solution of the inverse of $\mathbf{A}$.

3.4.3 Implementation on the Proposed Memory Architecture

The memory architecture described in Chapter 6 is a multiport, banked-switched cached system. The structure is repeated in Figure 3.1 for convenience.

Care must be taken when implementing an algorithm on the matrix engine to ensure that a minimum of stalls occur while waiting for data to cycle from output to input via the memory (either cache/main memory or the dual ported loop memories). The load/unload structure of the processing elements in the array, as defined by Marwood [58], is such that a matrix-matrix product can be initiated simultaneously with unloading the array. The square array is matched such that the time to load and apply a $p \times p$ matrix to each of the inputs and calculate the outer products is the same as the time to unload the previous product, assuming memory load/store times are equal.

The Gauss-Jordan elimination can be run in three parts, namely

- Invert pivot block
- Normalise pivot row
- Apply elementary row operations to zero all blocks on the same column as the pivot block.

Implementing Block Inversion

Inverting a pivot block is a cyclical operation, and takes advantage of the dual-port RAMs that feed data directly from the output back to either of the inputs. Additionally, use is made of the fact that the transpose of a matrix can be extracted directly from the array, as described in Chapter 6. Figures 3.5a) and 3.5b) show the extraction in the standard and transposed forms respectively.

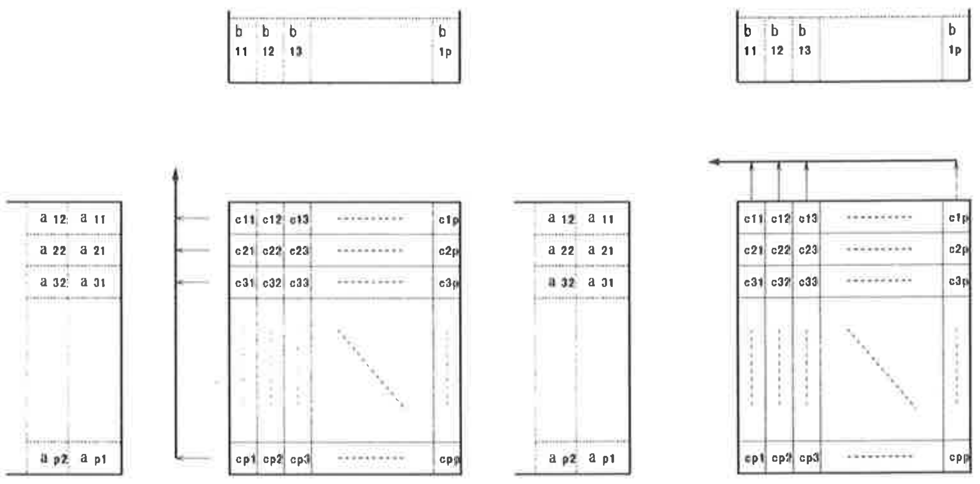


Figure 3.5: **Extracting a) Standard** **b) Transpose**

Initially, the matrix $\mathbf{A}_{ii}$ (the pivot block) is applied to the left-hand side (coefficients a_{jk} in Figure 3.5) and the matrix $\mathbf{Y}_0$ (the initial estimate of the inverse) applied from the top (coefficients y_{jk} in Figure 3.5). Recall that the initial estimate $\mathbf{Y}_0$ is generated 'on-the-fly' as the recipricals of the pivot elements of the input block $\mathbf{A}_{ii}$. If a negation is included in the calculation of $\mathbf{Y}_0$ and the transpose extracted from the array, the resulting product that is extracted is

$$Output(=\mathbf{X}_0) = -(\mathbf{A}.\mathbf{B}_0)^T \tag{3.10}$$

Noting the relationship

$$\mathbf{P}^T\mathbf{Q}^T = (\mathbf{QP})^T \tag{3.11}$$

and as $\mathbf{Y}_0$ is a diagonal matrix, then $\mathbf{Y}_0^T = \mathbf{Y}_0$, so that cycling the initial product $\mathbf{X}_0$ back to the left hand input and applying $\mathbf{Y}_0$ again from the top input produces the result

$$(-\mathbf{AY}_0)^T\mathbf{Y}_0 = (-\mathbf{Y}_0\mathbf{AY}_0)^T \tag{3.12}$$

If the matrices $2\mathbf{I}$ and $\mathbf{Y}_0$ are applied to the array while the previous product was being extracted, and the array is not cleared, the final product that results is the desired

$$\mathbf{Y}_1 = 2\mathbf{Y}_0 - \mathbf{Y}_0\mathbf{AY}_0 \tag{3.13}$$

Subsequent iterations proceed in a similar manner, except that the current estimate $\mathbf{Y}_n$ for the next estimate $\mathbf{Y}_{(2n+1)}$ may no longer be diagonal, so the relationship $\mathbf{Y}_0^T = \mathbf{Y}_0$ may not hold. However, as the estimate $\mathbf{Y}_n$ is now stored in the dual ported RAMS, and not calculated on the fly as it was for the first estimate, then the transposed matrix can be extracted simply from the RAMS and the same iteration procedure applies. The complete equations for later iterations become:

$$\begin{aligned} \mathbf{Y}_{(2n+1)} &= \left(2\mathbf{Y}_n^T + \left((-\mathbf{AY}_n)^T\mathbf{Y}_n^T\right)\right)^T && (3.14)\\ &= 2\mathbf{Y}_n - \mathbf{Y}_n\mathbf{AY}_n && (3.15)\end{aligned}$$

The sequence of 'n's will be

$$n = 0, 1, 3, 7, 15, 31, 63, 127, \ldots \tag{3.16}$$

Therefore, for 32-bit precision and setting $n = 0$ to start, five iterations are required. Once the final inverse has been determined, it is kept in the dual ported RAM for the normalisation phase.

Normalising the Pivot Row

This is a very simple part. Once the inverse of the pivot block is determined, each of the blocks in the pivot row are multiplied by the inverse of the pivot element. Thus, the pivot element becomes the identity matrix and does not need to be stored.

The one point that should be noted is the writing of data from the output. In traditional Gaussian elimination, storage is conserved by overwriting the original matrix with the updated version. However, as the memory architecture proposed does not allow simultaneous reads and writes to the same bank of memory, such a scheme would cause intolerable delays due to stalls. However, as memory is relatively cheap compared to improvements in processor performance, the updated matrix can be written to a new bank, which can then be swapped with the previous input bank for successive iterations.

Zero All Blocks in Pivot Column

This stage is the update phase, characterized by

$$\begin{aligned} \hat{\mathbf{A}}_{ji} &= \mathbf{A}_{ji} - \mathbf{A}_{jk}\mathbf{A}_{ki}\mathbf{A}_{kk}^{-1} \\ &= \mathbf{A}_{ji} - \mathbf{A}_{jk}\hat{\mathbf{A}}_{ki} \end{aligned} \tag{3.17}$$

Care needs to be taken to ensure that minimal memory conflicts occur during the update phase. Conflicts are likely in this algorithm and at this particular phase, as a block from the output must be fed into the input (namely the normalised block $\hat{\mathbf{A}}_{ki}$). Therefore, the output bank must be available to the input. Bearing in mind that the previously updated block will be in the process of being written to the output bank when the next update is started, then the normalised pivot row is not available. Therefore, to fill time, load the available block $\mathbf{A}_{ij}$ into the accumulators using the operation

$$\textit{Matrix Array} = \mathbf{A}_{ji}.I \tag{3.18}$$

By the time this operation is concluded, the previous write to the output bank should have concluded. Therefore, block $\hat{\mathbf{A}}_{ki}$ will be available, and can be applied to the top of the array. If this block is negated before it is applied, and the array is not cleared before applying the matrix product, the resulting matrix in the processing array will be

$$\textit{Matrix Array} = \mathbf{A}_{ji} - \mathbf{A}_{jk}\hat{\mathbf{A}}_{ki} \tag{3.19}$$

which is the desired update.

The ordering of the block update does have some significance, although only minimal losses occur if the wrong (non-optimal) ordering is used. Consider the row oriented form of Gauss Jordan elimination provided in Figure 3.4. If the pivot row is completely normalized before any zeroing is performed, then one of the inputs to the processing array must come from the memory bank that is being used to store the result, which will require a large amount of bank switching. This is generally not a problem, although it will result in a number of stalls due to cache tagging. If, however, the access is by *columns* of blocks, as shown in Figure 3.6, then the normalised block from the pivot row will be available in one of the feed-back dual-ported RAMs, and bank switching will not be required until the second full iteration of the algorithm. Each iteration will cause a minimum of stalls (only those relating to unmatched memory read/write times).

```
for $k = 1$ to $m$
    $\hat{a}_{kk} = a_{kk}^{-1}$
    for $i = k + 1$ to $n$
        $\hat{a}_{ki} = a_{ki}\hat{a}_{kk}$
        for $j = 1$ to $m$, $j \neq k$
            $\hat{a}_{ji} = a_{ji} - a_{jk}\hat{a}_{ki}$
```

Figure 3.6: **Column Oriented Gauss-Jordan Elimination**

3.4.4 Inverting Rather Than Solving a Set of Matrix Equations

The case may exist where it is desirable to invert the given matrix rather than solve it for a given set of 'right hand sides'. This is a simple extension of the Gauss-Jordan elimination technique, and can be implemented easily on the matrix engine.

Instead of performing elementary row operations on the matrix **A**, instead perform the same row operations on the augmented matrix $\tilde{\mathbf{A}} = [\mathbf{AI}]$. As Gauss-Jordan elimination on $\mathbf{AX} = \mathbf{I}$ implies that $\mathbf{X} = \mathbf{A}^{-1}\mathbf{I} = \mathbf{A}^{-1}$, then elimination on the new augmented matrix implies that, while **A** reduces to the identity **I**, the identity in the augmented matrix, $\mathbf{I}_a$, reduces to the desired inverse.

Of course, the entire augmented matrix need not be stored, as it can be created as needed. In the first iteration (on the first row of blocks), the only column of the identity matrix $\mathbf{I}_a$ that is affected is the first one. Therefore, as each column is eliminated from **A**, a new one is added onto the augmented matrix $\tilde{\mathbf{A}}$.

The actual operations that are required for the normalisation and update phases can be simplified when operating on the matrix $\mathbf{I}_a$ due to the inherent 'ones' and 'zeroes' in this matrix. When normalising the pivot row 'i', the operation for the 'new' column (column 'i' in $\mathbf{I}_a$) is

$$\begin{aligned} \hat{\mathbf{I}}_{a_{ii}} &= \mathbf{I}_{a_{ii}}.\mathbf{A}_{ii}^{-1} \\ &= \mathbf{A}_{ii}^{-1} \end{aligned}$$

which has already be calculated, so no extra work is required for the normalisation phase. The updating of all the other blocks in the same column is

$$\begin{aligned} \hat{\mathbf{I}}_{a_{ji}} &= \mathbf{I}_{a_{ji}} - \mathbf{A}_{ji}\mathbf{I}_{a_{ii}} \\ &= 0 - \mathbf{A}_{ji}\mathbf{A}_{ii}^{-1} \\ &= -\mathbf{A}_{ji}\mathbf{A}_{ii}^{-1} \end{aligned}$$

which is a single multiplication.

Thus the creation of the inverse can be achieved with simplicity by some intelligent extra control in the input and output data controllers that maintains rather than reducing the size of the applied matrix.

3.4.5 Solving Sets of Equations That Don't Fit Into the Cache

With 1 Megaword caches, Gauss-Jordan elimination can be used on matrices with up to approximately 1000×1000 elements without the need to load data more than once from main memory. However, systems of up to $10,000 \times 10,000$ or more can be solved efficiently, main memory size permitting[5]. It is important, however, to partition the problem into 'cache-sized chunks'.

The partitioning of the Gauss-Jordan algorithm to maximize cache reuse is similar to that used above to convert to block form (Section 3.4.1). Here, we will set the block size to accomodate a single block in a cache instead of in the processing array. Consider a cache size of 'C' Mwords. The maximum size (square) matrix that will fit in the cache is one of order $\sqrt{C}$ thousand elements. Therefore, divide the matrix into square blocks, each of size $\sqrt{C}$ $(= d)$ thousand element on a side. For an $M \times N$ matrix 'E', the division is as shown in Figure 3.7.

The pivot element E_{11} is initially inverted using Gauss-Jordan elimination as described previously. This will create a $d \times d$ matrix which can be multiplied by each of the other blocks in the pivot row, (ie $\forall E_{1:}$).

Once the pivot row has been normalised, the pivot column can be zeroed by subtracting the matrix product of the normalised pivot row and the matrix in the pivot column from each row

[5]Storing a matrix of size $10,000 \times 10,000$ in main memory requires 100 MWords = 400 Mbtyes of DRAM for 32-bit words. As the array requires a minimum of twice the matrix storage requirements to operate efficiently, at least 200 MWords of memory would be required. The DRAM densities of the near future make these figures feasible, especially when considering other similar performance systems. However, matrices of size 5000×5000 seems more reasonable. Note, however, that the main limitation is the *storage*, not the computation rate

$$\begin{pmatrix} E_{11} & E_{12} & \cdots & E_{1\lceil \frac{N}{d} \rceil} \\ E_{21} & E_{22} & \cdots & E_{2\lceil \frac{N}{d} \rceil} \\ \vdots & \vdots & \ddots & \vdots \\ E_{\lceil \frac{M}{d} \rceil 1} & E_{\lceil \frac{M}{d} \rceil 2} & \cdots & E_{\lceil \frac{M}{d} \rceil \lceil \frac{N}{d} \rceil} \end{pmatrix}$$

Figure 3.7: **Dividing Sets of Equations Into Blocks**

$E_{i:}$ in turn. These are all large matrix multiplication routines, and the matrix array will run at close to its peak speed.

3.4.6 Iteratively Improving a Set of Solutions

If a solution vector $\hat{\mathbf{x}}$ is obtained for a set of linear equations, it is possible that it could be different from the 'correct' solution $\mathbf{x}$ by the small amount $\delta\mathbf{x}$, due to numerical error accumulation. What has in fact been solved is the system of linear equations solved not for the vector of known coefficients $\mathbf{b}$, but rather for a slightly different set of coefficients $\hat{\mathbf{b}} = \mathbf{b} + \delta\mathbf{b}$, as shown in Equation 3.20.

$$\mathbf{A}.\hat{\mathbf{x}} = \hat{\mathbf{b}} \tag{3.20}$$

If the set of known coefficients from Equation 3.20 is subtracted, we get Equation 3.21

$$\begin{aligned} \hat{\mathbf{b}} - \mathbf{b} &= \mathbf{b} + \delta\mathbf{b} - \mathbf{b} \\ &= \delta\mathbf{b} \end{aligned} \tag{3.21}$$

and also Equation 3.22.

$$\begin{aligned} \hat{\mathbf{b}} - \mathbf{b} &= \mathbf{A}(\mathbf{x} + \delta\mathbf{x}) - \mathbf{A}\mathbf{x} \\ &= \mathbf{A}\delta\mathbf{x} \end{aligned} \tag{3.22}$$

Substituting leads to Equation 3.23, for which the entire right hand side is known, as $\mathbf{x} + \delta\mathbf{x}$ is solution that was calculated but incorrect and needs improving, and $\mathbf{b}$ is given.

$$\mathbf{A}\delta\mathbf{x} = \mathbf{A}(\mathbf{x} + \delta\mathbf{x}) - \mathbf{b} \tag{3.23}$$

Now $\mathbf{A}\delta\mathbf{x}$ can be solved for $\delta\mathbf{x}$, which can then be subtracted from $\mathbf{x} + \delta\mathbf{x}$ to return the corrected solution $\mathbf{x}$. The numerical error in $\delta\mathbf{x}$ will be approximately the same as for $\mathbf{x}$ in terms of the number of significant digits that can reliably be considered to be 'correct'. However, as the most significant bits of $\delta\mathbf{x}$ (the more 'correct' ones) are a very good approximation to the amount that $\mathbf{x} + \delta\mathbf{x}$ is different from $\mathbf{x}$, due to the difference in the orders of magnitudes, the overall numerical error of the corrected solution will be reduced.

Two important points need to be considered when implementing iterative improvement on the matrix processing array. Firstly, if only a single solution needs to be improved, ie $\hat{\mathbf{x}}$ is a vector rather than a matrix, matrix-vector operations will be required, which are inefficient on the matrix processing array. Therefore, where possible, the improvement should be used on *sets* of solutions, such that $\hat{\mathbf{x}}$ is a matrix, preferably of the same dimension as the processing array.

Secondly, as is pointed out by Press *et. al.*[73], if the solution vector is found using LU decomposition instead of Gauss-Jordan elimination, the LU-decomposed matrix can be reused for the solution to the left hand side of Equation 3.23, saving $O(n^3)$ operations. The matrix engine can be programmed to perform LU decomposition by simply saving the intermediate calculations rather than discarding them.

3.5 The Discrete Fourier Transform

The original aim of the matrix processor was to use it to compute fast digital signal processing algorithms. One of these is the Discrete Fourier Transform (DFT), which is of use when analysing the frequency spectrum of a given time-dependent signal. A signal defined in the *time domain* by the function *h(t)* has an equivalent function in the *frequency domain*, denoted *H(f)*. The Fourier Transform is simply the method of converting from one domain to another. The *Fourier transform* equations are :

$$H(f) = \int_{-\infty}^{\infty} h(t)e^{-2\pi\imath ft}dt \tag{3.24}$$

$$h(t) = \int_{-\infty}^{\infty} H(f)e^{2\pi\imath ft}df \tag{3.25}$$

Equations 3.24 and 3.25 are often written using angular frequency ω instead of f. However, using the non-angular frequency f, there will be fewer factors of 2π to consider.

For 'N' consecutively sampled data points sampled at intervals of 'Δ' time units, the sampled values 'h_k' are such that:

$$h_k = h(k\Delta) \qquad k = 0, 1, 2, \ldots, N-1 \tag{3.26}$$

This leads to the estimate being at the N discrete frequency points, given by

$$f_n \equiv \frac{n}{\Delta N} \tag{3.27}$$

where Δ is the time between samples.

Then the Fourier transform integrals of Equations 3.24 and 3.25 can be discretized to the form:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-2\pi\imath kn/N}$$

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)e^{2\pi\imath kn/N}$$

where $H(f_n) \simeq \Delta X(k)$ Substituting for $W_N = e^{-\imath 2\pi/N}$, then these discretized equations become:

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn} \tag{3.28}$$

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)W_N^{kn} \tag{3.29}$$

which are the equations describing the Discrete Fourier Transform (DFT).

Marwood derives the following two dimensional mapping of a one dimension DFT (ie Equations 3.28 & 3.29) [63, 55, 58]. Using the mappings

$$n = \langle M_1 n_1 + M_2 n_2 \rangle_N \tag{3.30}$$

$$k = \langle L_1 k_1 + L_2 k_2 \rangle_N \tag{3.31}$$

that translate the one dimensional quantities n and k into (n_1, n_2) and (k_1, k_2) respectively, with the constraints $0 \leq n_1, k_1 \leq N_1 - 1$ & $0 \leq n_2, k_2 \leq N_2 - 1$, for some constants $L_1, L_2, M_1 \& M_2$.

Then substituting these mappings into Equation 3.28, the DFT can be expressed in terms of the (reduced) two dimensional variables.

$$X(L_1k_1 + L_2k_2) = \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} x(M_1n_1 + M_2n_2)W_N^{kn}) \tag{3.32}$$

where

$$W_N^{kn} = W_N^{M_2L_2n_2k_2} W_N^{M_1L_2n_1k_2} W_N^{M_1L_1n_1k_1} W_N^{M_2L_1n_2k_1} \tag{3.33}$$

If the function $Y(k_1, k_2)$ is defined to be the mapping of $X(L_1k_1 + L_2k_2)$ to two dimensions, then Equation 3.32 becomes

$$Y(k_1, k_2) = \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} y(n_1, n_2)W_N^{kn} \tag{3.34}$$

Defining the mapping constants M_1, M_2, L_1 and L_2 to be

$$\begin{aligned} M_1 &= \alpha N_2 \\ M_2 &= \beta N_1 \\ L_1 &= \delta N_2 \\ L_2 &= \gamma N_1 \end{aligned}$$

and for the case where N_1 and N_2 are relatively prime, then the mapping becomes unique and the second and fourth factors in Equation 3.33 both equal one. Equation 3.34 becomes

$$Y(k_1, k_2) = \sum_{n_1=0}^{N_1-1}\left[\sum_{n_2=0}^{N_2-1} y(n_1, n_2)W_{N_2}^{\beta\gamma N_1n_2k_2}\right] W_{N_1}^{\alpha\delta N_2n_1k_1} \tag{3.35}$$

Adding the constraints suggested by Good [33]

$$\begin{aligned} \alpha &= 1 \\ \beta &= 1 \\ \delta &= \left\langle N_2^{-1}\right\rangle_{N_1} \\ \gamma &= \left\langle N_1^{-1}\right\rangle_{N_2} \end{aligned} \tag{3.36}$$

or by Burrus [12]

$$\alpha = \beta = \delta = \gamma = 1 \tag{3.37}$$

then Equation 3.35 can be written in terms of matrix operations as

$$Y = W_1XW_2 \tag{3.38}$$

where W_1 and W_2 are conventional Fourier matrices of dimension $N_1 \times N_1$ and $N_2 \times N_2$ respectively.

An important point to realize is that the two dimensional mapping can be extended to an arbitrary dimensioned mapping. In fact, Marwood and A.P.Clark show that it is necessary to use the higher dimensional mappings as the transform length increases if any performance advantage over conventional FFT algorithms is to be preserved [58, 60, 61]. From [58], the final result for a four dimensional case is

$$Y(k_1,k_2,k_3,k_4) = \sum_{n_1=0}^{N_1-1}\left[\sum_{n_2=0}^{N_2-1}\left[\sum_{n_3=0}^{N_3-1}\left[\sum_{n_4=0}^{N_4-1} y(n_1,n_2,n_3,n_4)\times W_{N_4^2}^{Nn_4k_4}\right]W_{N_3^2}^{Nn_3k_3}\right]W_{N_2^2}^{Nn_2k_2}\right]W_{N_1^2}^{Nn_1k_1} \tag{3.39}$$

and the coefficient matrix is again the matrix of Fourier coefficients such that

$$(W_{N_i})_{p,q} = e^{\frac{-\imath 2\pi Npq}{N_i^2}} \tag{3.40}$$

where

$$N = \prod_{i=1}^{4} N_i$$

From Equations 3.39 & 3.40, the extension to 'p' dimensions is obvious. It is shown in Equation 3.41 for completeness.

$$Y(k_1,k_2,\ldots,k_p) = \sum_{n_1=0}^{N_1-1}\left[\sum_{n_2=0}^{N_2-1}\left[\cdots\left[\sum_{n_p=0}^{N_p-1} y(n_1,n_2,\ldots,n_p)\times W_{N_p^2}^{Nn_pk_p}\right]\cdots\right]W_{N_2^2}^{Nn_2k_2}\right]W_{N_1^2}^{Nn_1k_1} \tag{3.41}$$

3.5.1 Implementation

Marwood [58] shows the procedure of implementing a four dimensional prime factored DFT. The procedure is described in terms of the operations

1. $X^I_{i,j} = W_{N_3}X_{i,j}$ $\quad 0 \le i,j \le N_1, N_2$
   where $X_{i,j}(p,q)$ is derived from $x(i,j,p,q) = i\times M_1 + j\times M_2 + p\times M_3 + q\times M_4$.
2. $X^{II}_{i,j} = W_{N_4}X^I_{i,j}$ $\quad 0 \le i,j \le N_2, N_3$
   where $X^I_{i,j}(p,q)$ is derived from $x(i,j,p,q) = i\times M_2 + j\times M_3 + p\times M_4 + q\times M_1$.
3. $X^{III}_{i,j} = W_{N_1}X^{II}_{i,j}$ $\quad 0 \le i,j \le N_3, N_4$
   where $X^{II}_{i,j}(p,q)$ is derived from $x(i,j,p,q) = i\times M_3 + j\times M_4 + p\times M_1 + q\times M_2$.
4. $X^{IV}_{i,j} = W_{N_2}X^{III}_{i,j}$ $\quad 0 \le i,j \le N_4, N_1$
   where $X^{III}_{i,j}(p,q)$ is derived from $x(i,j,p,q) = i\times M_4 + j\times M_1 + p\times M_2 + q\times M_3$.

A graphical representation of a multidimensional prime factor DFT can easily be shown in up to three dimensions ,as in Figure 3.8. This figure shows three coefficient matrices, $\mathbf{W_1}$, $\mathbf{W_2}$ & $\mathbf{W_3}$, each multiplied by the 'cube' of data '$\mathbf{X}$' in turn. Due to matrix associativity, the order in which the matrix products are calculated is not important.

From the diagram, it can be seen that the number of products and the sizes of such products are:

1. N_1 iterations of $(N_2\times N_2)\times(N_2\times N_3)$ & $(N_2\times N_3)\times(N_3\times N_3)$ products
2. N_2 iterations of $(N_3\times N_3)\times(N_3\times N_1)$ & $(N_3\times N_1)\times(N_1\times N_1)$ products
3. N_3 iterations of $(N_1\times N_1)\times(N_1\times N_2)$ & $(N_1\times N_2)\times(N_2\times N_2)$ products

This can be extended to one or more dimensions of two dimensional matrix products, which can each be executed in turn. Some performance estimates are given in Section 7.3 using this procedure. These show that the matrix engine performs very creditably on medium and large sized Fourier transforms problems.

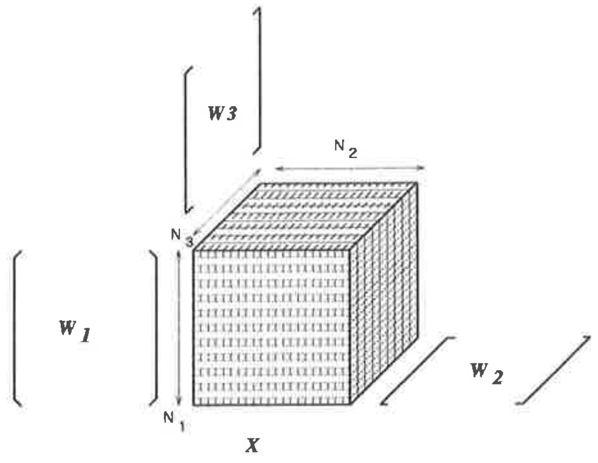


Figure 3.8: **3 Dimensional Representation of Prime Factor DFT**

3.6 The Kalman Filter

Often found in control systems, signal processing and communications, the Kalman filter is a very efficient method of providing minimum variance estimates from noisy measurements [45, 64, 31]. The Kalman filter is, however, of computational order $O(n^3)$, which provides quite a bottleneck if the filter is required for real-time broad-band applications.

Approximately ten years ago, the Kalman filter was used to estimate state vectors of size ten to twenty elements. As computational power has increased and powerful microprocessors and embedded systems have become cheaper, the number of elements in a state vector have increasedto approximately 50 to 70, which fully utilises the available power. However, high end control systems have always demanded computational power one to two orders of magnitude larger than the norm, capable of calculating the Kalman filter of state vectors three to five times larger than is common at the time. For example, the Kalman filter implemented on the Voyager I & II space-craft used 67 state variables with 3500 data points. The computer system for the filter was designed in 1977[14]. It seems that, no matter how much computational power is available, an application will be found that consumes the full amount, and possibly asks for more [88]. The aim of the implementation described here is therefore not to provide a competitor to existing systems, but to open the scope of the Kalman filter to include systems that were not implementable in previous systems.

There are several forms of the Kalman filter. The form presented here is that used by Gaston & Irwin [31]. They describe the Kalman filter in terms of the equations:

$$\mathbf{x}(k+1) = \mathbf{A}(k)\mathbf{x}(k) + \mathbf{B}(k)\mathbf{u}(k) + \mathbf{w}(k) \tag{3.42}$$
$$\mathbf{y}(k) = \mathbf{C}(k)\mathbf{x}(k) + \mathbf{v}(k) \tag{3.43}$$

where $\mathbf{x}(k)$ is the state vector of the system of dimension $(n \times 1)$, $\mathbf{y}(k)$ is the $(m \times 1)$ vector of measured output variables such that $m \geq n$, and $\mathbf{u}(k)$ is the $(p \times 1)$ control vector. The $(p \times 1)$ $\mathbf{v}(k)$ and $(m \times 1)$ $\mathbf{w}(k)$ vectors represent white noise with assumed zero mean and covariance. The matrices $\mathbf{A}(k)$, $\mathbf{B}(k)$ and $\mathbf{C}(k)$ are assumed to be known, and are of dimensions $(n \times n)$, $(n \times m)$ & $(p \times n)$ respectively.

The aim of the Kalman filter is to estimate the state of a system given a series of measurements, and as such, a prediction vector $\hat{\mathbf{x}}(k)$ is created that differs from the actual vector $\mathbf{x}(k)$

such that:

$$\mathbf{P}(k+1|k) = \mathbf{E}\left[(\mathbf{x}(k+1) - \hat{\mathbf{x}}(k+1|k)) \times (\mathbf{x}(k+1) - \hat{\mathbf{x}}(k+1|k))^T\right] \quad (3.44)$$
$$\mathbf{P}(k|k) = \mathbf{E}\left[(\mathbf{x}(k) - \hat{\mathbf{x}}(k|k))(\mathbf{x}(k) - \hat{\mathbf{x}}(k|k))^T\right] \quad (3.45)$$

where $\mathbf{P}$ is the error covariance matrix and the bracket notation '$(s|t)$' implies the estimate of the state at time 's' given measurements up to time 't', with $s \geq t$.

Papadourakis & Taylor [31] derive the filter equations shown in Equations 3.46 & 3.47 for a rectangular systolic array configured for matrix multiplication.

$$\hat{\mathbf{x}}(k+1|k) = \mathbf{A}(k)\hat{\mathbf{x}}(k|k-1) + \mathbf{A}(k)\mathbf{K}(k)\left[\mathbf{y}(k) - \mathbf{C}(k)\hat{\mathbf{x}}(k|k-1)\right] \quad (3.46)$$
$$\mathbf{P}(k+1|k) = \mathbf{A}(k)\mathbf{P}(k|k-1)\mathbf{A}^T(k) - \mathbf{A}(k)\mathbf{K}(k)\mathbf{C}(k)\mathbf{P}(k|k-1)\mathbf{A}^T(k) + \mathbf{W}(k) \quad (3.47)$$

where

$$\mathbf{K}(k) = \mathbf{P}(k|k-1)\mathbf{C}^T(k) \times \left[\mathbf{C}(k)\mathbf{P}(k|k-1)\mathbf{C}^T(k) + \mathbf{V}(k)\right]^{-1} \quad (3.48)$$

The covariances $\mathbf{W}(k)$ and $\mathbf{V}(k)$ may be assumed to be zero, although they are left in for completeness[6].

To implement the Kalman filters of Equations 3.46, 3.47 & 3.48, care must be taken to arrange the order of multiplications correctly for maximum efficiency. As there are a relatively large number of matrix multiplications per iteration, and the number of elements in the state vector will result in a small number of partitions, the ordering of the calculations should be considered in two separate cases, those of:

1. The number of state variables and measured variables are both smaller than the dimension of the processing array.

2. Either the number of state variables or the number of measured variables or both are larger than the dimension of the processing array.

For the first case, the ordering of computation, together with the labelling of intermediate calculations, is shown in Table 3.1. The table shows the procedure for calculating the Kalman filter directly, by showing the required inputs, the resultant calculation and a temporary name for this calculation, and whether to unload the calculation at the completion or to leave it in the array for accumulation. This ordering, assuming a bank-swapping memory architecture, requires only two delay slots, one before and one after the calculation of the inverse of $\mathbf{C}(k)\mathbf{P}(k|k-1)\mathbf{A}^T(k) + \mathbf{W}(k)$. These are required as the following calculation relies on the result of the current calculation. While these delays are not strictly necessary, as a future calculation can be brought forward to fill the delay, they are retained here to maintain coherence of the procedure by implementing the procedure directly.

If the dimensions of the state or observation vectors are larger than that of the processing array, the computations can be partitioned, which will help 'hide' some of the unload operations while other partitions are calculated. Table 3.2 shows an example ordering of the partitioned form. The subscripts following a matrix indicate the partition of the matrix, ie $\mathbf{X}(k)_{ij}$ implies the i^{th} row partition and the j^{th} column partition. If a colon is included, either a complete row or a complete column partition is implied, ie $\mathbf{X}(k)_{i:}$ is all columns of the i^{th} row partition and $\mathbf{X}(k)_{:j}$ is all rows of the j^{th} column partition.

[6]Their inclusion will not significantly affect the overall computation time

Input 1	Input 2	Calculation	Result Name	Leave/ Unload
		Compute $\mathbf{K}(k)$		
$\mathbf{P}(k\|k-1)$	$\mathbf{C}^T(k)$	$\mathbf{P}(k\|k-1)\mathbf{C}^T(k)$	$\mathbf{L}$	Unload
$\mathbf{V}(k)$	$\mathbf{I}$	$\mathbf{V}(k)$	-	Leave
$\mathbf{C}(k)$	$\mathbf{L}$	$\mathbf{C}(k)\mathbf{P}(k\|k-1)\mathbf{C}^T(k)+\mathbf{V}(k)$	$\mathbf{U}$	Unload
		Invert $\mathbf{U}$	$\mathbf{U}^{-1}$	Unload
$\mathbf{L}$	$\mathbf{U}^{-1}$	$\mathbf{L}.\mathbf{U}^{-1}$	$\mathbf{K}(k)$	Unload
		Compute $\mathbf{P}(k+1\|k)$		
$\mathbf{P}(k\|k-1)$	$\mathbf{A}^T(k)$	$\mathbf{P}(k\|k-1)\mathbf{A}^T(k)$	$\mathbf{M}$	Unload
$\mathbf{A}(k)$	$\mathbf{K}(k)$	$\mathbf{A}(k)\mathbf{K}(k)$	$\mathbf{N}$	Unload
$\mathbf{C}(k)$	$\mathbf{M}$	$\mathbf{C}(k)\mathbf{P}(k\|k-1)\mathbf{A}^T(k)$	$\mathbf{Q}$	Unload
$\mathbf{A}(k)$	$\mathbf{M}$	$\mathbf{A}(k)\mathbf{P}(k\|k-1)\mathbf{A}^T(k)$	$\mathbf{R}$	Leave
$\mathbf{N}$	$\mathbf{Q}$	$\mathbf{A}(k)\mathbf{K}(k)\mathbf{C}(k)\mathbf{P}(k\|k-1)\mathbf{A}^T(k)$	$\mathbf{S}$	Leave
$\mathbf{W}(k)$	$\mathbf{I}$	$\mathbf{W}(k)$	$\mathbf{P}(k+1\|k)$	Unload
		Compute $\hat{\mathbf{x}}(k+1\|k)$		
$\mathbf{C}(k)$	$\hat{\mathbf{x}}(k\|k-1)$	$\mathbf{C}(k)\hat{\mathbf{x}}(k\|k-1)$	-	Leave
$\mathbf{y}(k)$	$\mathbf{I}$	$\mathbf{y}(k)+\mathbf{C}(k)\hat{\mathbf{x}}(k\|k-1)$	$\mathbf{Z}$	Unload
$\mathbf{A}(k)$	$\hat{\mathbf{x}}(k\|k-1)$	$\mathbf{A}(k)\hat{\mathbf{x}}(k\|k-1)$	-	Leave
$\mathbf{N}$	$\mathbf{Z}$	$\mathbf{A}(k)\mathbf{K}(k)\mathbf{Z}$	$\hat{\mathbf{x}}(k+1\|k)$	Unload

Table 3.1: **Ordering when Number of State Vectors is Small**

Calculation	Result Name	Leave/ Unload
Compute $\mathbf{K}(k)$		
$\forall(i,j)$ $\mathbf{P}(k\|k-1)_{i:}\mathbf{C}^T(k)_{:j}$	$\mathbf{L}_{ij}$	Unload
$\forall(i,j)$ $\mathbf{V}(k)_{ij}\mathbf{I}+\mathbf{C}(k)_{i:}\mathbf{L}_{:j}$	$\mathbf{U}$	Unload
Invert $\mathbf{U}$	$\mathbf{U}^{-1}$	Unload
$\forall(i,j)$ $\mathbf{L}\mathbf{U}^{-1}$	$\mathbf{K}(k)$	Unload
Compute $\mathbf{P}(k+1\|k)$		
$\forall(i,j)$ $\mathbf{P}(k\|k-1)_{i:}\mathbf{A}^T(k)_{:j}$	$\mathbf{M}_{ij}$	Unload
$\forall(i,j)$ $\mathbf{A}(k)_{i:}\mathbf{K}(k)_{:j}$	$\mathbf{N}_{ij}$	Unload
$\forall(i,j)$ $\mathbf{C}(k)_{i:}\mathbf{M}_{:j}$	$\mathbf{Q}_{ij}$	Unload
$\forall(i,j)$ $\mathbf{A}(k)_{i:}\mathbf{M}_{j:}-\mathbf{N}_{i:}\mathbf{Q}_{:j}+\mathbf{W}(k)_{ij}\mathbf{I}$	$\mathbf{P}(k+1\|k)_{ij}$	Unload
Compute $\hat{\mathbf{x}}(k+1\|k)$		
$\forall(i)$ $-\mathbf{C}(k)_{i:}\hat{\mathbf{x}}(k\|k-1)+\mathbf{y}(k)$	$\mathbf{z}$	Unload
$\forall(i)$ $\mathbf{A}(k)_{i:}\hat{\mathbf{x}}(k\|k-1)+\mathbf{N}_{i:}\mathbf{z}$	$\hat{\mathbf{x}}(k+1\|k)$	Unload

Table 3.2: **Ordering when Number of State Vectors is Large**

The main inefficiency of directly implementing the routine on the matrix engine is Equation 3.46. Using a constant bandwidth array, all but one of the columns of the array would be fed with zeros, wasting a great deal of the potential performance.

Example performance estimates for simple systems are derived in Chapter 7. These suggest that the systolic matrix processing array will be very useful for calculations in large Kalman filter systems, and even possibly extend the range of applications for which Kalman filters are used to include hitherto unconsidered problems.

3.7 Conclusion

From the analysis of these few algorithms, it can be seen that a relatively simple processing array can perform more complex algorithms by using clever interface hardware. This combination will increase the versatility of the array while decreasing the cost compared to other systems of similar performance. The range of applications that the array can be used for extends into fields such as aerodynamics and fluid dynamics, embedded high speed signal processing and control, even neural networks, to name but a few.

Chapter 4

Arithmetic

As will be shown in Chapter 5, the address generator in the data controller is the main bottleneck of system performance. The address generator is implemented using an Extended Marwood Difference Engine, which provides modulo arithmetic using only the addition and subtraction operators [65]. Therefore, any improvement in the speed of addition and subtraction will result in a faster difference engine/address generator, and ultimately better system performance.

To maximise performance of this critical path without resorting to exotic technologies, fundamental arithmetic was reviewed, and some solutions proposed. In this chapter, we will explore the possibility of using non-conventional number systems to reduce the computational delay of an addition/subtraction operation. Specifically, a signed-digit arithmetic system, in which digits may take the values of not only zero and one but also negative one, is presented for use in the address generator.

In the second part of the chapter, division and multiplication is examined. The previous chapter on algorithms indicated the need for a divider in the data-input path to the array for the efficient implementation of Gauss-Jordan elimination. The data controller, which supply data to the processing array, is the logical position for the divider, as the input data must pass through the data controllers on its way to the processing array. An iterative divider that provides a good speed/area trade-off while still performing fast enough to not significantly affect the overall performance of the system is also presented, together with a signed-digit multiplier that is used in the divider.

4.1 Signed-Digit, or Carry-Propagation Free, Arithmetic

Although the concepts of signed digit arithmetic are not new [3, 4, 91], it is only more recently that they have become a viable alternative, due to the availability of VLSI circuits in which speed and not size is the major concern.

In a 'conventional' number system, such as two's complement binary or IEEE 754 floating point, each representable number has a unique representation, thus maximising the range of numbers that can be represented in a given number of bits. The redundant binary representation allows more than one representation of most numbers in the number set's range by using the 'digit set' of {-1,0,1} (-1 often represented by $\overline{1}$). Therefore, the representable range is not optimal for the number of bits required, although, as we shall see, there are other benefits.

Figure 4.1 shows an idealised version of a single digit cell in a signed-digit adder. The figure is idealised because it has been broken into three distinct parts, or stages, to demonstrate the carry-propagation path, whereas in practice, all the parts are merged to reduce hardware requirements and cell delay.

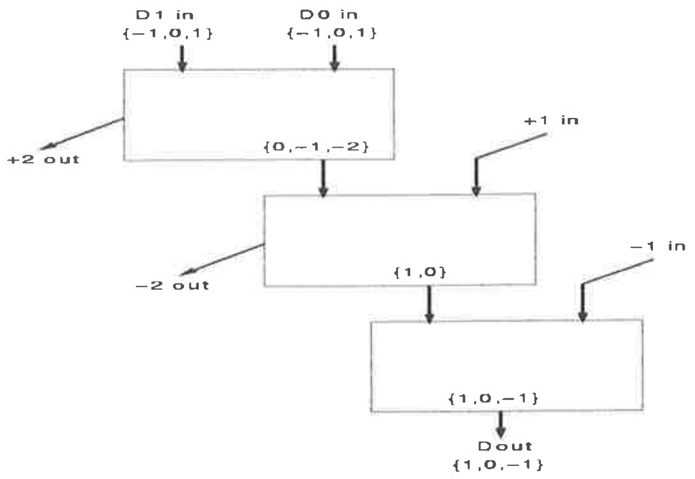


Figure 4.1: **Sign-Digit Addition Cell.**

Consider two numbers, X and Y (say), applied to the two inputs of the i^{th} digit cell. As X and Y are both signed-digits (ie each in the range {-1,0,1}), the sum of the two is in the range {-2,-1,0,1,2}. It is obvious that if the answer is '2', then the '+2 out' signal is flagged true. But what about if the answer is '1'? Due to the availability of negative numbers, '1' could be represented as *either* '+2 out' plus '-1' *or* '1'. This is due to the *redundancy* in the system.

The idea, then, is to minimise the set of possible outputs from each part of the digit cell. Therefore, in the case mentioned above of $X + Y = 1$, the choice would be to flag '+2 out' , as this reduces the set of possible outputs to the next stage to {-2,-1,0}.

Similarly for the second stage, the input from the first stage in the range {-2,-1,0} and the '+2 out' from the previous digit translated to the range {0,1} due to the increase in significance, results in the output being in the range {-2,-1,0,1}. If the '-2 out' flag is set for the cases of -2 and -1, the output from the second stage will be in the range {0,1}.

The third and final stage must output a number in the range {-1,0,1}, which is obviously the case. The input to the final stage from the second stage is in the range {0,1}, and the input from the '-2 out' of the previous digit is either '-1' or '0'. Therefore, the output from the final stage is in the range {-1,0,1} - another signed-digit number.

4.1.1 Why SD Arithmetic

The obvious question now is - why go to all the trouble of conversion to and from a redundant system and also pay the penalty of the extra area requirements? As hinted above, the answer to this is that a redundant system is 'carry-propagation free'. While each cell is not entirely independent of other cells, the dependency can be traced back exactly two cells.

To see this, it's easiest to work back from the output stage. Figure 4.2 shows the propagation paths for the 'i^{th}' cell in an adder.

The final output is dependent on the output from the second stage (i_m) and the '-1 in' signal from the previous cell ($(i - 1)_m$)[1]. The '-1 in' signal from stage $(i - 1)_m$ is dependent on the output from the first stage of cell $(i - 1)$ (ie stage $(i - 1)_f$) and the '+1 in' signal from stage $(i - 2)_f$. As the '+1 in' for stage $(i - 1)_m$ is actually '+2 out' from stage $(i - 2)_f$, which is

[1]Note that '±2 out' from one cell becomes '±1 in' to the next cell, due to the increased significance of the higher order cell.

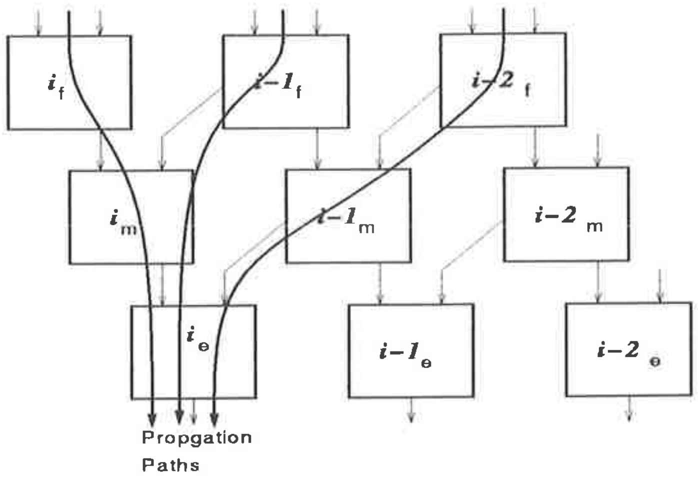


Figure 4.2: **Sign-Digit Propagation Path.**

dependent only on the two (signed-digit) inputs, the longest carry propagation path is from the two inputs of stage $(i-2)_f$ to the output of stage i_e - ie. two cells in length.

Now the two main advantages of Signed Digit Arithmetic become obvious. These are:

- faster calculation due to the reduced carry-propagation path. This path is now 2 cells long rather than 'n' cells for a carry-propagate adder. Although alternative schemes such as Carry Look-Ahead and Carry-Select have reduced computation times (log(n) and $\sqrt{n}$ respectively), they both have a greatly increased area over a standard ripple carry, and a propagation time still dependent on the size of the data.
- as the calculation time is fixed at the propagation time through two cells (independent of operand size), it is easier to match stages in a pipelined system with varying adder lengths. This is the case for a multiplier, in which the second stage adder will be longer than the first, the third stage longer than the second, etc.

The third advantage is a bit more esoteric. As IEEE floating point expresses the mantissa in as an unsigned binary number with a sign bit, conversion to signed-digit form is simply a matter of making the magnitude bits of all digits equal to the magnitude of the corresponding mantissa bit and the sign bit for each digit equal to the sign bit of the floating point number. Thus no addition is required as is the case when IEEE floating point is converted to two's compliment.

4.1.2 Sign Magnitude Specifics

As there are three distinct values in the number set {-1,0,1}, two bits will be required to encode the three possibilities for each digit. Generally, there can be held to be two signed-digit representations of this number set. These are:

- Signed-Bit Form

  The name Signed-Digit is often used here instead of Signed-Bit, but can often be confused with the *concept* Signed-Digit rather than the *implementation* Signed-Digit. In this notation, each bit represents a number of magnitude one or zero, one representing negative one and the other representing positive one. Thus the name refers to the fact that each bit represents either a positive or a negative unit. Then the sum of the combination of the two bits provides a digit in the range required.

The combination of the two bits, labelled n and p for negative one and positive one respectively, can be summarised in a table as shown below (Table 4.1).

BITS		*Value*
n	p	
0	0	0
0	1	+1
1	0	-1
1	1	0

Table 4.1: **Table of SD representation**

- Sign-Magnitude Form

  Similar in concept to the familiar sign magnitude form of the IEEE floating point standard, this approach provides a sign bit for *every* bit in a number. The two bits representing each digit can therefore be individually referred to as the sign bit, s, and the magnitude bit m. The number range {-1,0,1} is then represented as shown in Table 4.2.

BITS		*Value*
s	m	
0	0	+0
0	1	+1
1	0	-0
1	1	-1

Table 4.2: **Table of SM representation**

Note that the sign-magnitude notation implicitly includes a sign for every number, including zero. Therefore, there are distinct positive and negative zeros, unlike the signed-digit form.

4.1.3 Conversion To and From Signed Digit and Two's Complement

The main disadvantage of any new number format is that converting existing formats to the new one, and back again, can be complex. The latency of such a conversion done 'on-the-fly' is often too large to make a new format practical[2]. However, one of the main advantages of the redundant arithmetic described above is that conventional binary notation is a subset of the redundant representation, making conversion fast and simple. Note that the description here is limited to two's complement conversion, although floating point conversion is even simpler[3].

[2]This is generally the case for formats like residue arithmetic [92]

[3]As the IEEE floating point standard is in sign magnitude form [32], all that is needed to convert to signed-digit form is to assign the sign bit of the IEEE format number to all the digits in the signed-digit representation. Conversion the other way requires that the sign of the signed-digit number is determined, and assigned to the unsigned version of the signed-digit number

Converting From Two's Complement Notation

The conversion to a signed digit number from a positive, two's complement number is simply a matter of assigning each of the magnitude bits in the signed digit representation the value of the corresponding bit in the two's complement representation. The sign bits of all digits are set to zero, as shown in Figure 4.3.

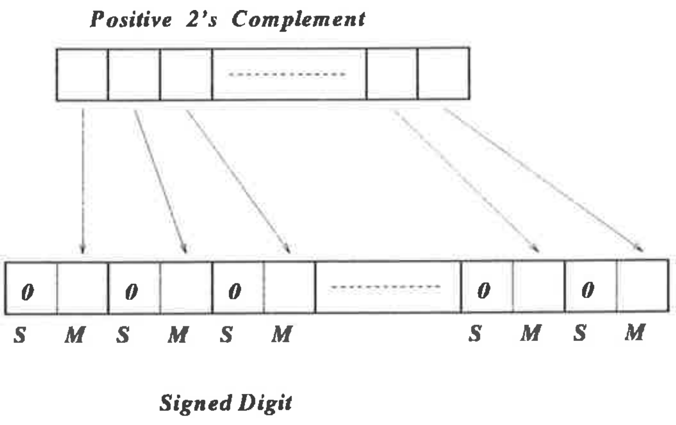


Figure 4.3: **Converting Positive 2's Complement to Signed Digit.**

Conversion of a negative two's complement number to signed digit involves converting to a positive number and setting all the sign bits in each digit to one. To convert a negative two's complement number to a positive one, all the bits are inverted, and one is added. The addition of one is often done 'on-the-fly' by applying a one to the 'carry-in' of the least significant adder. However, as the signed digit number is actually a negative number, negative one must be applied to the carry-in of the least significant adder, via the '*Neg_In*' input. This is shown in Figure 4.4.

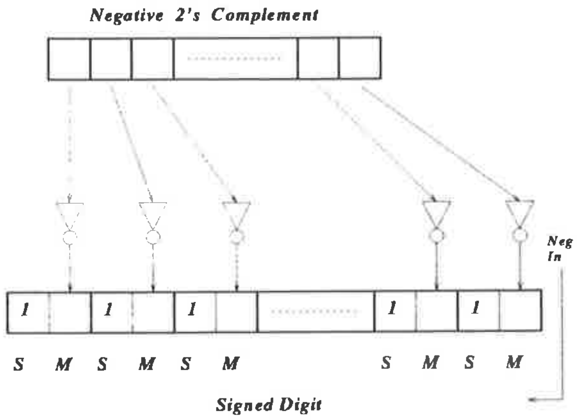


Figure 4.4: **Converting Negative 2's Complement to Signed Digit.**

Therefore, all that is required for the conversion from two's complement to signed digit notation is a series of inverters to complement a negative two's complement number, and several multiplexers controlled by the most significant bit of the two's complement number number to switch the between the positive and negative numbers.

Converting To Two's Complement Notation

This is a little more tricky than the other way round (Section 4.1.3), as each digit in a signed digit number can be either negative or positive. One possible approach is to start at the most significant bit of the signed digit number, and add or subtract each successive lower significant bit in turn. Another is to convert the signed digit number into a 'positive' number and a 'negative' number, and then perform a propagate addition. The latter arrangement appears to hold more promise, as configurations such as Carry-Select and CSA adders can be used to speed up the propagate addition, whereas the former is an inherently serial operation.

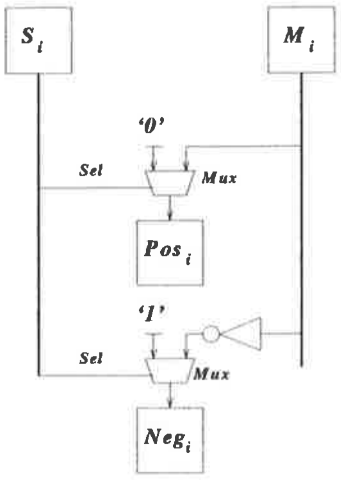


Figure 4.5: **Converting Signed Digit to 2's Complement.**

4.1.4 Signed Digit Implementation

In the practical implementation, there are many simplifications that are available by combining all three stages of the adder. The only requirement, then, is to recall that the '*pos_out*' signal depends only on the two signed magnitude inputs, the '*neg_out*' signal depends only on the two signed digit inputs and the '*pos_in*' input (*pos_out* from the previous stage), and the signed digit output is dependent on all the inputs. One possible implementation is given in the tables below.

Pos_Out		S_0M_0			
		00	01	11	10
S_1M_1	00	0	①	⓪	0
	01	①	①	1	①
	11	⓪	0	⓪	⓪
	10	0	①	⓪	0

Neg_Out		S_0M_0				S_0M_0					
		00	01	11	10	10	11	01	00		
S_1M_1	00	0	1	①	0	0	0	⓪	0	00	S_1M_1
	01	1	⓪	1	1	⓪	1	⓪	⓪	01	
	11	①	0	①	①	0	1	0	0	11	
	10	0	1	①	0	0	0	⓪	0	11	
Pos_In		0				1					

S_Out				*Pos_In*									
				0				1					
				S_0M_0				S_0M_0					
				00	01	11	10	10	11	01	00		
Neg In	0	S_1M_1	00	X	0	0	X	0	X	X	0	00	S_1M_1
			01	0	X	X	0	X	0	0	X	01	
			11	0	X	X	0	X	0	0	X	11	
			10	X	0	0	X	0	X	X	0	10	
	1	S_1M_1	10	1	X	X	1	X	1	1	X	10	S_1M_1
			11	X	1	1	X	1	X	X	1	11	
			01	X	1	1	X	1	X	X	1	01	
			00	1	X	X	1	X	1	1	X	00	

M_Out [a]				*Pos_In*									
				0				1					
				S_0M_0				S_0M_0					
				00	01	11	10	10	11	01	00		
Neg In	0	S_1M_1	00	0	1	1	0	1	0	0	1	00	S_1M_1
			01	1	0	0	1	0	1	1	0	01	
			11	1	0	0	1	0	1	1	0	11	
			10	0	1	1	0	1	0	0	1	10	
	1	S_1M_1	10	1	0	0	1	0	1	1	0	10	S_1M_1
			11	0	1	1	0	1	0	0	1	11	
			01	0	1	1	0	1	0	0	1	01	
			00	1	0	0	1	0	1	1	0	00	

[a] All the elements are essential for this Karnaugh Map; the surrounding circles are removed for clarity

Due to the redundancy available, there are several simultaneous degrees of freedom open to the designer. In the Karnaugh maps above, 'essential' combinations are circled, except for the case of M_{out}, for which all combinations are essential. An 'essential' combination is one that is essential for maintaining the requirements of Signed Digit (propagation free) arithmetic. For example, if either (S_1,M_1) or (S_0,M_0) or both equals positive one and neither equals negative one, then the pos_{out} signal must be asserted. Additionally, once pos_{out} has been asserted, the neg_{out} can be set to represent the number '+1' as either $+2-2+1$ (pos_{out} + neg_{out} + 1) or $+2-1$ (pos_{out} -1). Many other such possibilities exist. The groupings in Equations 4.1 to 4.7 were chosen as they appeared to minimise the logic equations. However, due to so many options being available, a completely automated process is not possible. The program 'ESPRESSO'[4] was used to position the essential combinations, and the groupings made by hand.

$$\begin{aligned}
pos_{out} &= \overline{S_1}M_1 + \overline{M_1 S_0}M_0 \\
&= \overline{\overline{(\overline{S_1}.M_1)}.\overline{(\overline{M_1}.\overline{S_0}.M_0)}} && (4.1) \\
\overline{pos_{out}} &= S_1.M_1 + \overline{M_1}.S_0 + \overline{M_1}.\overline{M_0} \\
&= \overline{\overline{(S_1.M_1)}.\overline{(\overline{M_1}.S_0)}.\overline{(\overline{M_1}.\overline{M_0})}} && (4.2) \\
neg_{out} &= (M_1 \oplus M_0).\overline{pos_{in}} + M_1.S_0.M_0 \\
&= \overline{\overline{((M_1 \oplus M_0).\overline{pos_{in}})}.\overline{(M_1.S_0.M_0)}} && (4.3) \\
\overline{neg_{out}} &= \overline{\overline{\overline{((M_1 \oplus M_0).\overline{pos_{in}})}.\overline{(M_1.S_0.M_0)}}} && (4.4) \\
S_{out} &= neg_{in} && (4.5) \\
M_{out} &= (((M_1 \oplus M_0) \oplus pos_{in}) \oplus neg_{in}) && (4.6) \\
\overline{M_{out}} &= \overline{(((M_1 \oplus M_0) \oplus pos_{in}) \oplus neg_{in})} && (4.7)
\end{aligned}$$

The logic equations (4.1 to 4.7) above have all been converted from AND/OR form to NAND/NAND form. Also, as the adders will be used in the address generator which is a critical bottleneck, each term is calculated with its conjugate, where the conjugate is required. This is only an inverter in the cases of $\overline{M_{out}}$ and $\overline{neg_{out}}$, but $\overline{pos_{out}}$ is computed simultaneously with pos_{out}. The other point to note is the order of evaluation of the term M_{out}. M_{out} is asserted if an odd number of the four inputs to a cell are asserted. A more balanced exclusive-or combination to achieve this function is

$$M'_{out} = ((M_1 \oplus M_0) \oplus (pos_{in} \oplus neg_{in}))$$

However, the terms M_1 & M_0 are available initially, pos_{in} from the previous stage is available only after two gate delays, and neg_{in} is only available a further two gate delays after pos_{in}. Hence, the ordering in Equation 4.6 uses the terms as they become available, resulting in a better balance and fewer spikes.

End Conditions

Although a single n–digit adder can be composed of n concatenated single digit adder cells, there are conditions for the first and last digits of the adders that simplify the requirements of those cells. For example, the first cell in a signed digit adder does not need the 'pos_in' and 'neg_in' propagate signals, and so these can be eliminated from the logic equations. In such a case, one of the several sets of possible Karnaugh maps for the output is:

[4]ESPRESSO is a Boolean logic minimization programme from the Octools design suite, written at the University of Califoria at Berkeley, at the Electronics Research Laboratory. Version 5.2 was used.

Pos_Out		S_0M_0			
		00	01	11	10
S_1M_1	00	0	1	0	0
	01	0	1	0	0
	11	0	1	0	0
	10	0	1	0	0

Figure 4.6: **a)**

Neg_Out		S_0M_0			
		00	01	11	10
S_1M_1	00	0	0	0	0
	01	0	0	0	0
	11	1	1	1	1
	10	0	0	0	0

b)

S_Out		S_0M_0			
		00	01	11	10
S_1M_1	00	0	1	1	0
	01	0	1	1	0
	11	0	1	1	0
	10	0	1	1	0

Figure 4.6: c)

M_Out		S_0M_0			
		00	01	11	10
S_1M_1	00	0	1	1	0
	01	1	0	0	1
	11	1	0	0	1
	10	0	1	1	0

d)

Figure 4.6: **Karnaugh Maps for First Digit Cell of an Adder**

The equations for these maps are:

$$\begin{aligned} pos_{out} &= \overline{S_0}.M_0 \\ &= \overline{S_0 + \overline{M_0}} \qquad (4.8) \\ neg_{out} &= S_1.M_1 \\ &= \overline{\overline{S_1} + \overline{M_1}} \qquad (4.9) \\ S_{out} &= M_0 \qquad (4.10) \\ M_{out} &= M_0 \oplus M_1 \qquad (4.11) \end{aligned}$$

If the number being applied to a signed-digit adder is a two's complement number, then the conversion to signed digit notation requires that there is a 'neg_in' signal, but a 'pos_in' signal is not required (see Section 4.1.3). This is the case for the adders in the address generators, which used two's complement notation for the 'delta's (see Chapter 5). A second requirement for the adders in the address generators is that the number 'zero' has an exclusive positive representation, ie the number '±0' must be represented as '+0'. This constraint, added to the lack of a 'pos_in' signal, produces the following set of possible Karnaugh maps.

The equations for this cell are as shown below in Equations 4.12 to 4.15.

$$\begin{aligned} pos_{out} &= \overline{S_0}.M_0 \\ &= \overline{S_0 + \overline{M_0}} \qquad (4.12) \\ neg_{out} &= S_1.M_1 + \overline{M_1}M_0 \\ &= \overline{\overline{S_1.M_1} + \overline{\overline{M_1}M_0}} \qquad (4.13) \\ S_{out} &= neg_{in}.(M_1 \oplus M_0) \qquad (4.14) \\ M_{out} &= (M_1 \oplus M_0) \oplus neg_{in} \qquad (4.15) \end{aligned}$$

Similarly, the last digit cell in the adder is not required to generate the 'pos_out' and 'neg_out' signals, although these may be of use if overflow detection is required. However, in some cases such as the multiplier, there may be only one input to an adder cell, plus the propagate signals. The simplifications thus made are shown in Figure 4.8 and in Equations 4.16 to 4.19.

Pos_Out		S_0M_0			
		00	01	11	10
S_1M_1	00	0	1	1	0
	01	0	1	1	0
	11	0	1	1	0
	10	0	1	1	0

Figure 4.7: **a)**

Neg_Out		S_0M_0			
		00	01	11	10
S_1M_1	00	0	1	1	0
	01	0	0	0	0
	11	1	1	1	1
	10	0	1	1	0

b)

S_Out		S_0M_0				S_0M_0					
		00	01	11	10	10	11	01	00		
S_1M_1	00	0	0	0	0	1	0	0	1	00	S_1M_1
	01	0	0	0	0	0	1	1	0	01	
	11	0	0	0	0	0	1	1	0	11	
	10	0	0	0	0	1	0	0	1	11	
Neg_In		0				1					

Figure 4.7: **c)**

M_Out		S_0M_0				S_0M_0					
		00	01	11	10	10	11	01	00		
S_1M_1	00	0	1	1	0	1	0	0	1	00	S_1M_1
	01	1	0	0	1	0	1	1	0	01	
	11	1	0	0	1	0	1	1	0	11	
	10	0	1	1	0	1	0	0	1	11	
Neg_In		0				1					

Figure 4.7: d)

Figure 4.7: **Karnaugh Maps for First Digit Cell of an Adder**

Pos_Out		$S_{in}M_{in}$			
		00	01	11	10
$pos_{in}neg_{in}$	00	0	1	0	0
	01	0	1	0	0
	11	0	1	0	0
	10	0	1	0	0

Figure 4.8: **a)**

Neg_Out		$S_{in}M_{in}$			
		00	01	11	10
$pos_{in}neg_{in}$	00	0	1	1	0
	01	0	1	1	0
	11	0	0	0	0
	10	0	0	0	0

b)

S_Out		$S_{in}M_{in}$			
		00	01	11	10
$pos_{in}neg_{in}$	00	0	0	0	0
	01	1	1	1	1
	11	1	1	1	1
	10	0	0	0	0

Figure 4.8: c)

M_Out		$S_{in}M_{in}$			
		00	01	11	10
$pos_{in}neg_{in}$	00	0	1	1	0
	01	1	0	0	1
	11	0	1	1	0
	10	1	0	0	1

d)

Figure 4.8: **Karnaugh Maps for Last Digit Cell of an Adder with Single Input**

$$\begin{aligned} pos_{out} &= \overline{S_{in}}.M_{in} \\ &= \overline{S_{in} + \overline{M_{in}}} \end{aligned} \tag{4.16}$$

$$\begin{aligned} neg_{out} &= \overline{pos_{in}}.M_{in} \\ &= \overline{pos_{in} + \overline{M_{in}}} \end{aligned} \tag{4.17}$$

$$S_{out} = neg_{in} \tag{4.18}$$

$$M_{out} = (M_{in} \oplus pos_{in}) \oplus neg_{in} \tag{4.19}$$

Additionally, the final propagated outputs, 'pos_out' and'neg_out', can be converted from signed bit form to sign magnitude form, and applied as inputs to another adder. Rather than generating pos_out and neg_out, the sign magnitude outputs, 'S2' and 'M2', can be generated directly. The logic equations for the new outputs are

$$S2_{out} = \overline{pos_{in}} \tag{4.20}$$

$$M2_{out} = (S_{in} \oplus pos_{in}).M_{in} \tag{4.21}$$

with S_{out} & M_{out} the same as in Equations 4.18 & 4.19.

4.1.5 VLSI Layout and Implementation

Three separate signed digit adders were ultimately implemented, two of which incorporated latches into the adder to facilitate cyclical addition/accumulation[5] and pipelining[6], and one of which incorporated multiplexers after the latches[7]. The simplest contained no latches or multiplexers, and is used in a non-pipelined multiplier[8].

The unlatched adder, the simplest of all, is shown in Figure 4.9. The input sign bits are applied to the top of the cell, and do not require a bus line as there are only three connections between the two inputs. The input magnitude bits occupy the bottom four internal bus lines, which can be connected to by either running metal2 down through the cell or by the side of the cell. The 'pos_in' and 'neg_in' signals are applied at the left hand side, and the generated 'pos_out' and 'neg_out' signal are output on the right hand side. The signed digit output number is generated at the bottom of the cell.

The overall size of the cell when implemented in 0.7μm es2 CMOS is approximately $0.089\text{mm}\times 0.103\text{mm} = 0.0092\text{mm}^2$. The cell was simulated using HSpice for all possible input combinations of the propagated and applied signals, with the propagated input signals being delayed by the same amount as it was determined for the output propagated signals to be generated. The adder was found to function correctly with a cycle time of just under five nanoseconds. The HSpice simulation results are shown in Figure 4.10. The es2 process is a double-metal, single polysilicon process with a 0.7 micron feature size. The simulations were run using 'typical' parameters throughout, with a five volt supply and assumed temperature of twenty-five degrees Celsius. The es2 process is from European Silicon Structures, which is available to research institutions from TIMA-CMP, in France.

The latched adder is shown in Figure 4.12. The latches are standard pass-transistor and two inverter designs, as shown in Figure 4.11. The input data is latched into the adder, and then applied to internal buses from which the outputs are derived. The data flow is again left

[5]as needed in Chapter 5 for address generation
[6]for a multiplier
[7]for selection of the correct address after the modulo operation
[8]required for the byte divider of Section 4.2

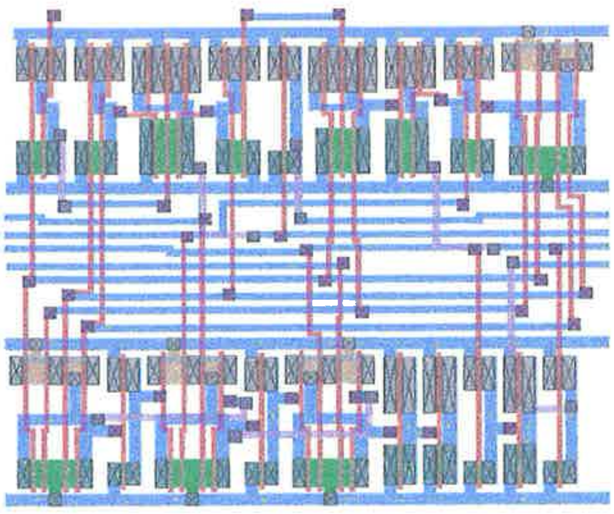


Figure 4.9: **Unlatched Signed-Digit Adder Cell**

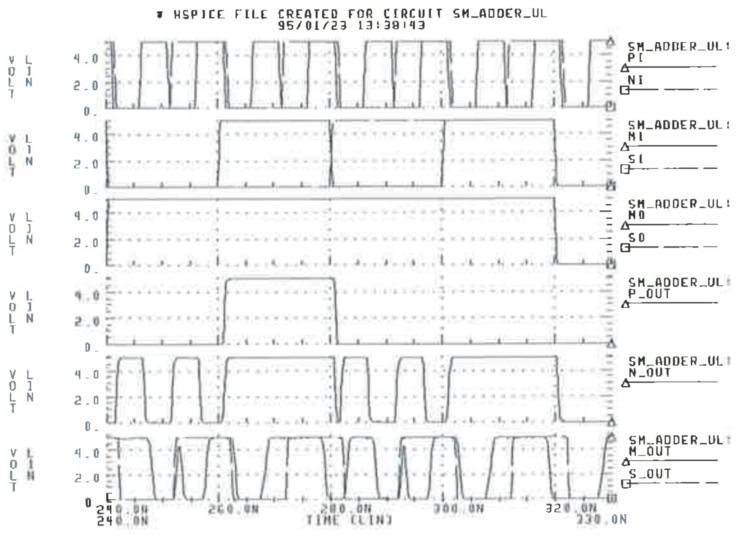


Figure 4.10: **HSpice Plots for Unlatched Adder Cell**

to right for the propagate signals (pos_in, pos_out, neg_in, neg_out), and top to bottom for the generated signals (s_in, s_out, m_in, m_out). For the 0.7μm es2 CMOS process, the size of a latched adder cell is approximately 0.097mm $\times$ 0.24mm = 0.023mm^2, which is approximately three times larger than the unlatched version. Note, however, that the latched version contains much more unused area that would be eliminated during a compaction phase.

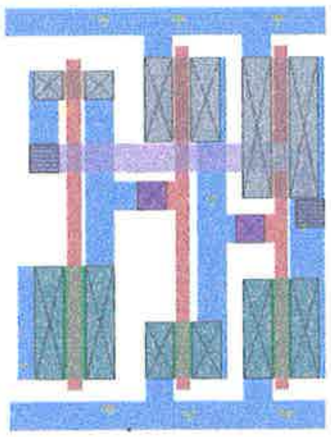


Figure 4.11: **Layout of Latch Cell**

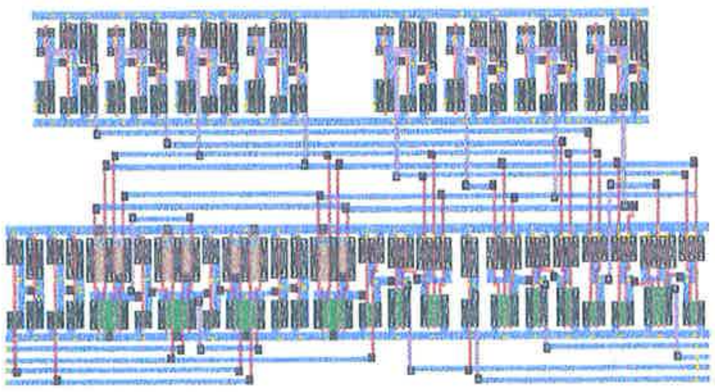


Figure 4.12: **Latched Signed-Digit Adder Cell**

The HSpice simulations of the latched adder indicated that the functionality was correct, and that a new addition could commence every five nanoseconds. The simulation results are shown in Figure 4.13.

The third adder cell was one designed specifically for the address generators described in Chapter 5. These include a pair of multiplexers at one of the inputs after the latch. The reason for this is to allow the multiplexer control signal to settle before being used, and to reduce the driving requirements on the output drivers of the adder cell.

The layout of the third adder is shown in Figure 4.14. The adder was also tested using HSpice, and found to operate correctly.

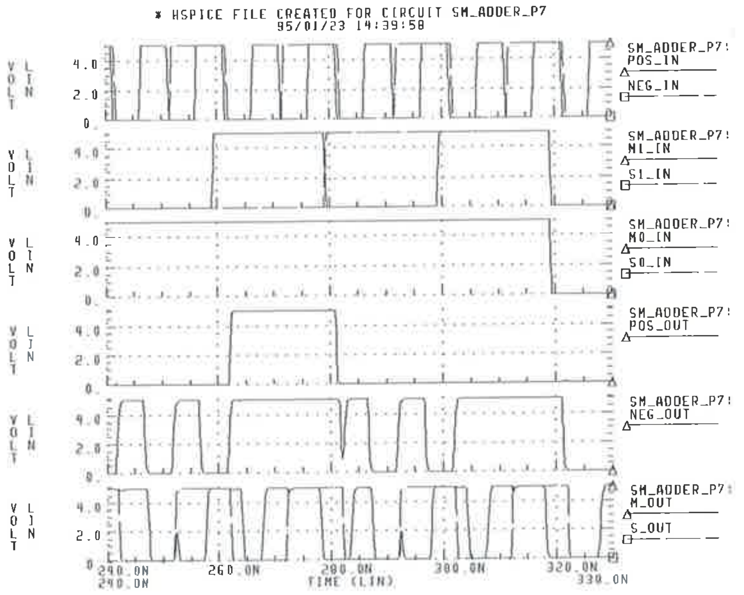


Figure 4.13: **HSpice Simulation for Latched Adder Cell**

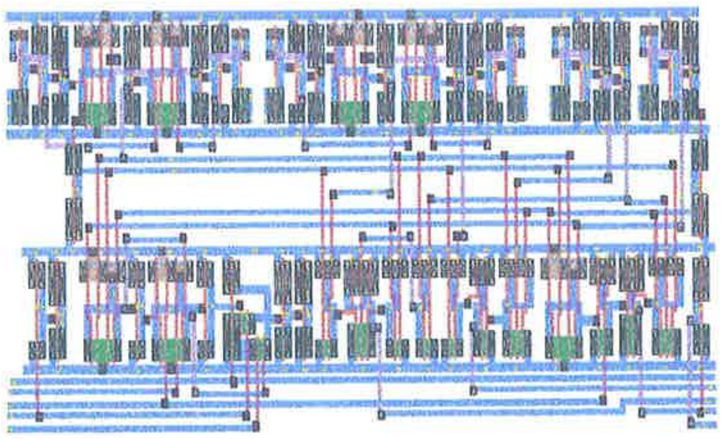


Figure 4.14: **Latched Adder Cell With Multiplexer**

VLSI Implementation of End Cells

The first cells and final cells described above in Section 4.1.4 were all laid out in 0.7μm es2 CMOS. The unlatch beginning cell is shown in Figure 4.15, and the HSpice simulation results are shown in Figure 4.16.

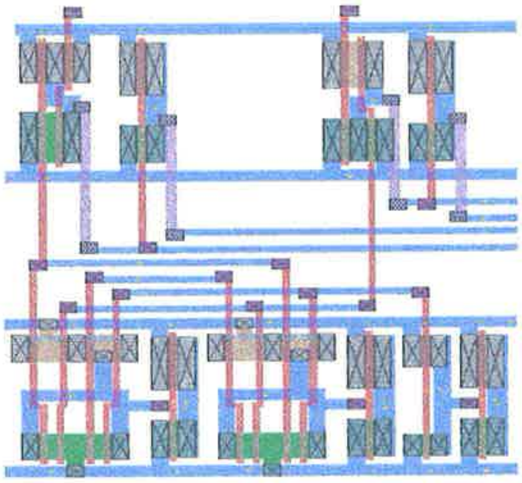


Figure 4.15: **Layout of Unlatched Beginning Cell**

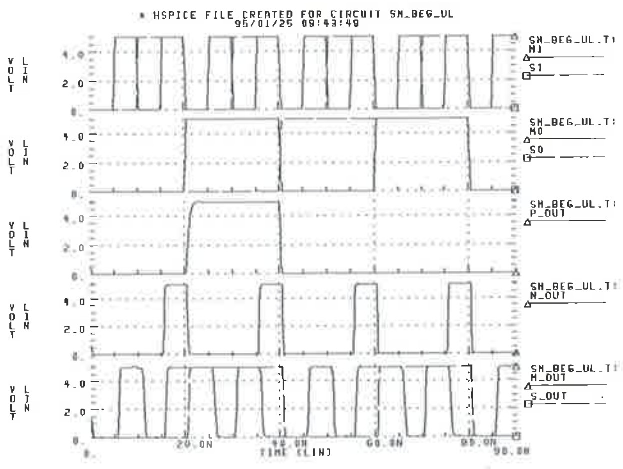


Figure 4.16: **HSpice Simulation Results for Unlatched Beginning Cell**

The cell is approximately 0.089mm × 0.061mm = 0.0054mm^2 in area. All the output signals are available less than two nanoseconds after the inputs are applied.

If the latches are added to the cell, a significant extra area is required. In this case, the latches consume approximately 31 percent of the total area. However, as the cell will be only a small part of a complete adder, the excess area is not a major concern. The layout of the latched beginning cell is shown in Figure 4.17.

When the multiplexers and the facility to apply a two's complement number are added for the cells used in the address generator, the area penalty of adding latches is mitigated. The

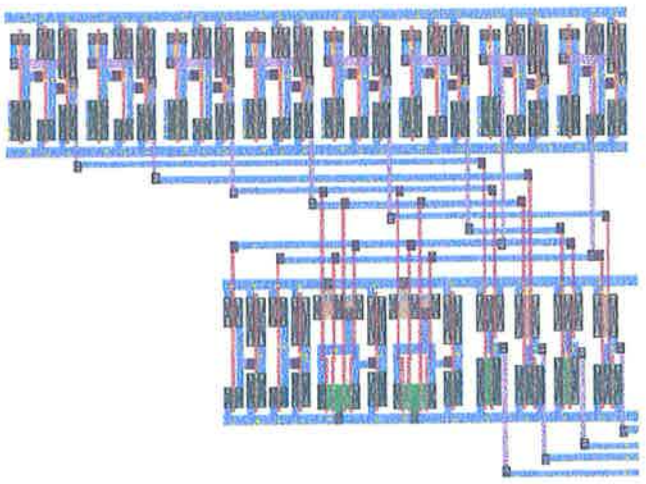


Figure 4.17: **Layout of Latched Beginning Cell**

VLSI layout of this beginning cell is shown in Figure 4.18.

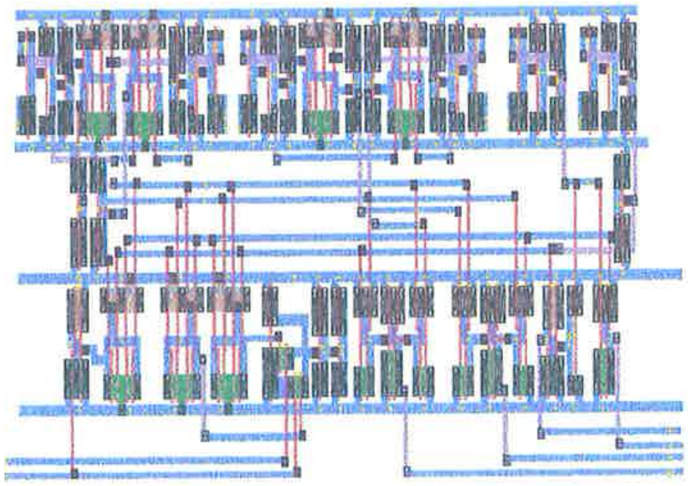


Figure 4.18: **Layout of Latch & Multiplexed Input Beginning Cell**

The area consumed by the cell is $0.10\text{mm} \times 0.24\text{mm} = 0.024\text{mm}^2$, which is approximately 30 percent more than for the latched beginning cell, and about four times the area of the unlatched cell.

The end cells, denoted sm_end and sm_end2, depending on whether the propagated outputs are plus two and minus two ($\{-2,+2\}$) or a sign and magnitude bit ($\{\pm,2\}$) are shown in Figures 4.19a) & b) respectively for the unlatched versions. Both cells consume an area of $0.0892\text{mm} \times 0.0785\text{mm} = 0.00700\text{mm}^2$ when implemented in 0.7μm es2 CMOS.

The end cells were both simulated using HSPICE, and were found to operate correctly. The simulation results are shown in Figures 4.20 a) & b).

As can be seen from these plots, the end cells all settle well before five nanoseconds, even for a propagated input delay of 1.5 nsec.

To pipeline the multiplier, the latched version of the adder was modified to produce a latched

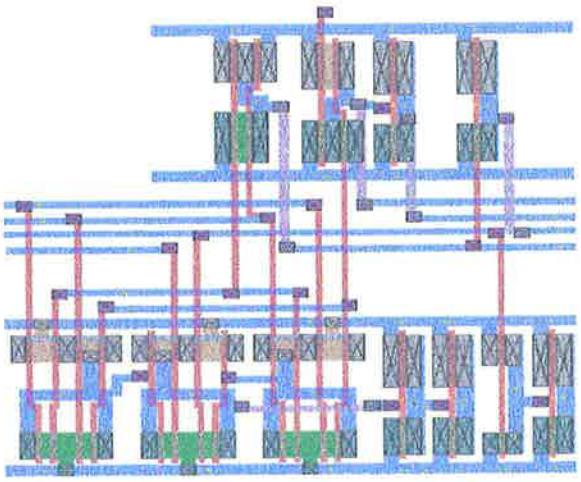


Figure 4.19: a) **Unlatched End Cell With Standard Propagated Outputs**

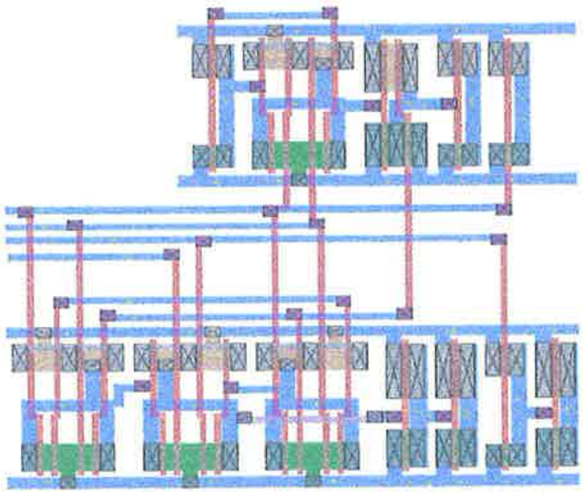


Figure 4.19: b) **Unlatched End Cell With Sign Magnitude Progopated Outputs**

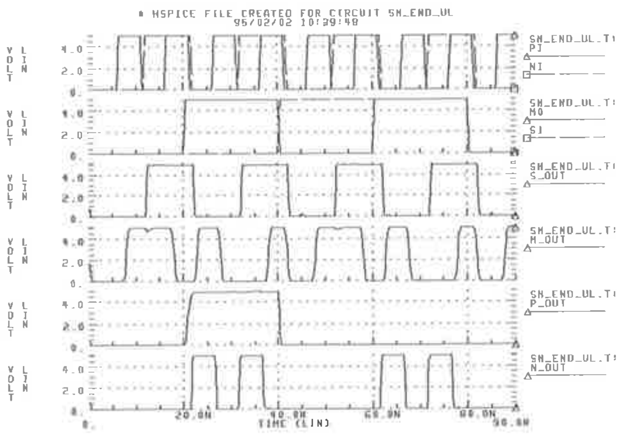


Figure 4.20: a) **Unlatched End Cell With Standard Propagated Outputs Simulation Results**

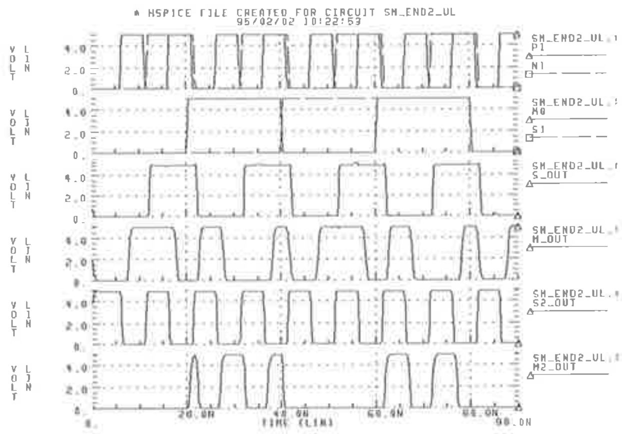


Figure 4.20: b) **Unlatched End Cell With Sign Magnitude Progopated Outputs Simulation Results**

end cell. The VLSI layout of the latched end cell with standard propagate output signal is shown in Figure 4.21a), and with sign magnitude outputs in Figure 4.21b). When implemented in 0.7μm es2 CMOS, they consume 0.0972mm $\times$ 0.1196mm $=$ 0.0116mm^2 and 0.0972mm $\times$ 0.1288mm $=$ 0.0125mm^2 of silicon respectively.

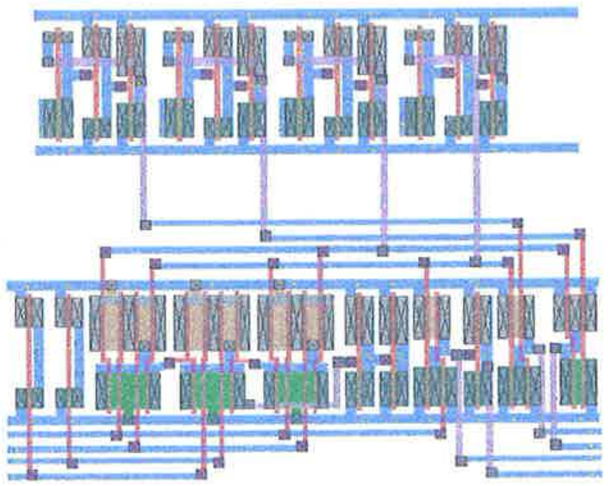


Figure 4.21: a) **Latched End Cell With Standard Propagated Outputs**

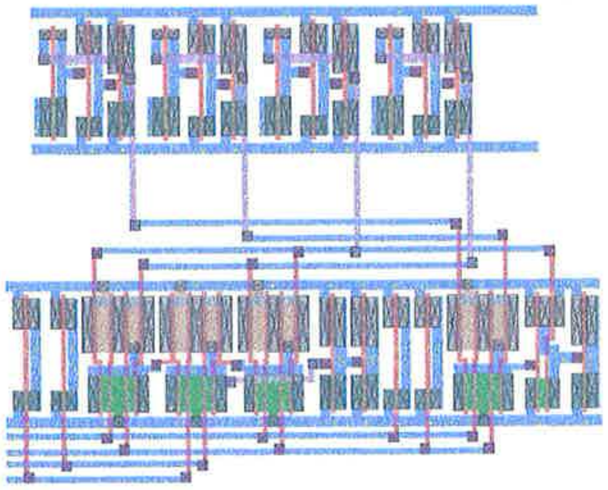


Figure 4.21: b) **Latched End Cell With Sign Magnitude Progopated Outputs**

The HSpice simulations of the latched end cells are shown in Figures 4.22a) & b). These simulations show that the cycle time between latched operations is less than five nanoseconds, as desired.

4.2 Byte Divider

As mentioned in Chapter 3 on Algorithms, if Gaussian elimination is to be run on the matrix engine, some form of divider is required to find the inverse of the pivot elements. The requirement here is for an area-efficient divider which is relatively fast. In effect, if the matrix engine is to run at full speed, a divider capable of one division every P loads (where P is the dimension of

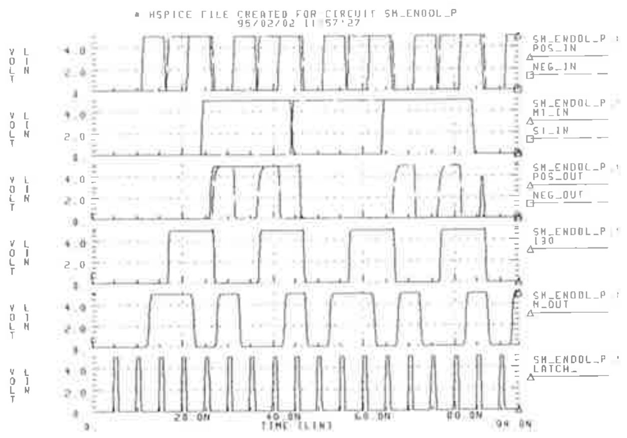


Figure 4.22: a) **Latched End Cell With Standard Propagated Outputs Simulation Results**

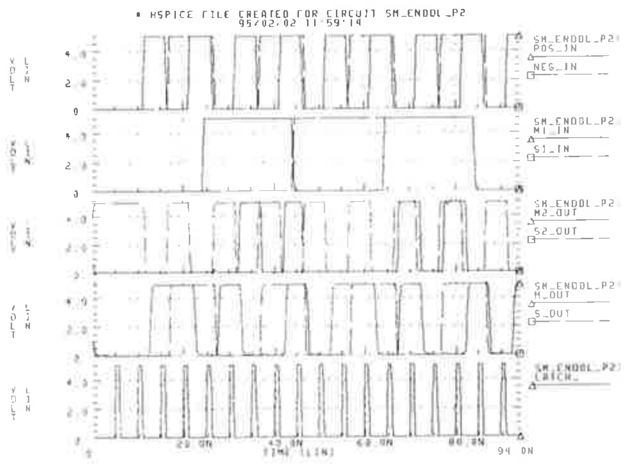


Figure 4.22: b) **Latched End Cell With Sign Magnitude Progopated Outputs Simulation Results**

the matrix array) is required. For a typical system, this may be in the order of 150 to 400 nsec, depending on array size, technology, etc.

A scheme that produces a divider that is both relatively small and also fast is one based on higher radix division. This is, in turn, based on the Newton-Raphson division algorithm that has been well documented [92, 32, 95]. A brief introduction to Newton-Raphson division is provided in the following Section, including starting and error considerations, followed by the conversion of this approach to higher radix division.

4.2.1 Newton-Raphson Iterative Divider

The concept behind Newton's iteration for root finding is that a series of iterations of a basic formula will produce an approximation to the true root of a function at each iteration that was a better approximation than the last. If the exact solution to the root is at $\overline{x}$, such that $f(\overline{x}) = 0$, and the i^{th} approximation to the root is x_i, then the next approximation to the root will be the old approximation plus a difference term.

$$x_{i+1} = x_i + \Delta x_i \tag{4.22}$$

The slope of a line draw from the function at the initial estimate to the function at the subsequent estimate will be:

$$m_i \approx -\frac{f(x_i)}{\Delta x_i} \tag{4.23}$$

where the negative sign is due to the function at the subsequent estimate being much closer to zero than at the initial estimate.

If the slope, m_i, is made to be the tangent of the function at the approximation for $\overline{x}$ such that

$$m_i = f'(x_i) \tag{4.24}$$

then the equation for the next approximation to $\overline{x}$ becomes

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.25}$$

Equation 4.25 is the Newton-Raphson formula for root finding.

To apply Equation 4.25 to the case of division, it is first necessary to formulate the problem in terms of finding the root of a function. The equation

$$f(x) = \frac{1}{x} - b \tag{4.26}$$

has a root at the point $x = 1/b$, and so can be used to find the inverse of the number represented by 'b'. Noting that $f'(x) = -\frac{1}{x^2}$, Equation 4.25 can be formulated as

$$\begin{aligned} x_{i+1} &= x_i - \left(\frac{f(x_i)}{f'(x_i)}\right)\left(x_i^2\right) \\ &= x_i(2 - x_i b) \end{aligned} \tag{4.27}$$

Then Equation 4.27 successively approximates the inverse of the number 'b' to the actual inverse $\overline{x}$.

For the sake of consistency with other texts, it is often the case that the 'x' variable in Equation 4.27 is replaced by 'q' for the specific case of the division formulation of Newton's iteration, and the *exact* inverse of b is denoted as 'Q'. The new representation is:

$$q_{i+1} = q_i(2 - q_i b) \tag{4.28}$$

Division of Floating Point Numbers

To apply Equation 4.28 to a floating point number, it is necessary to consider the range of the divisor and the quotient.The number will be composed of a mantissa field, a sign bit, a 'hidden' leading one and an exponent field.

As the mantissa is a fractional number, the mantissa and the hidden bit together comprise a number in the range $1 \leq 1.f < 2$, where the term 'f' represents the mantissa. The inverse, therefore, is in the range $\frac{1}{1} \geq \frac{1}{1.f} > \frac{1}{2}$, ie $0.5 < inv \leq 1$. The new exponent, of course, is simply the negative of the old exponent.

Self–Correction and Starting of Newton's Iteration

In the past, a Newton's Iteration divider often obtained its initial guess at the solution using a Read Only Memory (ROM) look-up table. However, considering the speed improvement that has occurred recently in CMOS VLSI, the speed advantage offered by such a table may not justify the amount of area it consumes. An alternative approach is to guess an approximate solution, and rely on the self correcting property of the algorithm.

Consider the '$(i+1)^{th}$' iteration that produces the estimate of the inverse 'q_{i+1}', for which the error is 'ϵ_{i+1}'. If the correct value of the inverse of 'b' is 'Q', then the error after the $(i+1)^{th}$ iteration can be obtained using Equation 4.28.

$$\begin{aligned}
q_{i+1} &= q_i(2 - bq_i) \\
&= (Q + \epsilon_i)(2 - b(Q + \epsilon_i)) \\
&= (Q + \epsilon_i)(2 - bQ - b\epsilon_i) \\
&= 2Q + 2\epsilon_i - Q.1 - \epsilon_i.1 - 1.\epsilon_i - b\epsilon_i^2 \qquad (Q \times b = 1) \\
&= Q - b\epsilon_i^2 \qquad (4.29)
\end{aligned}$$

Therefore, the precision of each iteration is the square of the precision of the iteration before it.

The important thing to note is that Equation 4.29 always gives the same answer irrespective of the sign of ϵ, and that the $b\epsilon_i^2$ term is prefaced by a minus sign. Therefore, even after an incorrect initial guess that estimates the inverse above the true inverse, successive approximations will approach the true inverse from below.

This important fact can be used to find a 'good' starting point.for Newton's iteration. One possible starting point for the iteration is the guess $x_0 = 0.5$, as we know that all solutions will be above this point, and we are trying to approach the solution from below. This approach has the advantage of simplicity. We know that the first (fractional) bit must be one, so it's easy just to set it and let the self correction of Newton's method look after the relatively poor guess.

Another approach is to recognize that the larger the dividend, the smaller the quotient. Thus, an estimation scheme that returns the maximum possible quotient for the minimum dividend, and vice versa, may have some merit. Such a scheme is proposed in the text by Burgess [11], the choice of starting estimate being:

$$q_0 = \begin{cases} 1 - f & \text{if } 1 \leq b < 1.5 \\ 1 - \frac{f}{2} & \text{otherwise} \end{cases} \qquad (4.30)$$

where 'f' is the fraction part of b.

The error associated with each can be easily calculated, as shown below in Equations 4.31 & 4.32. Here, 'b' is the number to be inverted, q_0 is the initial estimate and Q is the actual inverse. If the initial estimate is chosen to always be $q_0 = 0.5$, then the error is a linear function dependent on the actual quotient, ie

$$\begin{aligned}\varepsilon_0 &= Q - q_0 \\ &= Q - 0.5 \qquad (4.31)\end{aligned}$$

Alternatively, if the initial estimate is chosen to be as defined in Equation 4.30, then the error is as shown below in Equation 4.32 [9].

$$\begin{aligned}\delta_0 &= Q - q_0 \\ &= Q - \begin{cases} 1 - f & \text{if } 1 \leq b < 1.5 \\ 1 - \frac{f}{2} & \text{otherwise} \end{cases} \\ &= Q - \begin{cases} 1 - (b - 1) & \text{if } 1 \leq b < 1.5 \\ 1 - \frac{b-1}{2} & \text{otherwise} \end{cases} \\ &= Q - \begin{cases} 2 - \frac{1}{Q} & \text{if } 1 \leq b < 1.5 \\ \frac{3}{2} - \frac{1}{2Q} & \text{otherwise} \end{cases} \qquad (4.32)\end{aligned}$$

Comparing the errors between the two starting schemes graphically leads to the plots either separately in Figure 4.23, or together in Figure 4.24. Note that only the magnitudes of the errors are plotted, not the signs.

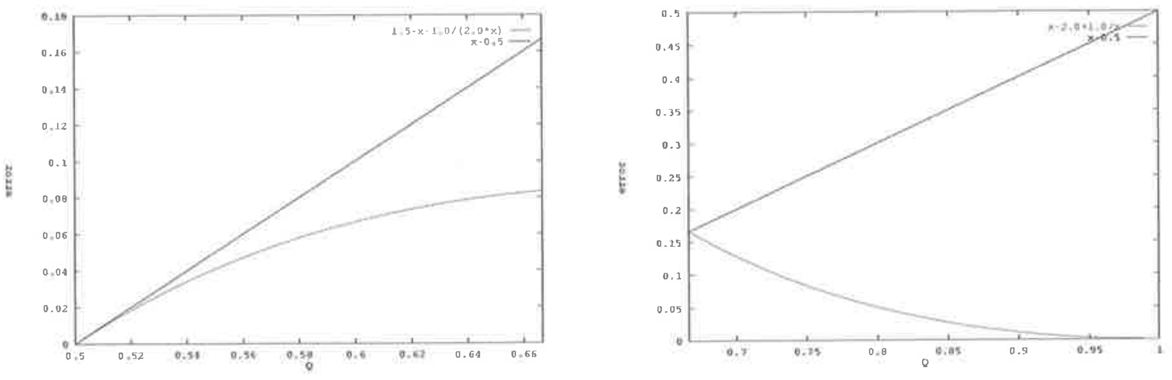


Figure 4.23: **Error graphs for a)** $0.5 < Q \leq \frac{2}{3}$ **b)** $\frac{2}{3} < Q \leq 1$

As can be seen in these figures, the difference between the schemes is quite large. This is apparent in Figure 4.25.

Although the initial estimate of $q_0 = 0.5$ is very simple, using the other starting estimate method (Equation 4.30) is not much more complicated, and produces a much better estimate. The error graphs shown in Figure 4.24 also indicate that the complexity of Equation 4.30 can be reduced with a minor estimate accuracy penalty by noting that the choice $q_0 = 1 - \frac{f}{2}$ has error characteristics over the range $1.0 \leq b < 1.5$ ($\frac{2}{3} < Q \leq 1$) that are very similar to the choice $q_0 = 1 - f$ over the same range. Recalling that the error in the initial estimate is

$$\delta_0 = Q - \begin{cases} 2 - \frac{1}{Q} & \text{if } 1 \leq b < 1.5 \\ \frac{3}{2} - \frac{1}{2Q} & \text{otherwise} \end{cases}$$

[9]The variables ε and δ are used for clarity's sake. They are both the same error variables, but applied to different functions.

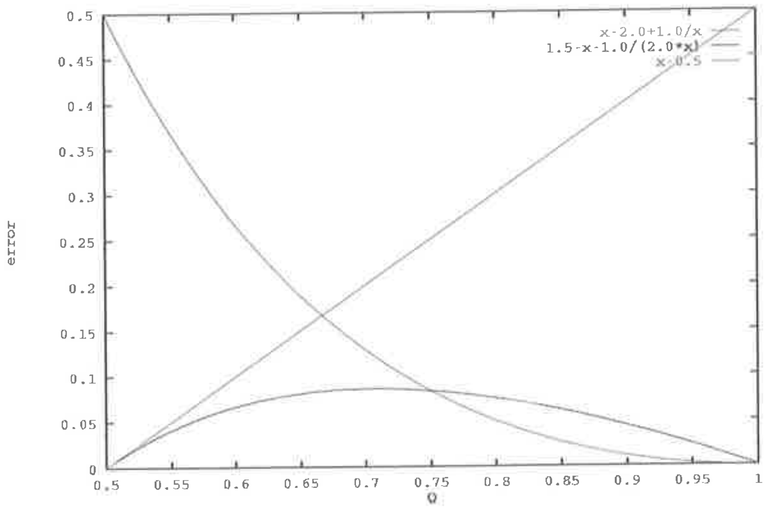


Figure 4.24: **Error graph over full range of Q**

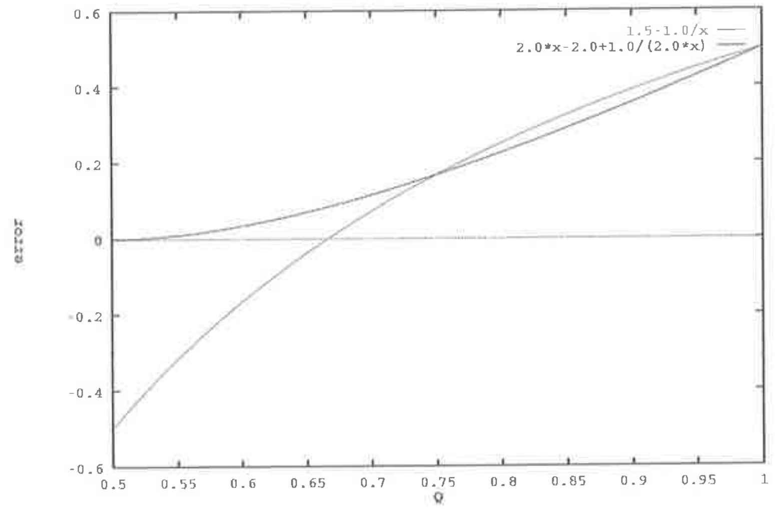


Figure 4.25: **Difference in Errors for Two Schemes**

then the integrals of the errors for the two choices over the range $\frac{2}{3} \leq Q < 1$ are:

$$\begin{aligned}
\int_{\frac{2}{3}}^{1} \delta_0 dQ &= \begin{cases} \int_{\frac{2}{3}}^{1}\left(Q - 2 + \frac{1}{Q}\right) dQ & \text{if } q_0 = 1 - f \\ \int_{\frac{2}{3}}^{1}\left(Q - 1.5 + \frac{1}{2Q}\right) dQ & \text{if } q_0 = 1 - \frac{f}{2} \end{cases} \\
&= \begin{cases} \left[\frac{1}{2}Q^2 - 2Q + \ln Q\right]_{\frac{2}{3}}^{1} & \text{if } q_0 = 1 - f \\ \left[\frac{1}{2}Q^2 - \frac{3}{2}Q + \frac{\ln Q}{2}\right]_{\frac{2}{3}}^{1} & \text{if } q_0 = 1 - \frac{f}{2} \end{cases} && (4.33) \\
&= \begin{cases} 0.0166 & \text{if } q_0 = 1 - f \\ -0.0195 & \text{if } q_0 = 1 - \frac{f}{2} \end{cases} && (4.34)
\end{aligned}$$

The integral of the error for the range $0.5 \leq Q < \frac{2}{3}$ is

$$\begin{aligned}
\int_{0.5}^{\frac{2}{3}} \delta_0 &= \int_{0.5}^{\frac{2}{3}} \left(Q - \frac{3}{2} + 12Q\right) dQ \\
&= \left[\frac{1}{2}Q^2 - \frac{3}{2}Q + \frac{1}{2}\ln Q\right]_{0.5}^{\frac{2}{3}} \\
&= -0.00894 && (4.35)
\end{aligned}$$

If the magnitudes of the error integral terms are added, then the total error integral using Equation 4.30 directly will be $0.0166 + 0.00894 = 0.0255$. If the modified version of Equation 4.30 is used such that q_0 always equals $1 - \frac{f}{2}$, then the total error integral will be $0.0166 + 0.00894 = 0.0284$. Therefore, using the simplified starting approximation of always assuming $q_0 = 1 - \frac{f}{2}$ will result in only an 11% increase in the starting error. Just for comparison sake, if the initial guess is chosen as $q_0 = 0.5$, then the total error integral is

$$\int_{0.5}^{1} (Q - 0.5)\, dQ = 0.125$$

which is more than four times larger than using $q_0 = 1 - \frac{f}{2}$. Thus a good compromise in minimising error and reducing complexity would be to use the estimate $q_0 = 1 - \frac{f}{2}$ always.

Convergence

The convergence of an iterative method of root finding is the rate at which the approximation converges to the actual solution, if it does at all. As the rate of reduction of error is related to the rate of convergence, then Equation 4.29, which was used to determine the error between any estimate and the actual solution, can also be used to determine the rate of convergence and also if convergence will actually occur. Equation 4.29 is reproduced below for convenience.

$$q_{i+1} = Q - b\epsilon_i^2$$

Also recall that the estimate q_i is different form the exact answer by an amount ϵ_i such that

$$\epsilon_i = Q - q_i \qquad (4.36)$$

were ϵ_i will in general be positive, except for the cases of either $i = 0$ (q_i is an initial estimate) or a numerical error has occurred. It is not a requirement that ϵ_i be positive.

The error term after each iteration in terms of the previous error is obviously

$$\begin{aligned}\epsilon_{i+1} &= Q - q_{i+1} \\ &= Q - \left(Q - b\epsilon_i^2\right) \\ &= b\epsilon_i^2\end{aligned}$$

Therefore, the error reduces *quadratically* if the estimate to the actual answer is sufficiently close. If the error reduces quadratically, then the precision *increases* quadratically at each iteration. Hence, if the initial estimate is correct to two places of precision, the first correction will be correct to four places, the next will be correct to eight, etc. If 32 place of precision was required, then four iterations would be required, while 64 places of precision would require five iterations.

4.2.2 Byte Divider

This is, in effect, division using a higher radix. The term 'byte divider' refers to the fact that past processors by Amdahl and Honeywell used an eight bit (one byte) radix [92]. The algorithm for division presented here is based on the formulae given by Flynn & Waser [92], although subsequent to the design of the divider, it was noted that Briggs and Matula presented a similar design at the Eleventh Symposium on Computer Architecture [10].

Consider Equation 4.28, reproduced below for convenience,

$$q_{i+1} = q_i\,(2 - q_i b)$$

The impetus for the higher radix division comes when it is considered what the precision of the intermediate, or early iteration, results will be. If the first iteration has one bit of precision, the multiplication $b \times q_i$ is an $N \times 1$ product, where N is the precision of b. In the second iteration, an $N \times 2$ multiplier will be required, in the third an $N \times 4$ multiplier, and so on, until an $N \times N$ multiplier is required. However, as the solution is approached from below, once the first 'P' bits of precision have been obtained, there is no need to recalculate them.

Once the first byte has been obtained, call it d_0, then it can effectively be 'retired' from the calculation, as it will not be modified again, with the exception of a possible carry-propagate in. Similarly, when the second byte, d_1, has been obtained, it will remain unchanged except for a possible carry-propagate in. Therefore, if we rearrange Newton's equation for division such that we calculate the *difference* between successive iterations, then we can arrange that the difference is, in fact, one byte long. Rearranging the formula is not at all difficult, as it is simply a matter of shifting a q_i from the right hand side to the left, ie.

$$\begin{aligned}d_{i+1} = q_{i+1} - q_i &= q_i(2 - b.q_i) - q_i \\ &= q_i(1 - b.q_i)\end{aligned} \tag{4.37}$$

so that

$$\begin{aligned}q_i &= q_{i-1} + d_i \\ &= (q_{i-2} + d_{i-1}) + d_i \\ &= \vdots \\ &= d_0 + d_1 + \ldots + d_i \quad (d_0 \equiv q_0) \\ &= \sum_{j=0}^{i} d_j\end{aligned} \tag{4.38}$$

Now consider the difference term on the right hand side of Equation 4.37. Substituting in the result of Equation 4.38 gives

$$\begin{aligned}
1 - b.q_i &= 1 - b.(d_0 + d_1 + \ldots + d_i) \\
&= 1 - b.(d_0 + d_1 + \ldots + d_{i-1}) - b.d_i
\end{aligned}$$

Defining $z_i = 1 - b.q_i$, the difference term, then

$$\begin{aligned}
z_i &= 1 - b.q_i \\
&= 1 - b.(d_0 + d_1 + \ldots + d_{i-1}) - b.d_i \\
&= z_{i-1} - b.d_i
\end{aligned} \tag{4.39}$$

Equation 4.37 now becomes

$$\begin{aligned}
d_{i+1} &= q_i.z_i \\
&= q_i.(z_{i-1} - b.d_i)
\end{aligned} \tag{4.40}$$

which is an $n \times 8$ product for an n–bit number 'b', followed by a sum $(z_{i-1} - b.d_i)$, which is stored for the next iteration, and finished by a product. As only the leading digit (byte) of z_i is used for the final product, then the final product is actually an 8×8 multiply. Once the difference d_{i+1} is obtained, the approximation to the quotient q_{i+1} can be formed. Note that, although the quotient is the sum of the estimates that are calculated at each iteration, the calculated d_i's are largely disjoint, and so any previous estimate of the quotient will only be effected if a 'carry-in' exists. Therefore, if the summing is achieved using signed digit addition, which has a maximum carry path of two digits and a constant addition time, then cycle times remain constant throughout the whole process and hence can be minimized.

Starting Byte Division

The procedure outlined in Section 4.2.2 above assumes an initial starting byte q_0 to use as a starting point for future iterations. Although this could be obtained from a look-up table, the table must contain approximately one thousand locations for eight bits of result (*not* $2^8 = 256$, as division is a non-linear operation). This sort of table would consume significant area and so an alternative method is sought.

The obvious answer is to use Newton-Raphson inversion to obtain the first eight bits, and then swap to byte division for the remaining $n - 32$ bits. If Newton-Raphson division is used to form a quotient estimate eight bits long, the $n \times 8$ multiplier used in byte division will be sufficient. Using the starting estimation schemes described above, two bits of precision are available initially. Two iterations of Newton's method will produce an eight bit result, requiring two passes through the $n \times 8$ multiplier and a single sum stage for each iteration. After a single byte of the quotient has been formed, the algorithm is switched to byte division, which requires two passes through the $n \times 8$ multiplier, and two additions.

Why Byte Division?

The advantage of using byte division is obvious when it is considered that the size of the multiplications that are required become very large towards the end of a standard Newton-Raphson type inversion, whereas they are a constant size for byte division. Assuming that a byte of quotient is available (either from table look-up or some other method) then full Newton's iteration will require two iterations to reach a 32-bit solution and three iterations to reach a 64-bit solution.

Again recalling Newton's iteration,

$$q_{i+1} = q_i\,(2 - b.q_i)$$

and considering the case of 'b' being $n-$bits of precision, then each iteration requires two $n \times n$ multiplications and one n-bit sum. To obtain the time advantage associated with a full Newton's iteration implementation, a full $n \times n$ multiplier would be required for the final iteration. However, for earlier iterations the multiplier would be under-utilised, performing $n \times \frac{n}{2}$, $n \times \frac{n}{4}$ etc. multiplications on a full $n \times n$ array. Therefore, the area penalty of providing a full $n \times n$ multiplier becomes significant.

Not only is the area required for the multiplier reduced by a factor of four for 32-bit division and by eight for 64-bit division when using byte division, in general the time required for a 32×8 or 64×8 multiplication is typically three fifths or one half the time required for a full square multiplication respectively[10]. Therefore, each iteration using byte division will generally run faster, although more iterations will be required.

Assuming that a 32-bit quotient is required, and that a 32×8 multiplier array takes 3/5 times the time to complete a product than a 32×32 array, that a full multiplier array cannot supply a solution except for at the final level of adders, and that an adder takes approximately the same time as a single level of a multiplier array, then a quantitative comparison of byte division against Newton's iteration can be made. The comparison is best made in terms of signed digit adder cell delays. Therefore, a single sum operation takes one adder delay, a 32×8 product takes three adder delays, and a 32×32 product takes five adder delays.

The first two bits of the quotient are available initially. Therefore, two iterations of Newton's method are required to produce an eight bit result. The byte division hardware requires the following time to produce eight bits of quotient:

$$\begin{aligned} T_{Byte\ Div.\ 8} &= 2 \times (T_{\text{add}} + 2T_{\text{mult}}) \\ &= 14T_{adder\ delays} \end{aligned} \tag{4.41}$$

whereas the Newton's iteration requires the same number of operations, but through the larger, slower multiplier:

$$\begin{aligned} T_{NR\ 8} &= 2 \times (T_{\text{add}} + 2T_{\text{mult}}) \\ &= 22T_{adder\ delays} \end{aligned} \tag{4.42}$$

To produce a 32-bit quotient, three iterations of byte division are required, using time:

$$\begin{aligned} T_{Byte\ Div.\ 32End} &= 3 \times (2T_{\text{add}} + 2T_{\text{mult}}) \\ &= 24T_{adder\ delays} \end{aligned} \tag{4.43}$$

for a total of 38 adder delays. Using Newton's iteration to produce the final 32-bit quotient, only two iterations are required, using time:

$$\begin{aligned} T_{NR\ End} &= 2 \times (T_{\text{add}} + 2T_{\text{mult}}) \\ &= 22T_{adder\ delays} \end{aligned} \tag{4.44}$$

[10]Using a signed digit multiplier array, a 32×8 array will use three levels of adders as opposed to five for a full 32×32 array. Similarly, a 64×8 array will still use three levels of adders compared to six for a full 64×64 array

Therefore, the time to produce a 32-bit result will be 44 adder delays, which is actually slightly *slower* than byte division.

If the quotient is a 64-bit number, then seven iterations of byte division are required after the initial eight bit result to produce the final answer. This will require time

$$\begin{aligned} T_{\mathit{Byte\ Div.\ 64End}} &= 7 \times (2T_{\text{add}} + 2T_{\text{mult}}) \\ &= 56T_{\mathit{adder\ delays}} \end{aligned} \tag{4.45}$$

for a total time of 70 adder delays.

Using Newton's method for the complete division will require five iterations, for a total time of:

$$\begin{aligned} T_{\mathit{NR\ 64}} &= 5 \times (T_{\text{add}} + 2T_{\text{mult}}) \\ &= 5 \times 13 \\ &= 65T_{\mathit{adder\ delays}} \end{aligned} \tag{4.46}$$

Therefore, on 64-bit division, Newton-Raphson division is marginally faster than byte division with a much greater area penalty.

4.3 8 $\times$ 8 Multiplier

As a multiplier is required as part of the iterative divider, a preliminary design was undertaken. In Section 3.4 on Gauss-Jordan elimination, it was pointed out that an inverse ($\frac{1}{b}$) is required every $(p+1)$ loads, where p is the dimension of the matrix array. However, as the divisor is the i^{th} element in the i^{th} wavefront, there are only $(p-i)$ load cycles (p is the matrix array dimension) available until the wavefront has been completely loaded. Therefore, to prevent the array from stalling, the multiplier needs to be relatively fast, and so a multiplier tree should be used.

A tree structure reduces the multiply time from order $O(n)$ to $O(\log n)$ for an $n-$bit multiply. For a 32$-$bit number, the reduction is a factor of $\frac{5}{32}$, while the factor becomes $\frac{6}{64}$ for double precision (64$-$bit). One very common multiplier tree that was investigated is the Wallace tree, shown in Figure 4.26. Here, CSA denotes a Carry-Save Adder and CLA denotes a Carry-Lookahead Adder. The Carry-Lookahead Adder is needed to convert the solution from CSA form to standard binary.

The Wallace tree was dropped in favour of a binary tree using signed-digit arithmetic. The three reasons for this were

- The signed digit adder cells had already been designed, and these provided the small propagation delays typical of redundant arithmetic systems.
- Although signed-digit notation requires two bits to represent three numbers $(\bar{1}, 0, 1)$, the Carry-Save Adders in Figure 4.26 require more hardware than propagation adders such as the CLA or Carry-Select, as carry bits must be saved at the outputs of each register. Therefore, the area for both the Wallace tree and the signed-digit multiplier tree are both order $O(n^2 \log n)$ [91].
- As the multiplier will be used in an iterative divider, the output will be fed back into the inputs, after suitable additions. The major disadvantage of signed-digit notation, that of conversion back to standard binary, can be left to the end of the division process rather than after each operation.

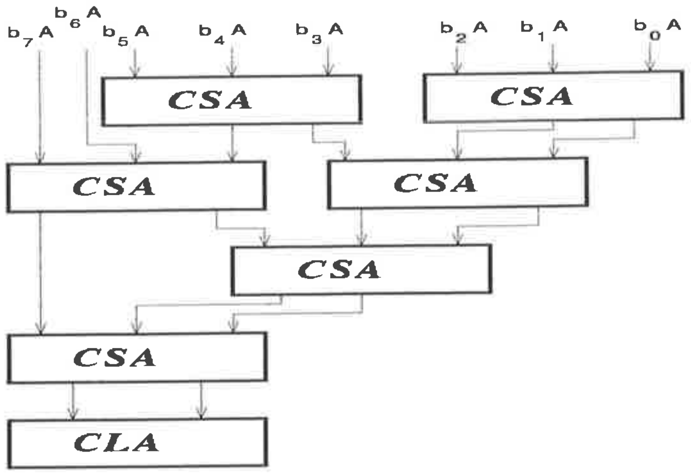


Figure 4.26: **Wallace-tree Multiplier.**

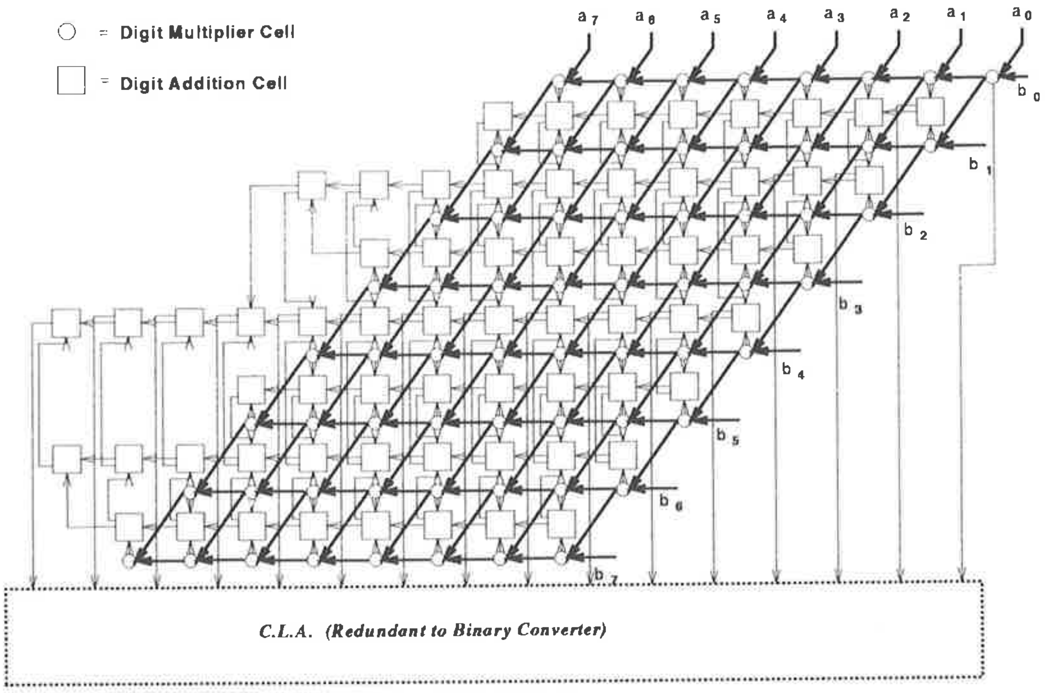


Figure 4.27: **Signed Digit Multiplier.**

The multiplier implemented is the one by Takagi *et al.* [91]. This is a high speed binary-tree using signed digit arithmetic. A block diagram of the implementation is given in Figure 4.27.

The digit addition cells are single bit adders as described in Section 4.1. They take two input digits and two carry bits, one generated in the previous cell, the other in the next previous cell, and produce a single output digit and two carry output bits.

The digit multiplier cells simply perform multiplication at the digit level. As the two multiplier inputs are each in the range {-1,0,1}, the output of the multiplier cell is the same ({-1,0,1}). In fact, the multiplier can be considered to act on one digit, according to the value of the other, were that action is:

- set the output to zero
- change the sign but not the magnitude
- do nothing

In table form, this is

Output		*Input*0		
		-1	0	1
*Input*1	-1	1	0	-1
	0	0	0	0
	1	-1	0	1

The individual magnitude and sign bits combine to produce the following Karnaugh maps, where the essential choices for the sign bit are circled (all other choices for S are optional groupings).

S_Out		S_0M_0			
		00	01	11	10
S_1M_1	00	0	0	1	1
	01	0	$\textcircled{0}$	$\textcircled{1}$	1
	11	1	$\textcircled{1}$	$\textcircled{0}$	0
	10	1	1	0	0

M_Out		S_0M_0			
		00	01	11	10
S_1M_1	00	0	0	0	0
	01	0	1	1	0
	11	0	1	1	0
	10	0	0	0	0

These Karnaugh maps produce the groupings

$$\begin{aligned} S_{Out} &= S_1.\overline{S_0} + S_0.\overline{S_1} \\ &= S_1 \oplus S_0 \qquad (4.47) \end{aligned}$$

and

$$\begin{aligned} M_{Out} &= M_1.M_0 \qquad (4.48) \\ \overline{M_{Out}} &= \overline{M_1.M_0} \end{aligned}$$

An extract of the full multiplier showing the digit multiply cell is shown in Figure 4.28. The layout shows the 'b' input bus running right to left and the 'a' input being applied directly to the multiplier cell, although in the complete multiplier the 'a' input will run top to bottom on a bus. In this way, the VLSI layout of the multiplier is very similar to the schematic in Figure 4.27, except that the bottom half has been inverted to ease connections.

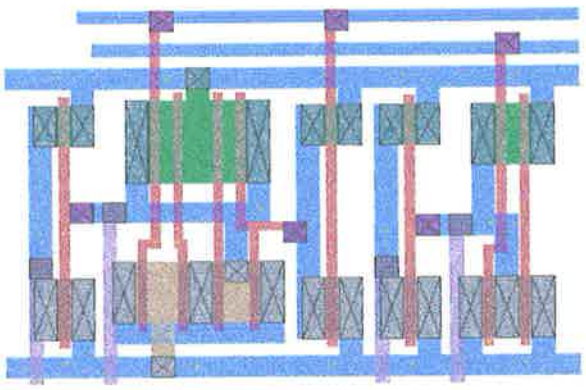


Figure 4.28: **VLSI Layout of Digit Multiply Cell**

The complete layout of the signed digit multiplier using unlatched adder cells is shown in Figure 4.29. A box draw around the multiplier, encompassing the full multiplier top to bottom and left to right, consumes an approximate area of $0.82\text{mm} \times 1.58\text{mm} = 1.30\text{mm}^2$, which includes unused area. The distance between successive cells in the horizontal direction is 0.1184mm. Therefore, extending the multiplier array to an 8×32 array would consume approximately $0.82\text{mm}(1.58 + 0.1184 \times 24) = 3.62\text{mm}^2$ of silicon in 0.7μm es2 CMOS. As a sign digit adder always takes the same time to perform a calculation irrespective of word length, the cycle time remains the same for an 8×32 as for an 8×8 multiplier.

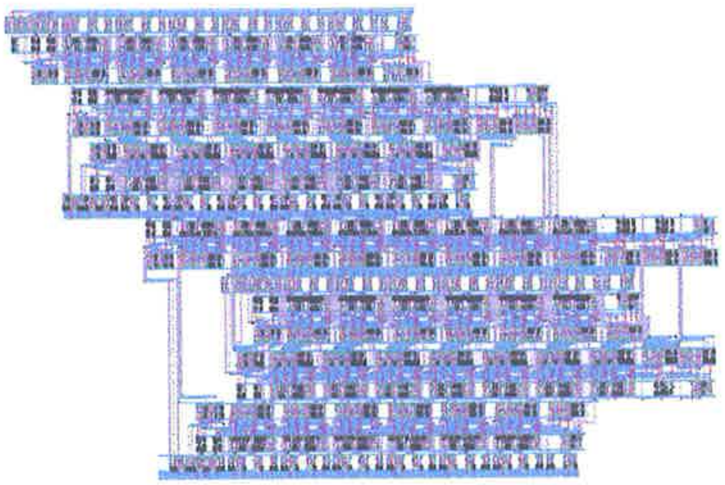
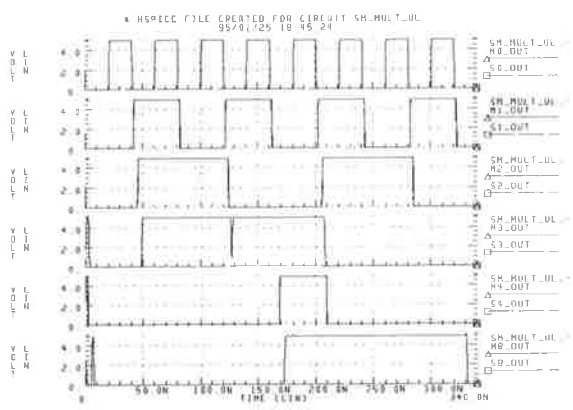


Figure 4.29: **VLSI Layout of** 8×8 **Non-pipelined Multiplier**

The multiplier is too large to test exhaustively, so a few test vectors were applied. For each test, the 'a' input was incremented from zero to eight and then from 248 to 255. The 'b' input was set for a complete simulation. The following HSpice plots show the relevant outputs for b

= 1, 64, 141, 255. All outputs that are not shown have a magnitude bit of zero.



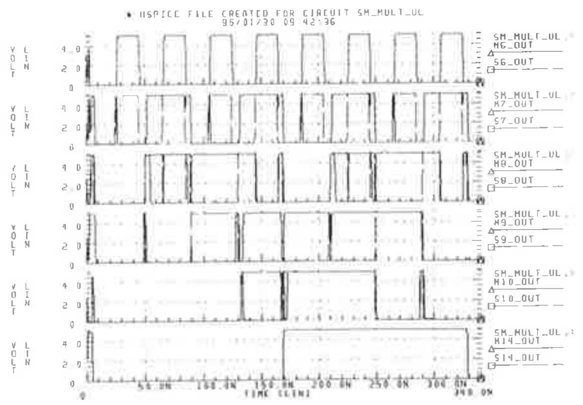
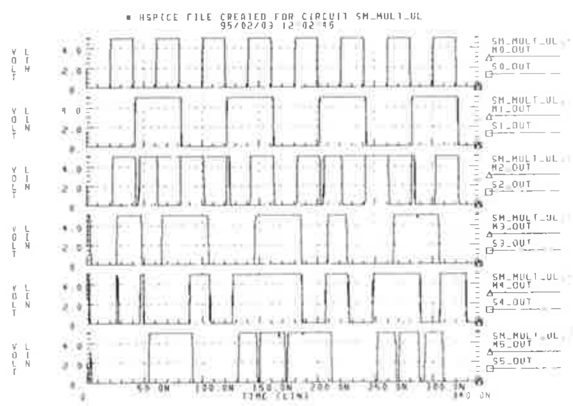


Figure 4.30: **a) b = 1** **b) b = 64**



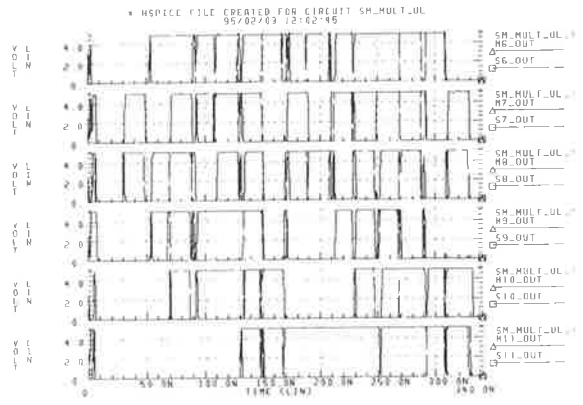
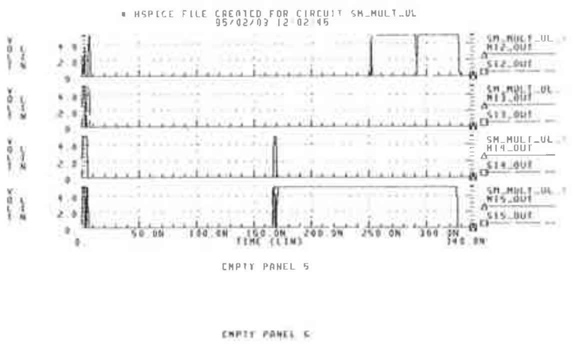


Figure 4.30: **ci) b = 141** **cii) b = 141**

The plots show that the multiply time for an 8×8 three level signed digit multiplier is typically less than ten nanoseconds, with a worst case delay of approximately twelve nanoseconds. Therefore, allowing fifteen nanoseconds for an 8×8 multiply is adequate. The actual output digits are difficult to read in the plots, so they are included as Tables 4.3 to 4.6. The first two tables show the non-zero digit values. The third and fourth tables shows the output in terms of the groups of six digits, in an effort to keep the viewed data to a minimum.

4.3.1 Pipelined Multiplier

Although a pipelined multiplier is not required for the data controllers, the multiplier above was pipelined to demonstrate the ease with which this can be done and the suitability of a signed-digit approach to pipelining. In addition, due to the heat dissipation problems in Gallium



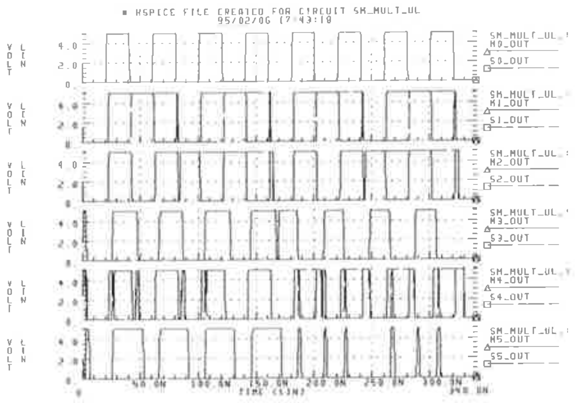
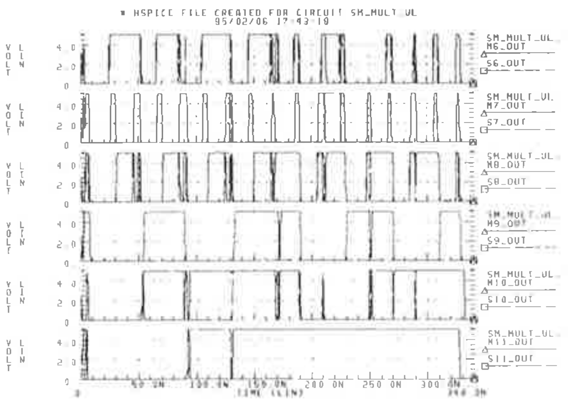


Figure 4.30: **ciii) b = 141** **d) b = 255**



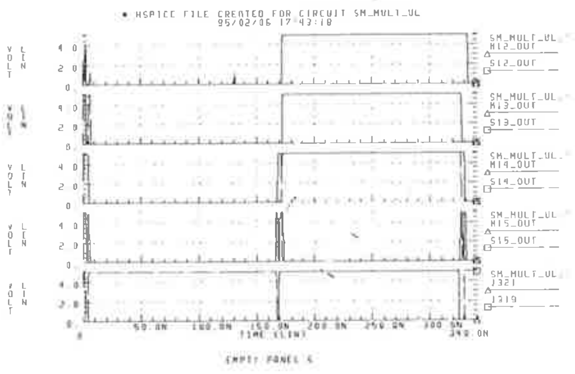


Figure 4.30: **dii) b = 255** **diii) b = 255**

D_0	D_1	D_2	D_3	D_4	D_8	Total
2^0	2^1	2^2	2^3	2^4	2^8	
0	0	0	0	0	0	0
1	0	0	0	0	0	1
0	-1	-1	1	0	0	2
1	-1	-1	1	0	0	3
0	0	-1	1	0	0	4
1	0	-1	1	0	0	5
0	-1	0	1	0	0	6
1	-1	0	1	0	0	7
0	0	0	1	-1	1	248
1	0	0	1	-1	1	249
0	-1	-1	0	0	1	250
1	-1	-1	0	0	1	251
0	0	-1	0	0	1	252
1	0	-1	0	0	1	253
0	-1	0	0	0	1	254
1	-1	0	0	0	1	255
0	0	0	0	0	0	0

Table 4.3: **Digit Outputs for Unlatched Multiplier, b=1**

D_6	D_7	D_8	D_9	$D_1$0	$D_1$4	Total
2^6	2^7	2^8	2^9	2^{10}	2^{14}	
0	0	0	0	0	0	0
1	0	0	0	0	0	64
0	1	0	0	0	0	128
1	-1	1	0	0	0	196
0	0	1	0	0	0	256
1	0	1	0	0	0	320
0	1	-1	1	0	0	384
1	-1	0	1	0	0	448
0	0	0	1	-1	1	14336
1	0	0	1	-1	1	14400
0	1	0	1	-1	1	14464
1	-1	1	1	-1	1	14528
0	0	1	-1	0	1	14592
1	0	1	-1	0	1	14656
0	1	-1	0	0	1	14720
1	-1	0	0	0	1	14784
0	0	0	0	0	0	0

Table 4.4: **Digit Outputs for Unlatched Multiplier, b=64**

$D_0 \rightarrow D_5$	$D_6 \rightarrow D_{11}$	$D_{12} \rightarrow D_{15}$	Total
13	128	0	141
26	256	0	282
39	384	0	423
-12	576	0	564
1	704	0	705
14	832	0	846
27	960	0	987
24	2176	32768	34968
37	2304	32768	35109
-14	2496	32768	35250
-1	2624	32768	35391
12	2752	32768	35532
25	2880	32768	35673
38	-1088	36864	35814
-13	-896	36864	35955

Table 4.5: **Digit Outputs for Unlatched Multiplier, b=141**

$D_0 \rightarrow D_5$	$D_6 \rightarrow D_{11}$	$D_{12} \rightarrow D_{16}$	Total
63	192	0	255
-2	512	0	510
61	704	0	765
-4	1024	0	1020
59	1216	0	1275
-6	1536	0	1530
57	1728	0	1785
8	1792	61440	63240
7	2048	61440	63495
6	2304	61440	63750
5	2560	61440	64005
4	2816	61440	64260
3	3072	61440	64515
2	3328	61440	64770
1	3584	61440	65025

Table 4.6: **Digit Outputs for Unlatched Multiplier, b=255**

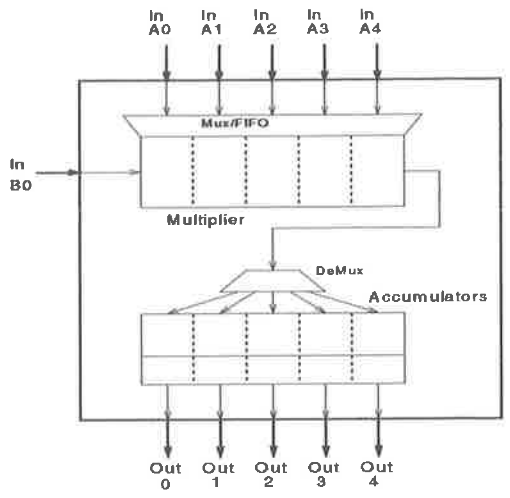


Figure 4.31: **Processing Cell Using Pipelined Multiplier**

Arsenide (GaAs) and the fact that the switching speeds of GaAs are not required[11], the lower power and higher density of CMOS is becoming increasingly attractive, and future generations of the processing array will probably be constructed, at least in part, in silicon CMOS. If that is the case, a single high speed pipelined multiplier can be used to 'simulate' the presence of several processing elements, although each element will contain its own accumulator and I/O port. The likely configuration would be as shown in Figure 4.31.

The signed-digit multiplier is particularly attractive to use as a pipelined multiplier due to the constant delay through each adder layer of the multiplier, irrespectively of the length of the adder. This means that each stage in the pipeline is perfectly matched, and no stage must wait for any other stage. The latches can be placed before any adder stage, for a varying number of pipeline stages. For example, if Booth recoding is used, an 8×8 multiplier comprises of three signed digit adders in two levels, so a two stage pipeline can be implemented. However, if no Booth recoding is used, an 8×8 multiplier will contain three levels of signed digit adders, so a three stage pipeline can be used. Additionally, a two stage pipeline could be used, except that one stage will have two adders and the other will have one adder, resulting in a pipeline imbalance. However, if the multiplier array is (say) 64×64, there will be six levels of signed digit adders, so the pipeline could have two, three or six matched pipeline stages. For the sake of demonstration, the 8×8 multiplier was pipelined into three stages, one for each level of adders.

The 8×8 multiplier VLSI design is shown in Figure 4.32. The design uses 0.7μm es2 CMOS, and consumes a rectangular area of $0.93\text{mm} \times 3.2\text{mm} = 2.976\text{mm}^2$, although this includes some unused space. The difference between successive cells horizontally is 0.25mm, so that extending the multiplier to an 8×32 array be approximately $3.2\text{mm} + 24 \times 0.25\text{mm} = 9.2\text{mm}$ in length, and consume $9.2\text{mm} \times 0.93\text{mm} = 8.56\text{mm}^2$ of area.

The multiplier was simultated using HSpice, although again not exhaustively. The input for the first simulation if shown in Figure 4.33a), which produced the results in Figure 4.33b).

The applied input was the number sequence '0 1 0 3 2 5 4 7 6 9 8 11 10 13 12 15 14 241 240 243 242 245 244 247 246 249 248 251 250 253 252 255 254' for the 'b' input, which was applied for an 'a' input of '0 & 1'. As can be seen in the output plot (Figure 4.33b)), the result of the simulation is an output of zero for the first half of the simulation, followed by the same sequence

[11]The processing elements themselves need not be very fast. In fact, slower P.E.'s have several advantages

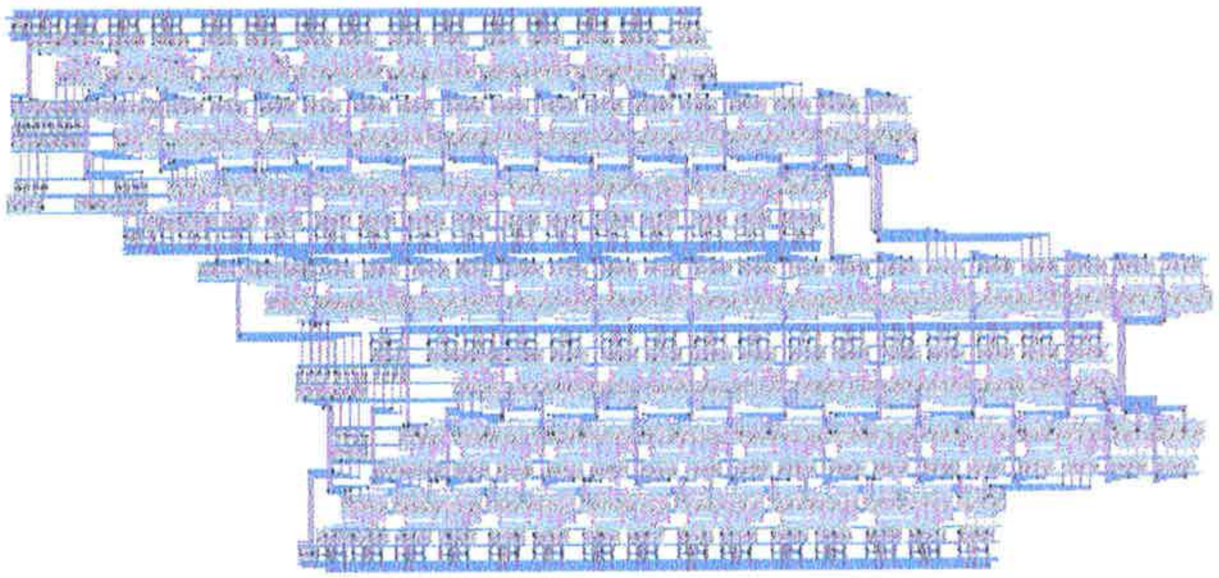


Figure 4.32: **VLSI Layout of Pipelined** 8×8 **Multiplier**

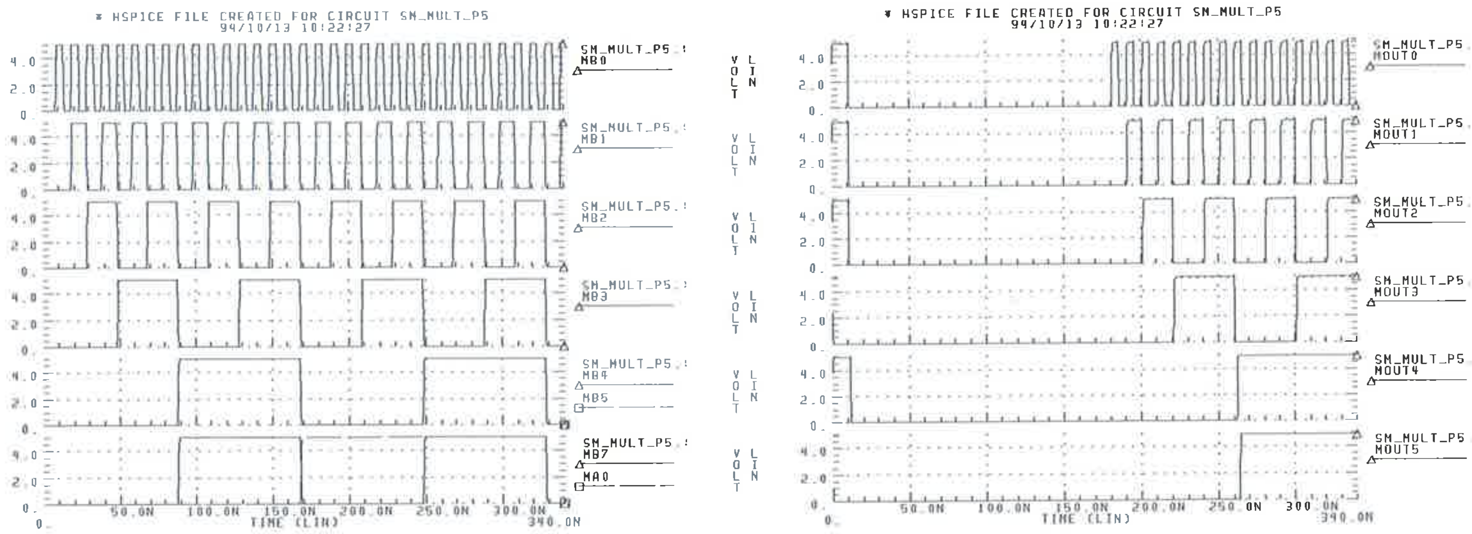


Figure 4.33: **HSpice Simulation Plots** a) **Input** b) **Output**

as the input. A new output is available every five nanoseconds, after an initial delay of fifteen nanoseconds (three cycles). Other simulations were run, and the multiplier found to operate correctly.

4.4 Conclusion

In this chapter, propagation-free signed digit arithmetic was introduced and the conversion process between signed digit and standard binary notations shown. Basic addition cells were designed, laid out and simulated in the 0.7μm es2 CMOS process, and found to operate correctly with addition cycle times of less than five nanoseconds. When the word-length independence of the cycle time of signed-digit arithmetic is recalled, the cells can be used to implement very high bandwidth computational systems.

Support arithmetic components were also considered, and a high-radix divider proposed that provided very favourable speed/area trade-offs, while still providing the performance required by the system. To complete the design of the divider, a signed-digit multiplier architecture was designed using the cells that were designed earlier in the chapter.

Chapter 5

Address Generator

As the computation rates of computers have increased, the ability of a processor to generate operand addresses at a correspondingly high rate has been severely tested. Specialised hardware has been described in the literature by Marwood and others [69, 72] that directly generate fundamental addressing patterns to help alleviate this bottleneck.

The majority of these papers, with the exception of those by Marwood, attempt to optimise the address generation for vector-type operations, leaving matrix operations as a superset of vector operations. Unfortunately, there are several matrix algorithms that do not map well onto vector addressing patterns, such as prime factor mappings, etc. Marwood, however, considers matrix algorithms separately, and has implemented designs that directly support many matrix structures. Marwood's address generation difference engine is used here, due to its versatility of function and elegance of design.

In fact, the address generator difference engine is the fastest component of the whole system, and the speed of the generator is the main limit on system performance. The importance of memory bandwidth for a systolic array was pointed out by Katona in his lattice model for cellular algorithms [46, 58], which can be translated into operand address generation bandwidth if the addresses are generated with a slower cycle time than the memory system. Marwood applied the results to the evaluation of matrix products on a two dimensional systolic array [58], a brief summary of which follows.

The lattice model involves the concept of two matrices, **A** & **B**, moving toward the processing array with velocities v_a & v_b respectively, to form the matrix product **C** within the processing array. This is shown graphically in Figure 5.1 from [58]. For ease, the processing array is set to be a square $N \times N$ array, whereas Marwood uses the more general rectangular $M \times N$ array. He reaches the conclusion that the bandwidth requirement of the two input arrays (β_{AB}) is given by

$$\beta_{AB} = \beta_A + \beta_B = \frac{2N}{\tau} \tag{5.1}$$

where β_A and β_B are the bandwidth requirements of each input, and that the output bandwidth (β_C) can be matched to those of the input by setting

$$\beta_C = \frac{2N}{\tau} \tag{5.2}$$

where τ is a time step.

By dividing the number of processor operations by the time in which they occur, the computational rate can be determined. Bearing in mind that a multiply and accumulate takes place in a single time step and that there are N^2 processors, the computational rate (R) is

$$R = \frac{2N^2}{\tau} \tag{5.3}$$

Substituting Equation 5.1 into 5.3 leads to

$$\begin{aligned} R &= \frac{2\left(\frac{\beta_{AB}\tau}{2}\right)^2}{\tau} \\ &= \frac{\beta^2_{AB}\tau}{2} \\ &= \beta_{AB}N \end{aligned} \quad (5.4)$$

This is a slightly surprising result, as it indicates that not only is the computational power of the square processing array proportional to the square of the array bandwidth, but also that it is *inversely* proportional to the computational speed of the processing elements. The logical explanation is that the computational rate, R, in Equation 5.4, is determined for the case when the matrix processor is being supplied with data at its maximum rate. The bandwidth required for an $N \times N$ processing array is the same as for a $2N \times 2N$ array with each element operating at half the speed. As there are four times the number of systolic elements in the larger array, each operating at half the speed of the smaller array, the larger array has twice the computational rate using the same bandwidth.

As Marwood points out, assuming that the problem order is greater than the system hardware order, there are two possible alternatives for increasing system performance:

1. Increase the bandwidth to the array. This is technologically constrained by the limits on how fast data can be moved.

2. Increase the execution time of the processing elements to allow a greater number of processing elements to operate with the same overall bandwidth. This is not constrained by technology, as the processing elements can be made arbitrarily slow. Slower PEs often mean smaller PEs, since serialization of the arithmetic operation can be used to reduce PE size at the cost of longer execution time. The limit here is due to practical considerations on the physical size of the array, and the constraints of the problem size (the size of processing array needs to smaller than the order of the problem size to increase performance by increasing array size. Ideally, the problem size should be an integral multiple of the array size).

This is also of significance if Marwood's *constant bandwidth* model is used [58]. Here, rather than stalling if the entire array is not being used, the array is composed of variable speed components along bands. The computational speed is selected depending on array utilisation. A range of technologies can be used, with the faster 'edge' components using more expensive technology (eg. Gallium Arsenide) and the components getting progressively slower further from the array. Thus, although an $N \times N$ array will be N times faster than a single computational element running at the same speed, if a scalar calculation is required, the single computational element will be N times *faster* than the array.

5.0.1 Matrix Addressing

The original system proposed by Marwood [56] describes the mapping algorithm for an arbitrary dimensioned mapping, although the implementation shown is for a two dimensional mapping. This is shown in Figure 5.2, below. This work was extended [65] to a include up to four dimensions, which will allow either a four dimensional structure to be mapped on to a a one dimensional addressing space, or for a two dimensional, arbitrary-sized structure to be automatically partitioned to fit a fixed size processing array.

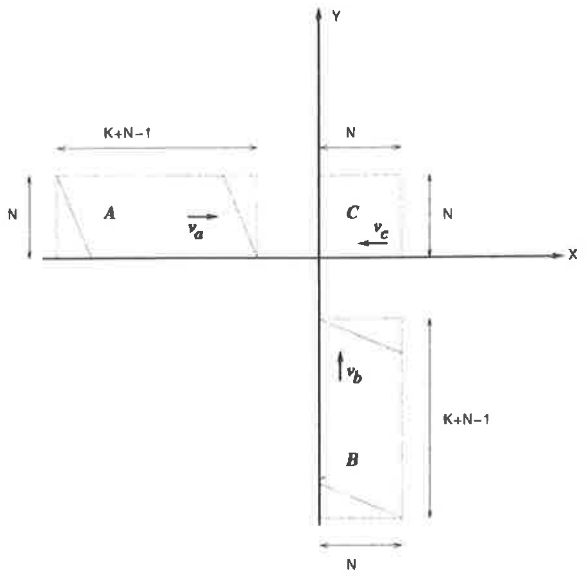


Figure 5.1: **Lattice Representation of a Matrix Product**

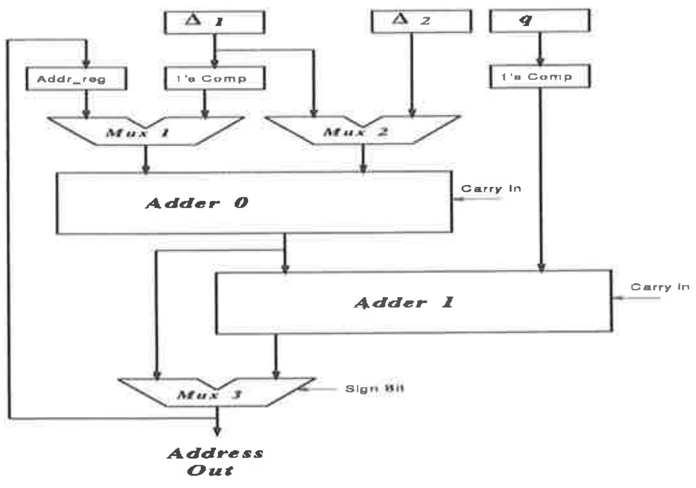


Figure 5.2: **2D Addr Gen.**

Marwood's example mapping is the Alternative Integer Mapping, although he points out that the Chinese Remainder Theorem and other simple mappings can be implemented on the proposed architecture.
The Alternative Integer Representation is

$$n = \langle n_1 N_2 + n_2 N_1 \rangle_N \tag{5.5}$$

where N_1 and N_2 are mutually prime and $N_1 N_2 = N$, and $\langle a \rangle_N$ means 'a *modulo* N'.

This is very similar to the conventional mapping used to store a two or more dimensional matrix in a linear memory address space, ie

$$n = n_1 N_2 + n_2 \tag{5.6}$$

The Chinese remainder theorem is slightly more complicated, although still implementable on the proposed hardware.

$$n = \left\langle n_1 N_2 \left\langle N_2^{-1} \right\rangle_{N_1} + n_2 N_1 \left\langle N_1^{-1} \right\rangle_{N_2} \right\rangle_N \tag{5.7}$$

If appropriate constants are chosen and a modulo capability is available, Equations 5.5 to 5.7 can be implemented on a two dimensional difference engine. Additionally, an offset base address can be included if added after the difference engine. This leads to the following expression:

$$n = \mathit{base\_address} + \langle n_1 \Delta_1 + n_2 \Delta_2 \rangle_q \tag{5.8}$$

It would be desirable to remove the need for the divider that is generally used to perform the modulo calculation. This can be done by noting that at each address calculation, a difference, Δ_i, is added to the previous address, '*prev_addr*', which is already 'modulo q'. Therefore, if the difference between 'q' and the previous address is greater than Δ_i, the sum of the previous address and Δ_i is correct modulo q. Otherwise, it is required that q be subtracted from the result of the sum. This can be summarised as:

$$\mathit{next\_addr} = \begin{cases} \mathit{prev\_addr} + \Delta_i & \text{if } q - \mathit{prev\_addr} - \Delta_i > 0 \\ \mathit{prev\_addr} + \Delta_i - q & \text{otherwise} \end{cases} \tag{5.9}$$

This is implemented on Marwood's difference engine serially, with the initial calculation being $\mathit{prev\_addr} + \Delta_i$ followed by $(\mathit{prev\_addr} + \Delta_i) - q$. The sign of the second calculation is then used as the control bit for the multiplexers.

5.0.2 Parallelizing and Expanding the Difference Engine

As Equation 5.9 is a loop required for each iteration containing the previous address, it would be difficult to pipeline this stage. However, parallelizing is a relatively simple matter. Rather than calculating '$\mathit{prev\_addr} + \Delta_i$', and then subtracting the modulo 'q' from this, if $\Delta_i - q$ is also supplied to the address generator, the sum of '$\mathit{prev\_addr} + (\Delta_i - q)$' can be calculated in parallel with the original sum (of $\mathit{prev\_addr} + \Delta_i$). The required result can just be switched using a multiplexer, depending on the sign of the result of the sum $\mathit{prev\_addr} + (\Delta_i - q)$. Figure 5.3 shows a parallel implementation of the difference engine. As is obvious from this figure, the parallel version of the difference engine requires an extra (*dimensions* - 1) registers over the serial version, but no extra adders.

Expansion of the number of dimensions from two to four is also a simple matter. This can be achieved by adding four additional registers and four additional multiplexers.

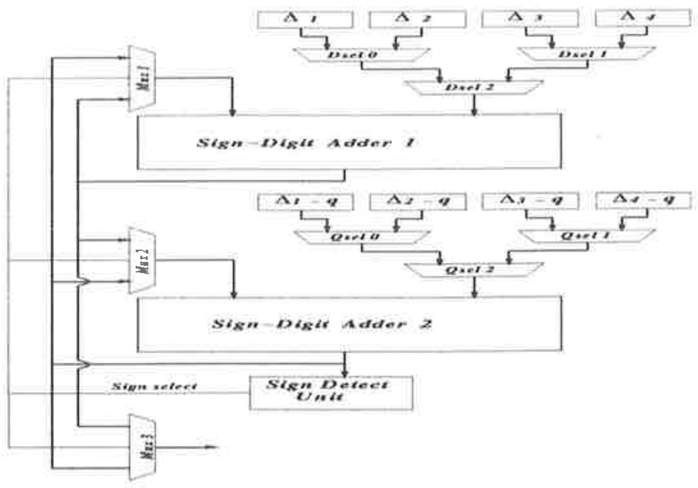


Figure 5.3: **4D Addr Gen.**

5.0.3 Example Mappings on the Difference Engine

Marwood provides several example address mappings implemented on his address generator, which can also be implemented on the expanded, four dimensional address generator[56, 58]. The required input data for a selected mapping and the corresponding output sequence of addresses is presented in Table 5.1.

These same mappings can be calculated on the expanded address generator, as presented in Table 5.2. Note that these do not use the third and fourth dimensions of the array, as only the equivalent mappings of the two dimensional version are presented.

5.1 Address Generator Components

5.1.1 Sign Detector

One of the main disadvantages of signed digit arithmetic is that the sign of an integer is not readily apparent, as it is with two's complement arithmetic. The sign of a signed digit number is the sign of the most significant non-zero digit, which may be at any of the 32 positions for a 32-bit number.

If each bit were checked in turn, an order $O(n)$ operation, the advantage of the inherent parallelism of signed digit arithmetic would be lost. An order $O(\log n)$ sign detect unit can be constructed using a binary tree, with each cell transmitting the sign of the larger of two adjacent digits to the layer beneath. The cells themselves consist of three multiplexers, which select whether the more or less significant digit will be propagated to the next layer. The select signal for the multiplexers is the magnitude bit of the more significant digit. A block diagram is shown in Figure 5.4.

The equations governing the operation of the cell are:

$$S_{out} = M_1.S_1 + \overline{M_1}.S_0 \tag{5.10}$$
$$M_{out} = M_1 + \overline{M_1}.M_0 \tag{5.11}$$
$$\overline{M_{out}} = \overline{M_1}.\overline{M_0} \tag{5.12}$$

The least significant edge cell can be reduced in size, as the outputs M_1 & M_0 are not required. The full sign detect unit is shown in Figure 5.5.

Type	Normal	Transposed	Prime Factor	Transposed Prime Factor	Circulant	Circulant Skew	Sub Matrix	Constant
Base	0	0	0	0	0	0	6	0
$\Delta 1$	1	5	3	5	1	1	1	0
$\Delta 2$	1	-9	8	8	0	2	4	0
n1	5	3	5	3	5	5	2	5
n2	3	5	3	5	3	3	2	3
q	15	15	15	15	5	5	15	1
Addr. Out	0	0	0	0	0	0	6	0
	1	5	3	5	1	1	7	0
	2	10	6	10	2	2	11	0
	3	1	9	3	3	3	12	0
	4	6	12	8	4	4		0
	5	11	5	13	4	1		0
	6	2	8	6	0	2		0
	7	7	11	11	1	3		0
	8	12	14	1	2	4		0
	9	3	2	9	3	0		0
	10	8	10	14	3	2		0
	11	13	13	4	4	3		0
	12	4	1	12	0	4		0
	13	9	4	2	1	0		0
	14	14	7	7	2	1		0

Table 5.1: **2 Dimensional Mappings on Difference Engine**

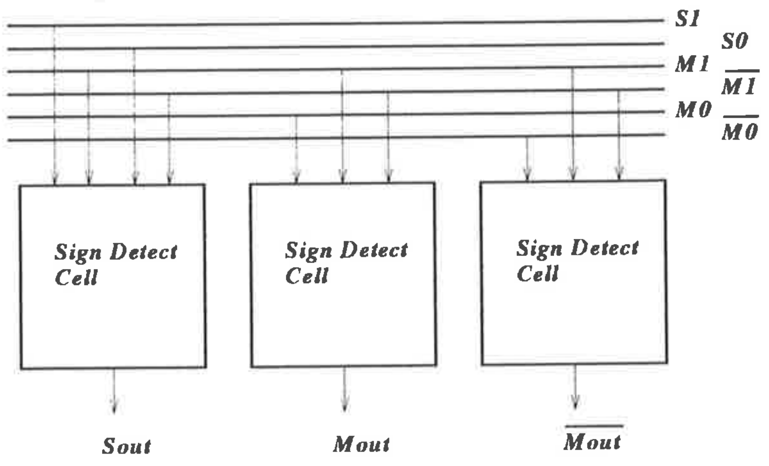


Figure 5.4: **Sign Detect Unit Cell**

Type	Normal	Transposed	Prime Factor	Transposed Prime Factor	Circulant	Circulant Skew	Sub Matrix	Constant
Base	0	0	0	0	0	0	6	0
$\Delta 1$	1	5	3	5	1	1	1	0
$\Delta 2$	1	-9	8	8	0	2	4	0
$\Delta 3$	0	0	0	0	0	0	0	0
$\Delta 4$	0	0	0	0	0	0	0	0
$\Delta 1 - Q$	-14	-10	-12	-10	-4	-4	-14	-1
$\Delta 2 - Q$	-14	-24	-7	-7	-5	-3	-11	-1
$\Delta 3 - Q$	-15	-15	-15	-15	-5	-5	-15	-1
$\Delta 4 - Q$	-15	-15	-15	-15	-5	-5	-15	-1
n1	5	3	5	3	5	5	2	5
n2	3	5	3	5	3	3	2	3
n3	0	0	0	0	0	0	0	0
n4	0	0	0	0	0	0	0	0
Addr.	0	0	0	0	0	0	6	0
Out	1	5	3	5	1	1	7	0
	2	10	6	10	2	2	11	0
	3	1	9	3	3	3	12	0
	4	6	12	8	4	4		0
	5	11	5	13	4	1		0
	6	2	8	6	0	2		0
	7	7	11	11	1	3		0
	8	12	14	1	2	4		0
	9	3	2	9	3	0		0
	10	8	10	14	3	2		0
	11	13	13	4	4	3		0
	12	4	1	12	0	4		0
	13	9	4	2	1	0		0
	14	14	7	7	2	1		0

Table 5.2: **4 Dimensional Mappings on Difference Engine**

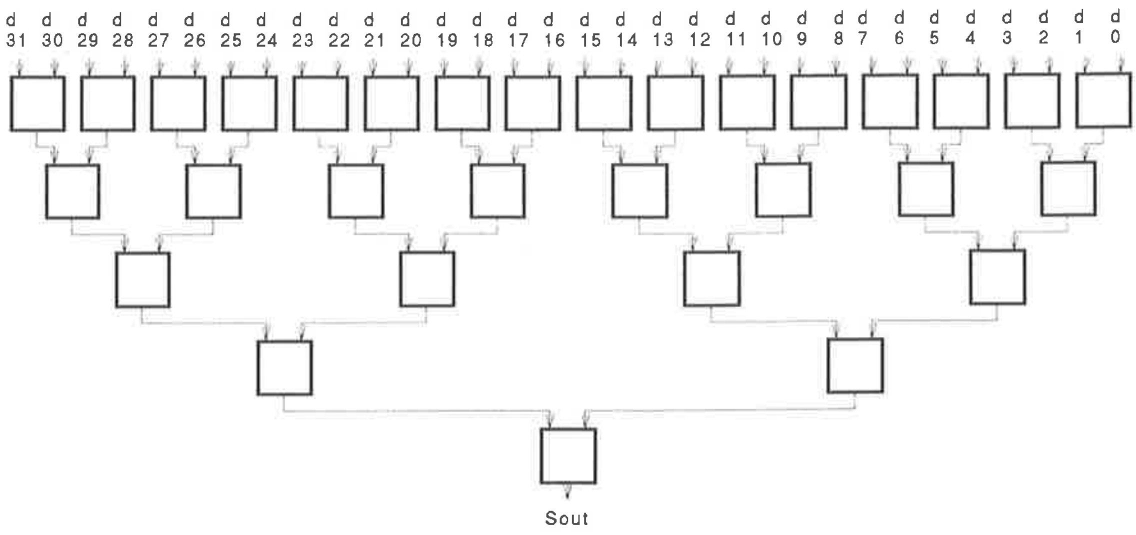


Figure 5.5: **Sign Detect Unit**

5.1.2 Initialisation

The difference engine must always start with the first output address being zero. Any offset in this address is included by means of the base address register and adder. In Figure 5.2, initialisation is achieved by adding the value in register D1 to the negative of itself, with the result placed in the '*Address Out*' register. As the result of a number added to the negative of itself is always zero, the *Address Out* register is effectively initialized to zero. While this achieves the desired result, there will be a minor speed penalty due to the extra multiplexer in the critical path (Mux1 in Figure 5.2). As the use of redundant coded binary is suggested for the adders, the delay through a multiplexer may account for approximately ten percent of the cycle time.

An alternative initialisation option is available depending on the technology used. In the suggested CMOS design, domino logic was used in the final stage of the calculation of the 'm' output of the adders. The domino logic multiplexer used is shown in Figure 5.6, below.

As can be seen seen from the figure, the input to the inverter is held high until the signal 'phi' is brought high. 'Phi' is an evaluate signal that is clocked every cycle during operation. However, 'phi' can also be combined with a 'run' signal, so that 'phi' follows a clock during normal operation, or else is held low for initialisation or if a stall occurs. The combination is a simple AND gate, ie

$$phi = run.clk^I$$

where clk^I is a clock signal. Therefore, if the *Address Out* register is latched while 'phi' is low, ie not in evaluate phase, the data at the input of the *Address Out* register will be zero. Therefore, all that is required for initialisation is that the *Address Out* register(s) receives the latch signal before the system starts running.

5.1.3 Dimension Counter

Within the address generator framework, there is a need for as many counters as there are dimensions. At least one of these counters must be very fast (the one of lowest dimension), as

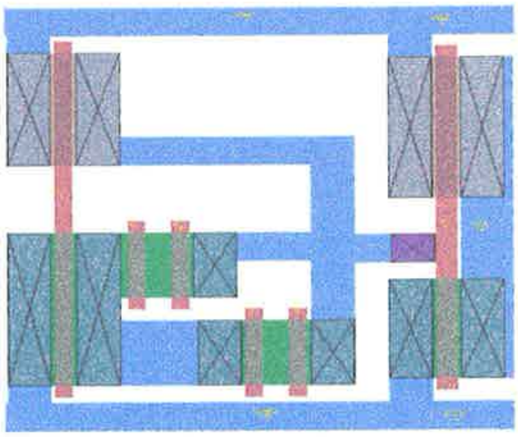


Figure 5.6: **Domino Logic Exclusive-OR Gate**

there will only be a single cycle to calculate which 'delta' to use and switch it in. Therefore, the speed of signed digit arithmetic is again valuable.

Decrement Cell

Rather than implementing a full adder/latch cell, use can be made of the fact that one input to the decrementer is already know to be zero in all but the least significant cell. In fact, if the negative carry input is used to apply the '-1' to the decrementer, all the cells can be made identical, with only one input and the appropriate carry lines.

Let the inputs to the cell be labelled $\{S_{in}, M_{in}\}$, and the carry input be *Neg_In*[1]. The requirements of signed digit arithmetic are such that the *Neg_Out* signal must be independent of the *Neg_In* input signal. This is the only restriction on the cell logic. The resulting cell behaviour is described in Table 5.3. The figure 'X' denotes a 'don't care' in the table, and 'Δ' indicates a 'pseudo don't care', in which the table entry in which it is placed can be either a '0' or a '1', but the choice will effect other table entries.

S_{in}	M_{in}	Neg_{in}	Result	S_{out}	M_{out}	Neg_{out}
0	0	0	0	X	0	0
0	0	1	-1	Δ	1	$\overline{\Delta}$
0	1	0	1	0	1	0
0	1	1	0	X	0	0
1	0	0	0	X	0	0
1	0	1	-1	Δ	1	$\overline{\Delta}$
1	1	0	-1	Δ	1	$\overline{\Delta}$
1	1	1	-2	X	0	1

Table 5.3: **Decrement Cell Combinations**

[1]As will be shown later in the analysis, a positive carry signal is not required

The position {1,1,1} (corresponding to $\{S_{in}, M_{in}, Neg\_In\}$) requires that Neg_Out be set for this combination of inputs. As Neg_Out can only depend on $\{S_{in}, M_{in}\}$, then position {1,1,0} must also set Neg_Out. This is the only combination of S_{in} and M_{in} for which Neg_Out is set. The resulting Karnaugh maps for the decrement cell are given in Tables 5.4 a) & b).

S_{out}, M_{out}		$S_{in}M_{in}$			
		00	01	11	10
Neg_In	0	1 0	0 1	0 1	1 0
	1	1 1	0 0	0 0	1 1

Neg_Out		M_{in}	
		0	1
S_{in}	0	0	0
	1	0	1

Table 5.4: **a)** S_{out} & M_{out} **b)**Neg_Out

From these Karnaugh maps, the obvious groupings are:

- $S_{out} = \overline{M_{in}}$
- $M_{out} = Neg\_In \oplus M_{in}$
- $Neg\_Out = S_{in}.M_{in}$

A further simplification can be made if it is noted that the *exclusive-nor* operator is just as easy to implement as the *exclusive-or*, and that, as an *and* operator is formed using a *nand* gate and an inverter, the *nand* operator is simpler than the *and* operator. Therefore, if $\overline{Neg\_Out}$ is propagated between cells instead of Neg_Out, then the gate count and propagation delay can be reduced by one. The new equations are:

- $S_{out} = \overline{M_{in}}$
- $M_{out} = \overline{\overline{Neg\_In} \oplus M_{in}}$
- $\overline{Neg\_Out} = \overline{S_{in}.M_{in}}$

The flow diagram representing the decrementer is shown in Figure 5.7.

The decrement cell is shown in Figure 5.8. This cell includes the iteration loop latches, the input multiplexers and the arithmetic logic.

To enable enough time to achieve the zero detection and the appropriate switching of the 'delta' multiplexers, the value stored in the 'zeroth' counter can be decremented before it is stored (run once through the decrementer). This requires an extra pair of multiplexers and an extra pair of latches. The extra latches are required to store the 'new' decremented starting value, while the multiplexers select whether the original value or the decremented value is loaded into the new lathes. These multiplexers are only used at initilisation. The only other consideration for constructing a full digit cell is the output. Rather than routing wires from the iteration loop latches, it was decided that another pair of latches at the output would consume minimal area and save routing. Therefore, these latches have been included in the full digit cell, although they could just as easily be left out. The layout of the full digit cell is shown in Figure 5.9, with the decrement cell outlined.

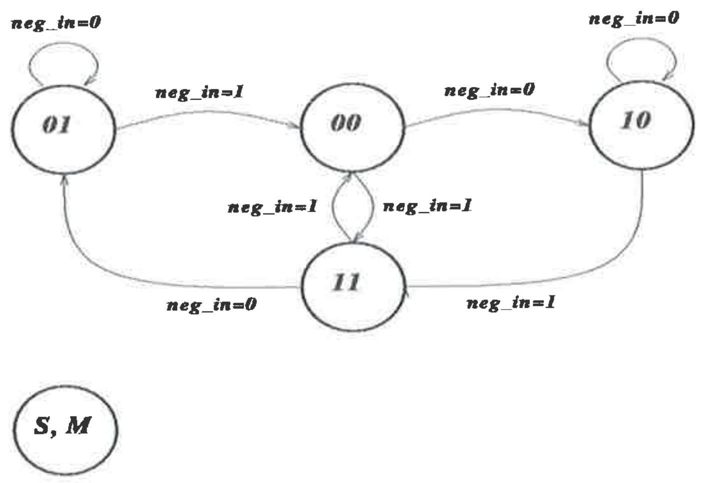


Figure 5.7: **Flow Diagram for Decrement Cell**

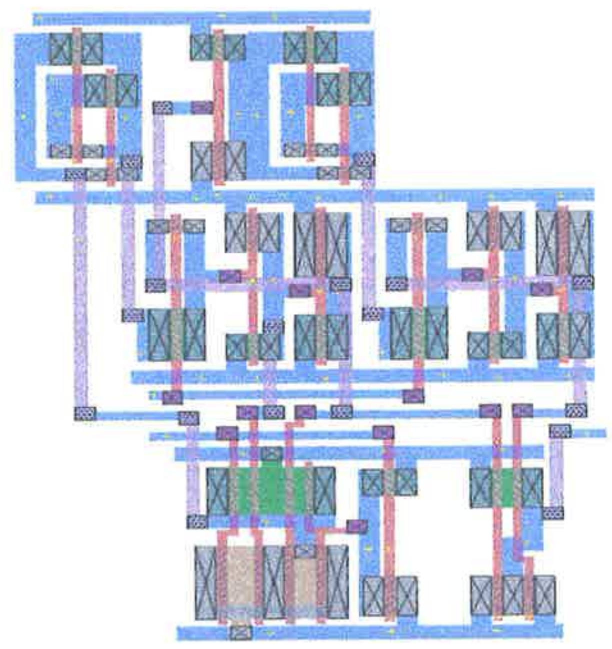


Figure 5.8: **Decrement Cell**

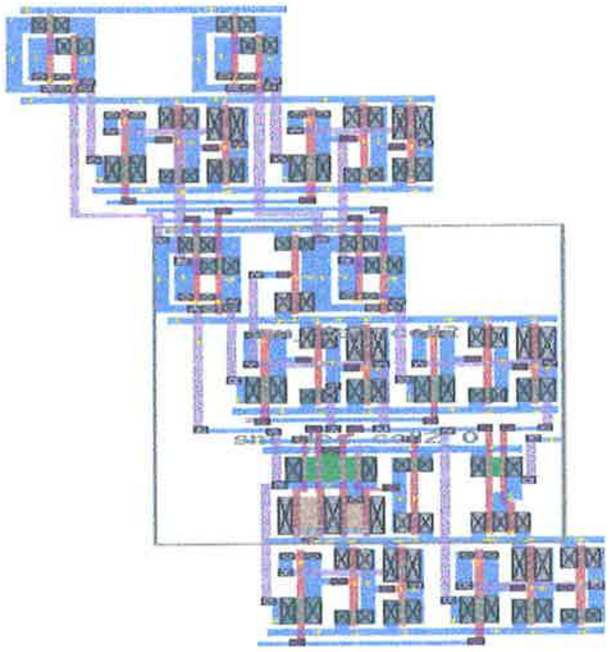


Figure 5.9: **Full Decrement Cell**

(RE)Loading the Decrement Counters

The decrementers must be loaded initially with the correct dimension, and then reloaded with the original dimension minus one every time they reach zero, until all decrementers have counted down to zero. To control this, a reset signal is used. The only input control signal to the decrementer is the '*load*' signal, with the sequencing controlled internally and also using the '$\overline{\textit{latch}}$' signal. When the '*zero*' signal is asserted, a '*reset*' signal is asserted, and the starting value (original value minus one) reloaded into the counters.

Zero Detect Cell

One of the requirements of the redundant set proposed by Avizienis [3] was that there is a unique representation for the number zero. Although the signed magnitude notation described here does not have a strictly unique representation for zero, there is a unique combination for the magnitude bits (they are all zero), and it is only the sign bits that change for various zero representations. Thus, to determine if a signed digit number is zero it is necessary to determine if each and every magnitude bit is zero[2]

The approach taken to detect if the decrementer value was zero was a simple NAND-NOR tree structure, which is a logical equivalent to an OR-OR structure for zero detection. For a 32-bit zero detection circuit, $\log_2 32 = 5$ levels of binary zero detection are needed, which corresponds to three NOR-gate levels and two NAND-gate levels. Therefore, a zero at the input will be indicated by a one at the output.

The VLSI structure of the zero detection cell is shown in Figure 5.10. This is a four input cell, constructed of two NOR gates and one NAND gate. The pyramidical structure is clearly shown, although in practice, a large zero detection cell can be compressed into a single linear array of NAND and NOR gates. This is shown later for the case of a sixteen bit decrementer.

[2]This is the same as for a standard binary representation.

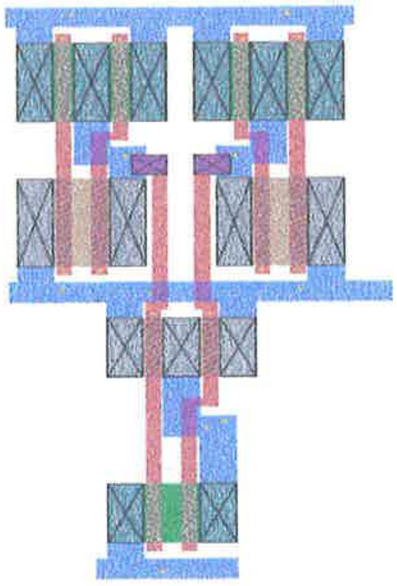


Figure 5.10: **Four Input Zero Detection Cell**

5.1.4 Complete Decrement Units

To test the design of the decrement cells, a four bit and a sixteen bit decrementer were designed and simulated. The four bit decrementer cell can count down from a maximum of fifteen and the sixteen bit decrementer can count down from 65535. The VLSI layout of the four bit decrementer is shown in Figure 5.11. Without compaction, the four bit decrementer consumes an area of $0.277\text{mm} \times 0.246\text{mm} = 0.068\text{mm}^2$.

The testing of the decrementer is very simple, as it only needs to be tested for the case when the largest possible input is applied, as the decrementer will pass through all other states. The output from HSpice is shown in Figure 5.12. It can be seen from this that the decrementer initially decrements the input from fifteen to fourteen, and then proceeds to decrement the output every clock pulse until the output is zero, at a clock rate of 200MHz. When the zero flag is asserted, the decrementer is reloaded with the stored value, in this case fourteen. However, the decrementer has another cycle before the stored value is loaded (the cycle in which zero is asserted), so the output actually goes to minus one. The next output is the stored value minus one, in this case thirteen, and then the cycle repeats.

The sixteen bit decrementer is shown in Figure 5.13. The four levels of zero detection are compressed into a single layer, on the bottom of the cell. The complete cell consumes $0.245\text{mm} \times 0.886\text{mm} = 0.217\text{mm}^2$ in $0.7\mu\text{m}$ es2 CMOS. Most of the area is consumed with initialisation and reset hardware, with the actual decrement cell area consuming approximately one third of the total.

Full testing of the sixteen bit decrementer is not possible, as a decrement from 65535 at five nanoseconds per count would require a simulation time of $65535 \times 5 \times 10^{-9} = 327.68\mu\text{S}$, which, apart from the huge amount of time required for such a simulation, would consume more storage than we have available. However, a decrement from fifteen was run, which shows the reset and reload functions, and also the zero detection. As signed digit arithmetic is word length independent, adding more significant digits will not effect the speed at which the decrementer

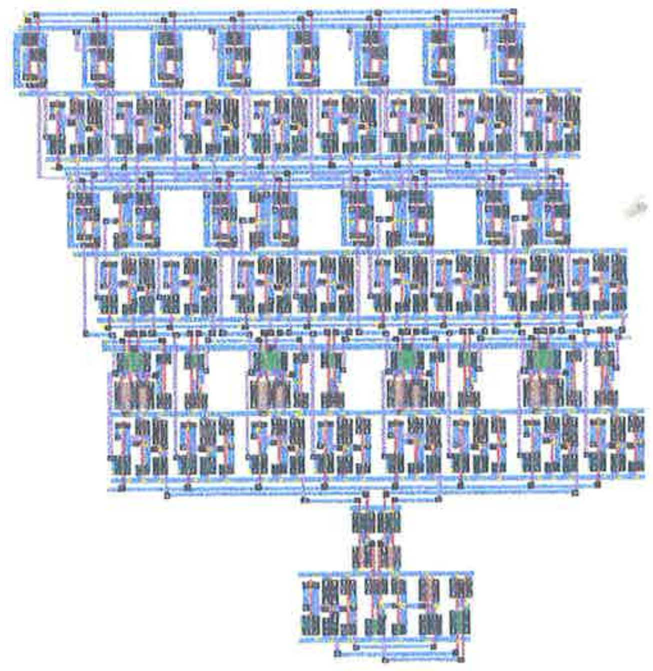


Figure 5.11: **Four-Bit Decrement Unit**

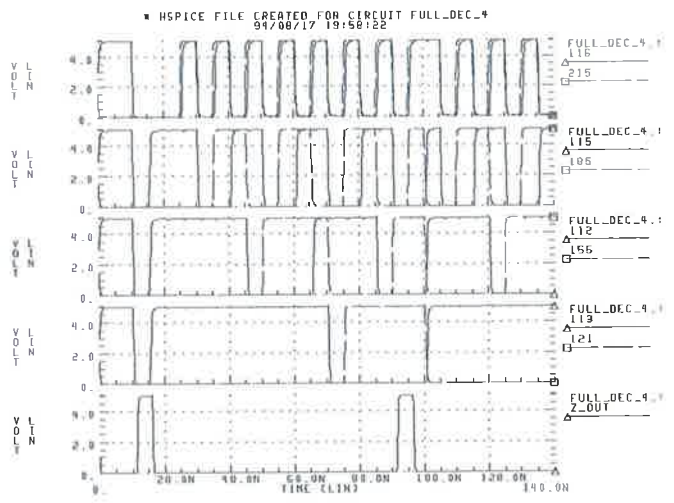


Figure 5.12: **Simulation Results of Four Bit Decrement Cell**

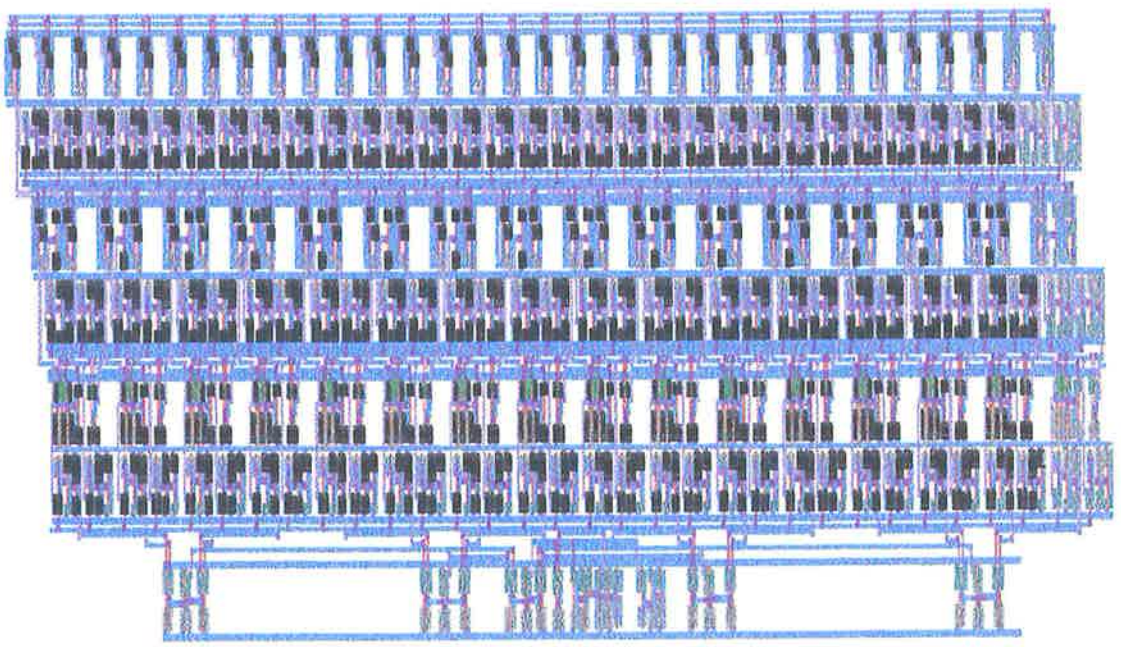


Figure 5.13: **Sixteen-Bit Decrement Unit**

operates. These results are shown in Figure 5.14. All output magnitudes not shown are zero. As can be seen from this figure, the decrementer function correctly.

5.1.5 Multiplexer Selection

The multiplexers that are used to choose between the 'deltas' must be selected in a hierarchical manner. The delta that is chosen is the one with the smallest index for which the corresponding counter output is not zero. Therefore, if 'SelA' and 'SelB' are the two multiplexer controls as defined in the Address Generator Section (Section 5.2), *zero0 ... zero3* are the outputs of the four zero detectors, and *mux0*, *mux1*, *mux2* are temporary variables denoting

- select *delta*0/1
- select *delta*2/3
- select *delta*(0,1)/(1,2)

respectively, then the multiplexer selection equations become

- *mux2* = zero0 . zero1
- *mux1* = zero2
- *mux0* = zero0

The schematic of the multiplexers showing both sets of select signals (*mux0 ... 3* and SelA & SelB) is given in Figure 5.15.

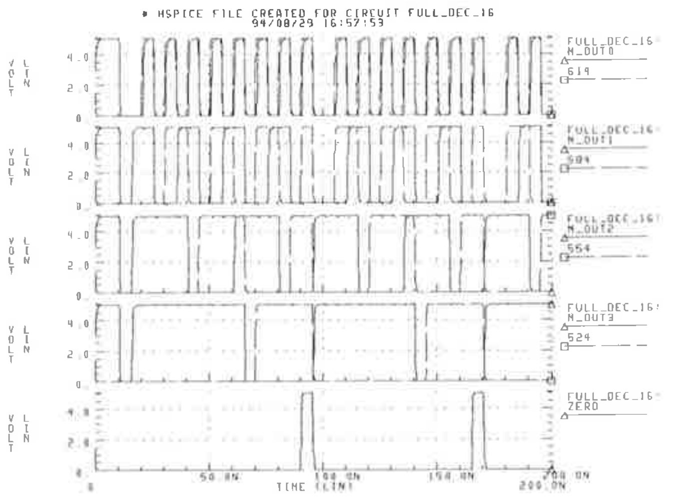


Figure 5.14: **HSpice Simulation Results for Sixteen Bit Decrementer**

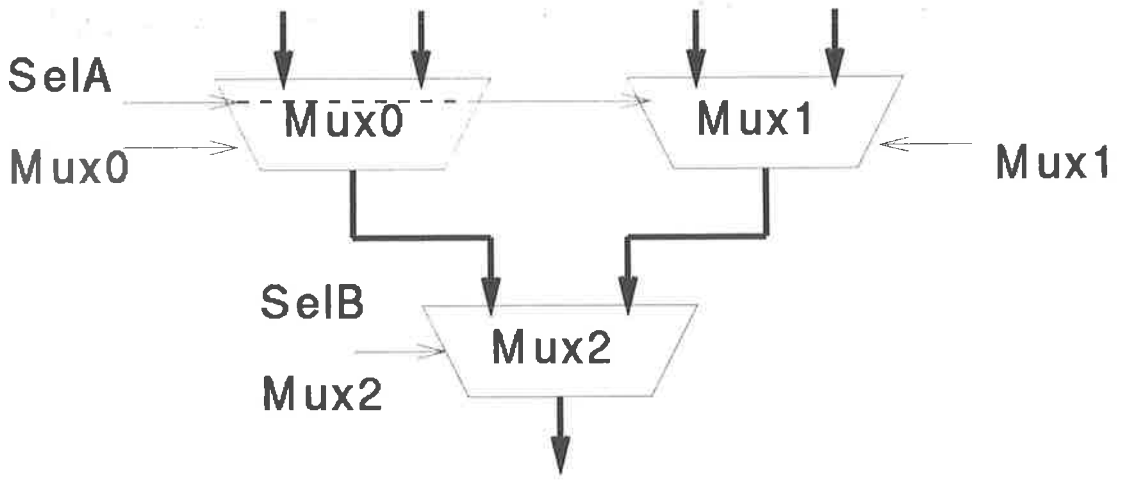


Figure 5.15: **Delta Register Multiplexers**

To convert the separate multiplexer control signals to the previously defined SelA and SelB, it is obvious that SelB is simply equivalent to *mux2*. SelA is a combination of all the mux variables, and can be written as

$$\begin{aligned} SelA &= \overline{mux2}.zero0 + mux2.zero2 \\ &= \overline{zero0.zero1}.zero0 + zero0.zero1.zero2 \\ &= (\overline{zero0} + \overline{zero1}).zero0 + zero0.zero1.zero2 \\ &= \overline{zero0}.zero0 + \overline{zero1}.zero0 + zero0.zero1.zero2 \\ &= \overline{zero1}.zero0 + zero0.zero1.zero2 \end{aligned} \tag{5.13}$$

5.2 Implementation

The Address Generator was implemented in 0.7μm es2 CMOS, using two signed digit adders with multiplexers, one sign detection unit, four dimension counters and four difference-register pairs. As only initial design investigation and verification was undertaken, an eight bit address generator was implemented, although extension to 32 bits is a simple matter. The propagation free properties of sign-digit adders means that a 32-bit adder takes the same computation time as an eight bit adder, and the sign detection unit will require another two multiplexers. As a single adder and the sign detection unit comprise the critical path, it is felt that the extension to 32-bits will require of the order of a single nanosecond extra per cycle.

The VLSI layout is of a similar format to that of the schematic in Figure 5.3. The signed digit adders with multiplexers are laid out one above the other, with the adder with the 'delta' offsets, Adder1, on top and the adder with the 'delta-Q' offsets, Adder2, on the bottom. The sign detection unit in placed below the bottom adder, Adder2. The outputs from both the adder units are directed upwards towards the output latches, after which they can be multiplexed to select the correct output. The latches that contain the differences for the address generation are on the top right of each adder, and are loaded via an input bus that runs from left to right above the difference latches. The latches feed the adders via three 2-1 multiplexers configured as 4-1 multiplexers. A series of control buses run between the difference latches and the adders. These are fed with control signals generated by the dimension counters and the sign detect unit. The VLSI layout of an eight bit address generator is shown in Figure 5.16

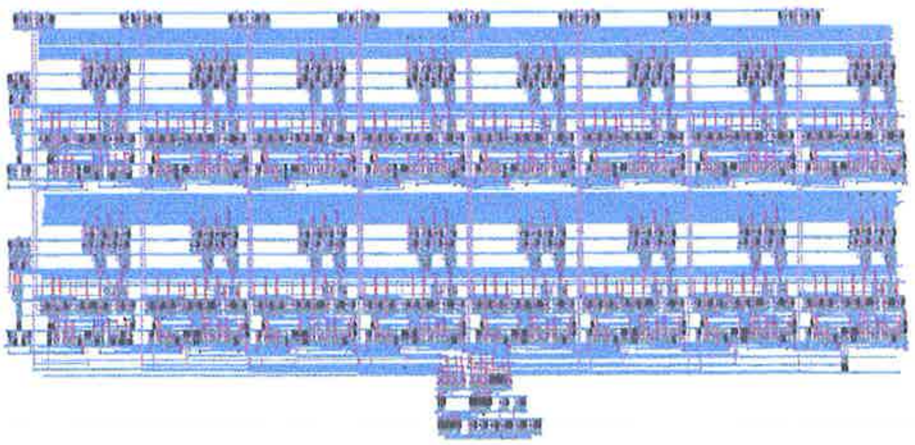


Figure 5.16: **VLSI Layout of an Eight-bit Address Generator**

The eight bit address generator consumes approximately 0.7956mm × 2.11mm = 1.68mm^2 of silicon when implemented in 0.7μm es2 CMOS, and uses 4383 transistors. The separation between cells is approximately 0.2569 mm, so a 32-bit implementation would consume approximately $1.68\text{mm}^2 + 24 \times 0.2569\text{mm} \times 0.7956\text{mm} = 6.59\text{mm}^2$ of silicon. There are 491 transistors in each digit cell plus 24 for each output latch set, so the 32-bit implementation would contain approximately $(491 + 24) \times 24 + 4383 = 16743$ transistors.

The eight bit address generator was simulated for speeds up to 100 MHz, using HSpice. The results of the simulations are shown in Figures 5.17 to 5.19 for a variety of input data. The input data used demonstrate selected cases of the examples presented in Section 5.0.3 and Table 5.2.

Figures 5.17a) to d) show the case of accessing a matrix in normal form. The input data, taken directly from Table 5.2, is:

Input Data:	Base	$\Delta 1$	$\Delta 2$	$\Delta 3$	$\Delta 4$	$\Delta 1 - Q$	$\Delta 2 - Q$
Value:	0	1	1	0	0	-14	-14

Input Data:	$\Delta 3 - Q$	$\Delta 4 - Q$	n1	n2	n3	n4
Value:	-15	-15	5	3	0	0

The tabulated results are shown in Table 5.5. This provides the 'D' output from *Adder1* and the 'Q' output from *Adder2*, as well as the 'Neg' output from the sign detector unit.

Figures 5.18a) to d) show the case of accessing a transposed two dimensional matrix. The input data is:

Input Data:	Base	$\Delta 1$	$\Delta 2$	$\Delta 3$	$\Delta 4$	$\Delta 1 - Q$	$\Delta 2 - Q$
Value:	0	5	-9	0	0	-10	-24

Input Data:	$\Delta 3 - Q$	$\Delta 4 - Q$	n1	n2	n3	n4
Value:	-15	-15	3	5	0	0

These simulation plots have been tabulated in Table 5.6, showing the output 'D' data from *Adder1*, the output 'Q' data from *Adder2* and the Sign Detector output 'Neg'.

Figures 5.19a) to d) are the HSpice simulation results of a prime factor mapping. The input data is:

Input Data:	Base	$\Delta 1$	$\Delta 2$	$\Delta 3$	$\Delta 4$	$\Delta 1 - Q$	$\Delta 2 - Q$
Value:	0	3	8	0	0	-12	-7

Input Data:	$\Delta 3 - Q$	$\Delta 4 - Q$	n1	n2	n3	n4
Value:	-15	-15	5	3	0	0

These simulation plots have been tabulated in Table 5.7, showing the output 'D' data from *Adder1*, the output 'Q' data from *Adder2* and the Sign Detector output 'Neg'. Note that output address is from the 'D' digits if '*Neg*' is high, and from the 'Q' digits if '*Neg*' is low.

Other mappings such as submatrix extraction and circulant addressing were simulated, and found to operate correctly. It is felt that, as the arithmetic is propagation free, the address generator would operate correctly when extended to 32 bits, especially with the move to the 0.5μm CMOS process that will soon be available to the Department.

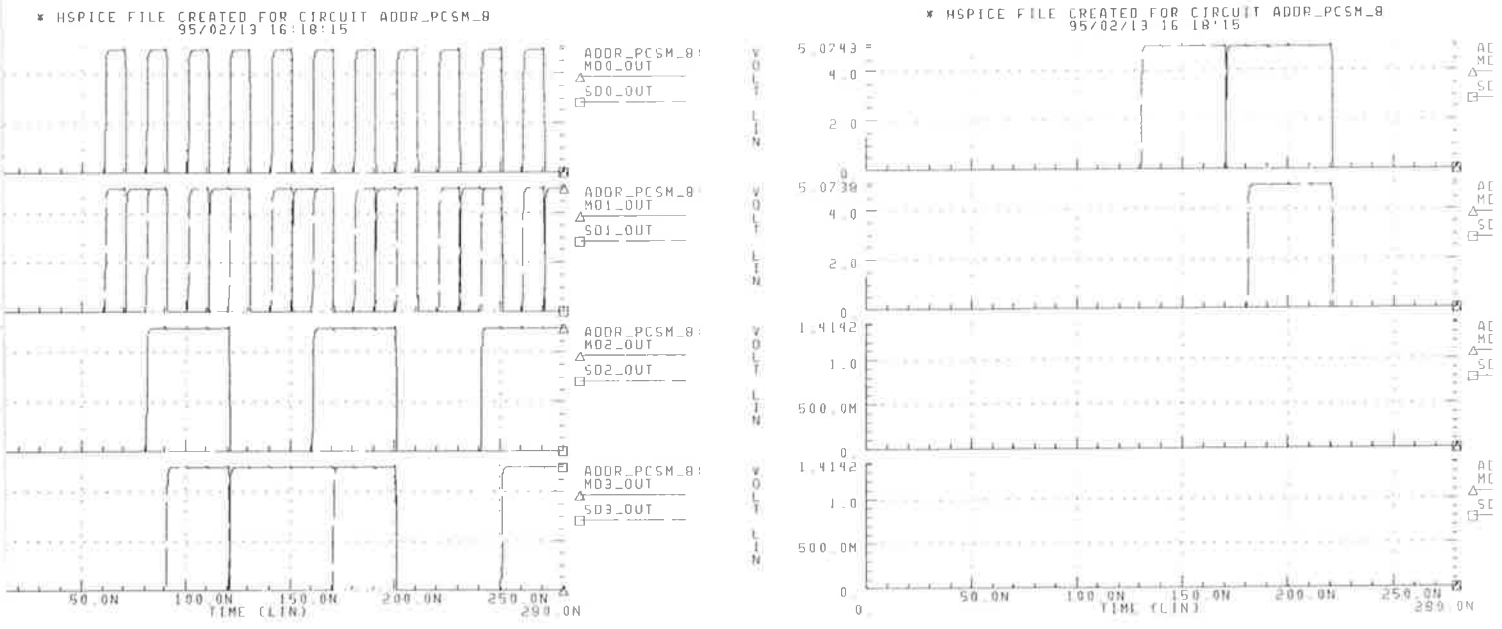


Figure 5.17: a) **D0 to D3** b) **D4 to D7**

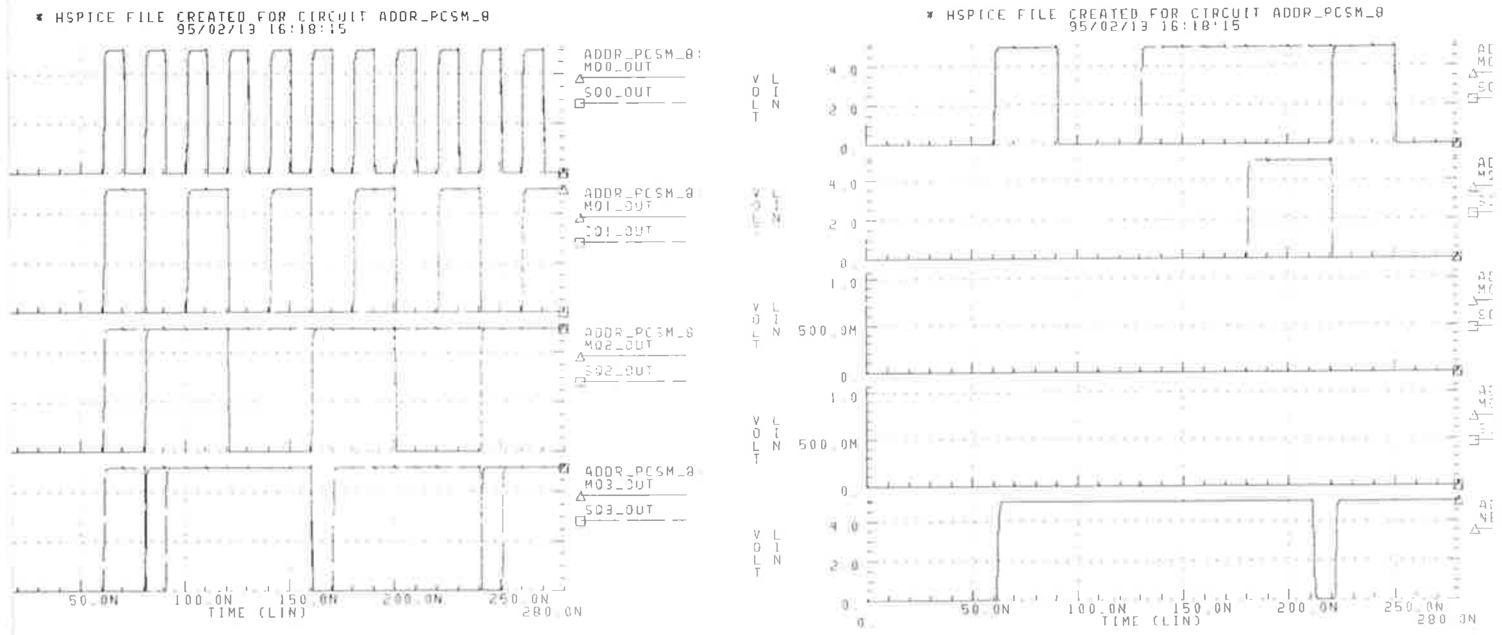


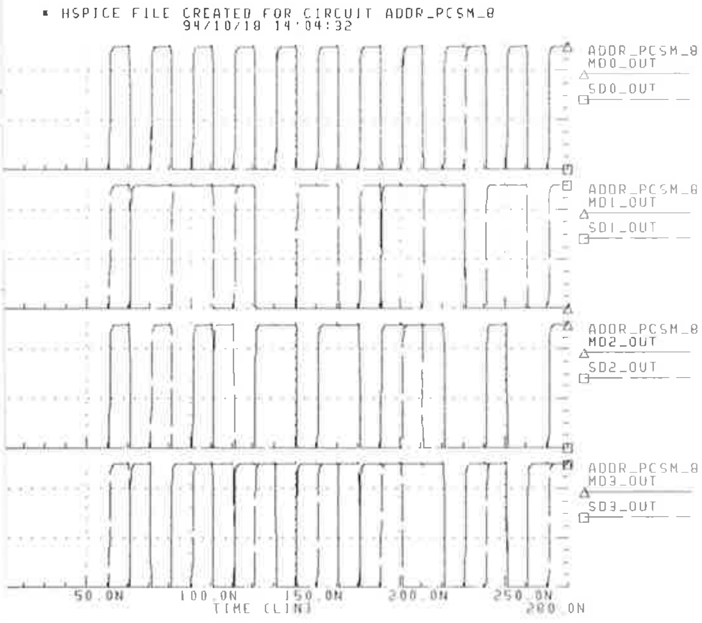
Figure 5.17: c) **Q0 to Q3** d) **Q4 to Q7 & Neg**

Figure 5.17: **Simulation of Normal Address Generation**

Cycle	D0	D1	D2	D3	D4	D5	D6	D7	Addr. Out
1	0	0	0	0	0	0	0	0	0
2	1	-0	0	0	0	0	0	0	1
3	0	1	0	0	0	0	0	0	2
4	1	-1	1	0	0	0	0	0	3
5	0	0	1	-0	0	0	0	0	4
6	1	-0	1	-0	0	0	0	0	5
7	0	1	1	-0	0	0	0	0	6
8	1	-1	0	1	0	0	0	0	7
9	0	0	0	1	-0	0	0	0	8
10	1	-0	0	1	-0	0	0	0	9
11	0	1	0	1	-0	0	0	0	10
12	1	-1	1	1	-0	0	0	0	11
13	0	0	1	-1	1	0	0	0	12
14	1	-0	1	-1	1	-0	0	0	13
15	0	1	1	-1	1	-0	0	0	14

Cycle	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Neg. Out
1	0	0	0	0	0	0	0	0	0
2	-1	1	-0	-0	-1	0	0	0	1
3	0	1	-0	-0	-1	0	0	0	1
4	-1	0	-1	1	-1	0	0	0	1
5	0	0	-1	-1	0	0	0	0	1
6	-1	1	-1	-1	0	0	0	0	1
7	0	1	-1	-1	0	0	0	0	1
8	-1	0	-0	-1	0	0	0	0	1
9	0	0	-0	-1	-0	0	0	0	1
10	-1	1	-0	-1	-0	0	0	0	1
11	0	1	-0	-1	-0	0	0	0	1
12	-1	0	-1	0	-0	0	0	0	1
13	0	0	-1	-0	-0	0	0	0	1
14	-1	1	-1	-0	-0	-0	0	0	1
15	0	1	-1	-0	-0	-0	0	0	1

Table 5.5: **Resultant Addresses for Normal Addressing Data**



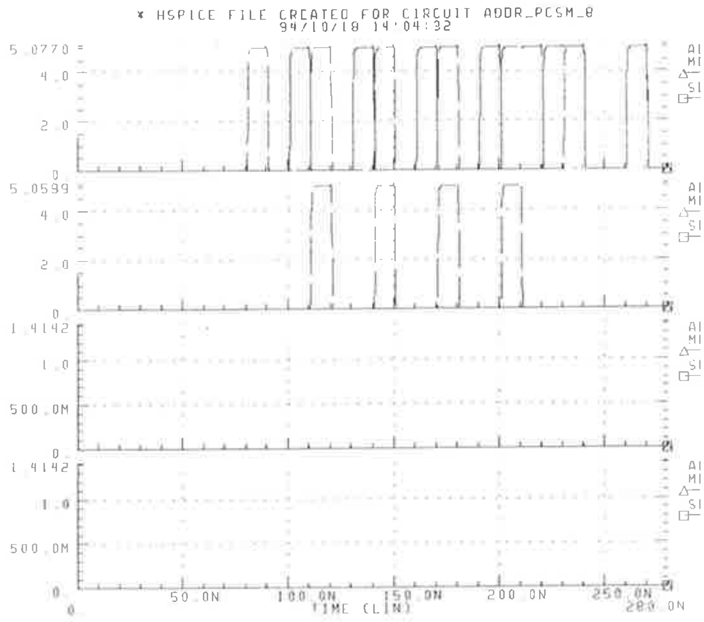
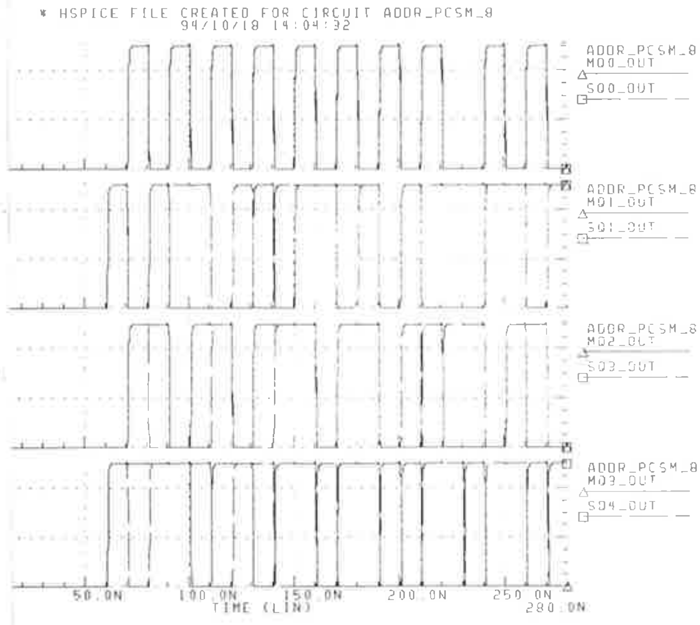


Figure 5.18: a) **D0 to D3** b) **D4 to D7**



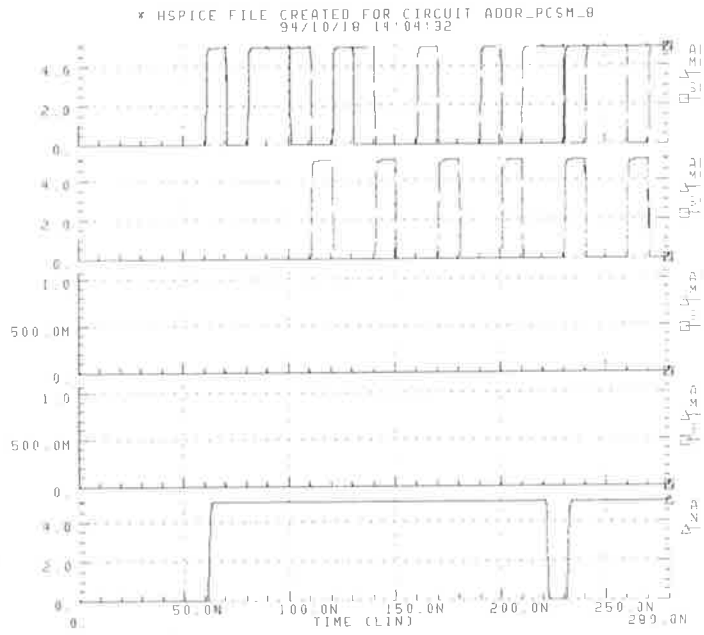
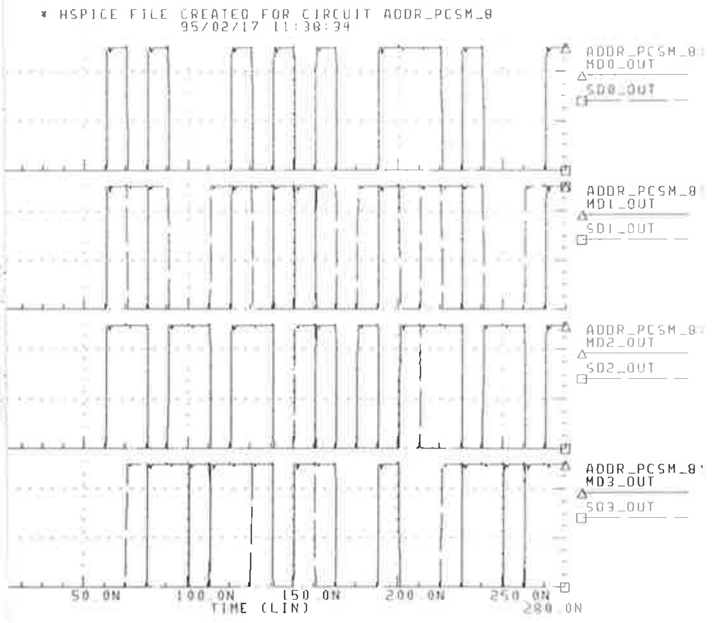


Figure 5.18: c) **Q0 to Q3** d) **Q4 to Q7 and Neg Out**

Figure 5.18: **Simulation of Transposed Address Generation**

Cycle	D0	D1	D2	D3	D4	D5	D6	D7	Addr. Out
1	0	0	0	0	0	0	0	0	0
2	1	-0	1	-0	0	0	0	0	5
3	0	1	0	1	0	0	0	0	10
4	-1	1	-0	0	-0	0	0	0	1
5	0	-1	0	1	0	0	0	0	6
6	1	-1	1	-1	1	0	0	0	11
7	0	1	-0	-0	-0	-0	0	0	2
8	1	-1	0	1	0	0	0	0	7
9	0	0	1	-1	1	0	0	0	12
10	-1	0	1	-0	-0	-0	0	0	3
11	0	-0	0	1	0	0	0	0	8
12	1	-0	1	-1	1	0	0	0	13
13	0	0	1	0	-0	-0	0	0	4
14	1	-0	0	1	0	0	0	0	9
15	0	1	1	-1	1	0	0	0	14
Cycle	**Q0**	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Neg. Out**
1	0	0	0	0	-0	0	0	0	0
2	0	-1	0	-1	-1	0	0	0	1
3	-1	0	-1	1	0	0	0	0	1
4	0	-1	1	-1	-1	0	0	0	1
5	-1	-0	0	-1	-1	0	0	0	1
6	0	-0	1	-0	-0	0	0	0	1
7	-1	0	-1	1	0	-0	0	0	1
8	0	-0	0	-1	-1	0	0	0	1
9	-1	1	1	-0	-0	0	0	0	1
10	0	-0	-1	1	0	-0	0	0	1
11	-1	-1	-1	1	0	0	0	0	1
12	0	-1	0	-0	-0	0	0	0	1
13	-1	1	-1	1	0	-0	0	0	1
14	0	-1	-1	1	0	0	0	0	1
15	-1	0	0	-0	-0	0	0	0	1

Table 5.6: **Result of Simulation of Addressing for Transposed Access**



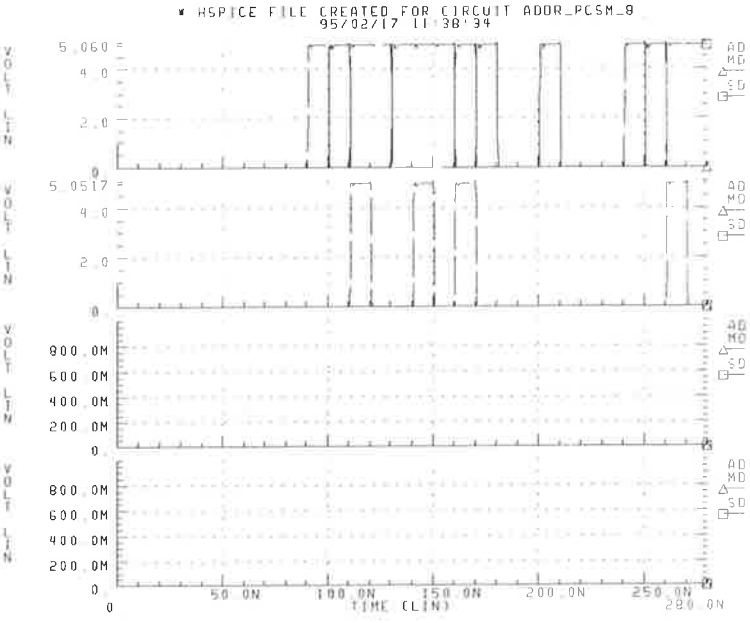
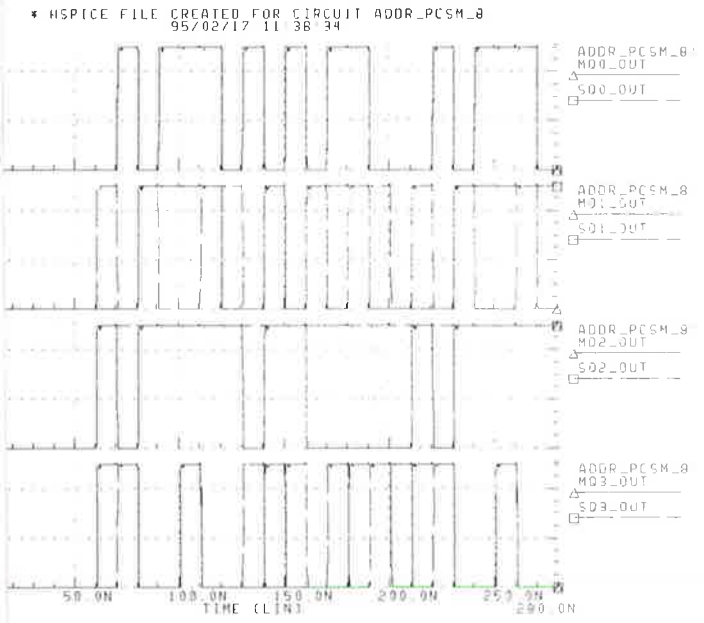


Figure 5.19: a) **D0 to D3** b) **D4 to D7**



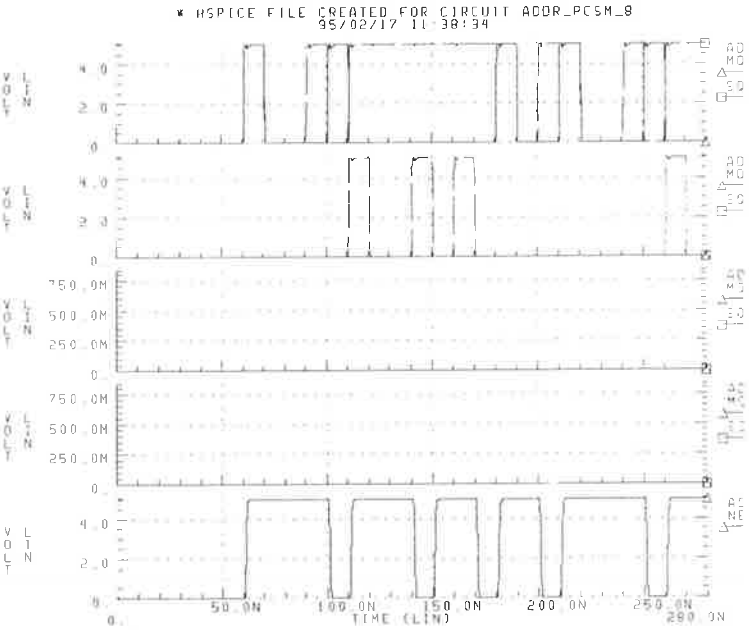


Figure 5.19: c) **Q0 to Q3** d) **Q4 to Q7 and Neg Out**

Figure 5.19: **Simulation of Prime Factored Address Generation**

Cycle	D0	D1	D2	D3	D4	D5	D6	D7	Addr. Out
1	0	0	0	0	0	0	0	0	0
2	1	-1	1	0	0	0	0	0	3
3	0	1	1	-0	0	0	0	0	6
4	1	-0	0	1	0	0	0	0	9
5	0	0	1	1	-0	0	0	0	12
6	0	0	1	-0	1	0	0	0	5
7	0	-0	0	1	-0	-0	0	0	8
8	1	-1	1	1	-0	0	0	0	11
9	0	1	1	-1	1	0	0	0	14
10	1	-0	0	0	1	-0	0	0	2
11	0	1	-0	-1	1	0	0	0	10
12	1	-0	1	1	-0	-0	0	0	13
13	0	0	0	0	1	0	0	0	1
14	0	-0	1	0	0	0	0	0	4
15	1	-1	0	1	0	0	0	0	7
Cycle	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Neg. Out
1	0	0	0	0	-0	0	0	0	0
2	0	-0	-1	1	-1	0	0	0	1
3	-1	0	-0	-1	0	0	0	0	1
4	0	-1	-1	0	0	0	0	0	1
5	-1	1	-1	0	-0	0	0	0	1
6	-1	1	-1	-1	1	0	0	0	0
7	-1	-1	-1	0	-0	-0	0	0	1
8	0	-0	-1	0	-0	0	0	0	1
9	-1	0	-0	-0	-0	0	0	0	1
10	0	-1	-1	1	-0	-0	0	0	0
11	-1	0	-1	-0	-0	0	0	0	1
12	0	-1	-0	0	-0	-0	0	0	1
13	-1	1	-0	-0	-0	0	0	0	0
14	-1	-1	-0	1	-1	0	0	0	1
15	0	-0	-0	-1	0	0	0	0	1

Table 5.7: **Result of Simulation of Addressing for Prime Factored Access**

5.3 After Calculating the Offset

Once the offset has been calculated, it must be converted to an unsigned binary number, and have the base address added to it. As the address stream is directed at fetching data and there are no instruction fetches during the computation of a matrix producted, there will be no non-deterministic branches or jumps in the address stream. Therefore, the conversion to unsigned binary and the addition of a base address can be heavily pipelined operations. However, if a signed digit adder is used to add the base offset, this will only take a single cycle, and the conversion to binary form can take place after the base offset has been included.

One more addition to Marwood's difference engine that is of use is an increase in the number of base offsets that can be selected to be added. The reason for this is for the direct solution of Gauss-Jordan elimination. One base register will hold the next pivot block starting (base) address, one will hold the row normalisation starting address, one will hold the update phase starting address and the fourth will hold the starting address of the next phase. Each starting (base) address is actually calculated in the previous phase, so no storage or extra calculation will be required, merely a four way base-select multiplexer and feedback path from the base address to the base offset registers.

The full extended address generator is shown in block form in Figure 5.20.

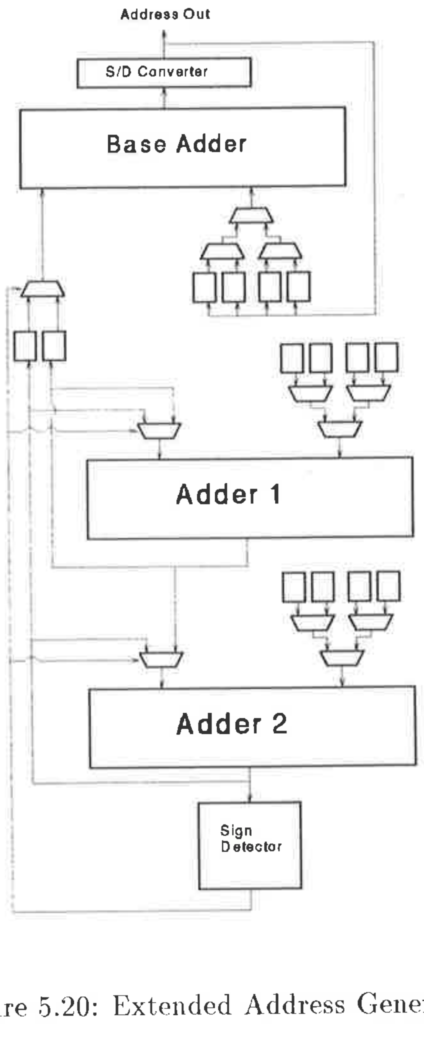


Figure 5.20: Extended Address Generator

5.4 Conclusion

In this chapter, Marwood's difference engine was extended to four dimensions and implemented using signed-digit techniques. The inclusion of the third and fourth dimensions allow the difference engine to directly compute the addresses required for partitioning an arbitrary sized matrix on a fixed sized array without external intervention. The use of signed-digit arithmetic for this critical component, together with the new parallel configuration, allows the system to produce new addresses at a rate of approximately 100 MHz.

Chapter 6

Memory Interface

6.1 Introduction

The memory subsystem for the matrix engine must be capable of supplying two operands and receiving one answer every cycle, which from Chapter 5 was suggested to be approximately ten nanoseconds. Such a high data request and retirement rate is typical of vector supercomputers, for which elaborate memory systems have already been designed [89, 77] *etc.*. These typically exploit vector and matrix structures by implementing heavily interleaved memory systems that rely on the assumption that a fetched matrix will have logically adjacent matrix elements in physically adjacent memory locations. Unfortunately, in many embedded applications, such as the Discrete Fourier Transform with prime factor mapping, this is not the case, as the applied matrices will be created from selected data within a larger amount of data. Therefore, a system that can support both the 'standard' access patterns typically used in vector systems and the 'non-standard' access patterns used for prime factor mapping, transposed mappings, etc is required. This chapter will describe such a system, based on a four-bank bank-switching cached system, using standard components except for a custom bank-switching chip.

6.2 Applying and Extracting the Outer Product

As the only function performed by the array is the outer product, with other functions handled by the data controllers, a brief discussion of how the outer product is extracted is useful.

6.2.1 Extracting the Outer Product

When the array is unloaded (the outer product is extracted), each accumulator in the array moves its data into a latch. The data is then shifted serially along a row or column (depending on whether storage in row- or column- major is required) until it reaches the array boundary, where it is unloaded in parallel form and then serially retired to memory. An extract from the processing array showing the unloading hardware is shown in Figure 6.1.

Note that the transpose of a matrix stored in row-major form is the same matrix stored in column-major form, and vice-versa. Therefore, if the transpose of a matrix is required[1], the matrix can simply be extracted in the opposite form from usual.

[1]As is noted in Chapter 3, the transpose is often required due to the common use of the matrix relationship $\mathbf{A}^T\mathbf{B}^T = (\mathbf{BA})^T$ to achieve the correct ordering.

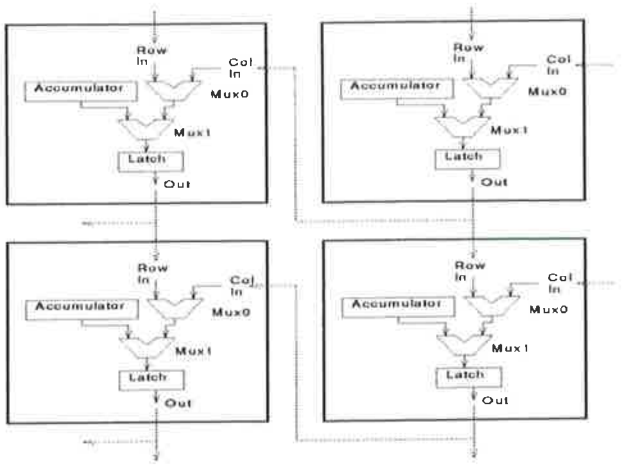


Figure 6.1: **Extraction Hardware**

6.3 Memory Overview

Although the memory bandwidth requirement for the matrix engine has been reduced by a factor of P for a $P \times P$ array compared to a MIMD[2] architecture with a similar number of processors, the memory bandwidth required to sustain the high speed possible is still very large.

The sort of memory access bandwidths required for the matrix engine are reminiscent of those of vector supercomputers such as the CRAY 2, Y-MP, Fujitsu VP200 and the NEC SX-2, in which data accesses are required at rates of the order of eight to sixteen nanoseconds per word [81, 80, 9]. These provide the data at the required rate by allowing only load and store accesses between memory and a number of 'vector registers', while the vector computational units operate only on data from the vector registers. This type of architecture is known as a *vector-register* architecture, as distinct from a *memory-memory* architecture in which all data operands are loaded from and stored to memory. Additionally, the only major vector architecture that routes data through a cache is the IBM 3090VF [35].

The memory system that was designed is a combination of *vector- register* and *memory-memory* architectures. The advantages of recycling data through the array indicates that a large data cache is desirable.

The complete memory system can be considered in one of two ways, either

- a cached *memory-memory* system, in which the accesses are all to and from a cache which is supplied from a large main memory, or
- a *vector-register* system that uses static RAM modules to construct a very large register bank for which each vector has only a single place to which it can be allocated.

6.3.1 Difference From Vector Memory Systems

Although the array memory system has many similarities to those commonly found in a vector memory system, there are some differences. These can be summarised as:

- Non-uniform strides.
  The stride of an access is the difference in address of successive accesses. Thus, the access

[2]Multiple-Instruction, Multiple-Data

pattern

$$0, 1, 2, 3, \ldots$$

has a stride of one, or a *unit-stride*, whereas the pattern

$$0, 5, 10, 15, \ldots$$

has a stride of five. The access patterns of a vector architecture are greatly simplified by the assumption that all of the two loads and one store have the same stride. Therefore, a large, heavily-interleaved single memory can service all memory pipes, as shown below in Figure 6.2 for an interleaving factor of eight[3].

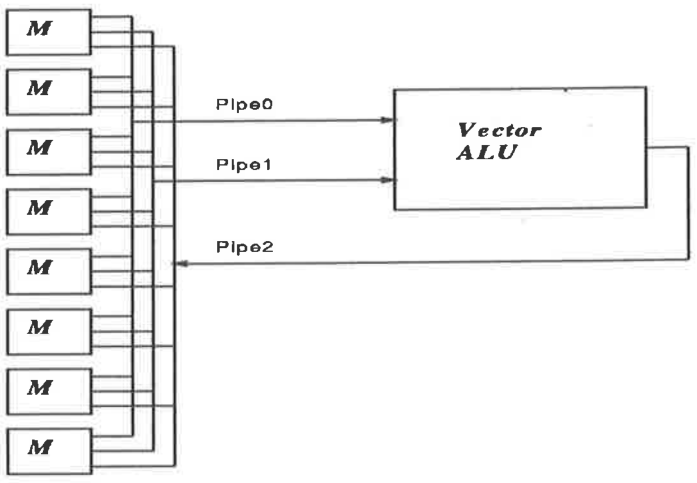


Figure 6.2: **Vector Memory Architecture.**

With reference to Figure 6.2, if memory pipe 0 accesses memory bank $M0$, pipe1 accesses bank $M2$ and pipe2 accesses bank 4, then in the next access cycle, a unit stride access will cause pipe 0 to access bank $M1$, pipe 1 will access bank $M3$ and pipe2 will access bank $M5$. If suitable delays are introduced to pipes 1 and 2, then arbitrary offsets in the access addresses can be included, while non-unit strides require only that the memory access time combined with the interleaving factor is less than the ALU cycle time. Thus, a stride of two in the system in Figure 6.2 requires that the ALU cycle time is greater than or equal to the memory cycle time (every second bank is skipped), whereas a stride of three requires that the ALU cycle time is greater than or equal to half the memory cycle time (all banks are accessed).

However, if the accesses have strides that are not all the same (non-uniform strides), then memory bank conflicts occur. For example, if pipe 0 has a stride of one starting at bank $M0$, and pipe 1 has a stride of two starting at bank $M2$, the bank access patterns will be those shown in Figure 6.3

The circled access of bank 6 shows the conflict that occurs when pipe 1 'catches up' to pipe 0.

Although great care can be taken with numerical computations to ensure that the strides of all accesses are the same, there are other cases such as the DFT using prime factor mapping that *require* the strides to be non-uniform. Therefore, a memory system capable of two load and one store access to memory with non-uniform strides is desirable.

[3]Most current supercomputers have at least 64 banks. Eight banks is chosen for convenience

*Pipe*0	0	1	2	3	4	5	⑥	7 …
*Pipe*1	2	4	6	0	2	4	⑥	0 …

Figure 6.3: **Non-uniform Stride Conflict.**

- Predictable Address Sequences
  Although vector memories are typically heavily pipelined [89, 80], they are also subject to the affects of instruction branch operations. Therefore, if the memory system is too deep, long delays may result in the event of a branch or jump instruction. However, as the matrix operation performs a complete matrix operation (multiplication, Gaussian elimination etc) with one instruction, no branch instructions will occur during the matrix operation, so a relatively large number of addresses can be precalculated before being sent to memory, thus preparing the system in advance to select the correct memory bank, check tags, etc.

6.4 Cache

With the decision to use bank swapping to move data quickly between the output and inputs, another problem presented itself. If the main memory is swapped, the data in the data controller caches would need to be flushed before either the memory bank or the cache can be read or written to. The simple solution to this is to attach the cache to the memory bank rather than the data controller, and move the cache with the memory.

The questions that arose in the design of the cache included:

- How large to build the cache?
- What should the block size of the cache be?
- What associativity should be used?
- How soon after a miss can the next address be accessed?
- How is a 'write' dealt with?

These questions can have different answers, depending on the application run. However, answers to the questions are suggested below.

6.4.1 Cache Size

The answer to this is the same as for any other cache system - as large as possible while still remaining cost effective. However, a cache that is to small may be more hinderence than use, as the tagging of cache blocks may cause an additional overhead with no added benefit.

6.4.2 Block Size

Increasing the block size attempts to reduce the block fetch penalty by amortising it over a larger number of accesses.

If the miss penalty is P, the refill time per word from memory is R, the block contains B words and the time to read a word from the cache is H, the time to access all words in a block is:

$$Time1 = P + (B - 1)(H + R) \tag{6.1}$$

so the average access time per word is:

$$av.\ Time1 = \frac{P + (B-1)(H+R)}{B} \tag{6.2}$$

If the block size is increased such that $(B-1)(H+R) >> P$ then the average access time per word approaches $(H + R)$. This suggests that a very large block size is used. However, if not all of the block is used by the processor, the unused accesses from memory represent wasted cycles. If there are W words in the block that are not required before the block is overwritten, then the time to access all the required words is:

$$\begin{aligned} Time2 &= P + (B-1)R + (B-W-1)H \\ &= P + (B-1)(R+H) - W.H \end{aligned} \tag{6.3}$$

and the average access time per *used* word is:

$$\begin{aligned} av.\ Time2 &= \frac{P + (B-1)R + (B-W-1)H}{B-W} \\ &= \frac{P + (B-1)(R+H) - W.H}{B-W} \end{aligned} \tag{6.4}$$

Therefore, if W is close to zero, Equation 6.4 approximates Equation 6.2. However, as W becomes large ($W \rightarrow B$), the average access time in Equation 6.4 becomes

$$av.\ Time2 \rightarrow \frac{P + (B-1)R}{0}$$

which tends toward infinity[4]. Additionally, the larger the block size, the fewer blocks can be kept in a particular sized cache. Therefore, a block size that is too large potentially can reduce the hit rate of the cache if only parts of each block are required.

Obviously, the idea is to use a block size that is as close as possible to the number of words used by the processor. The optimum size of a block, therefore, depends not only on the algorithm being executed, but also on problem specific considerations such as problem size, data configuration (the access pattern required to extract the data),etc. As the address pipeline can be quite deep, it is feasible to shift the internal cache tag by an arbitrary amount before tag comparison. Therefore, the block size can be chosen dynamically from a range of values from (say) 16 to 1024 at initialisation, depending on the hardware included to support a particular block size.

6.4.3 Associativity

Cache associativity is the number of positions in a cache that a block can be placed when it is loaded into the cache. The choices are:

- A single position. Each block from main memory may be present in only one place in the cache. This is known as a *Direct mapped* cache. Generally, the block is mapped into the cache block that is the (memory block address) modulo (the number of blocks in the cache). The advantage of this scheme is that only a single tag location needs to be checked to see if a block is present in the cache.

[4]Of course, the zero on the denominator is only present for the case of $B = W$, ie none of the words in the cache block load are used - clearly a ridiculous case

- Any position, known as *Fully Associative*. Therefore, a supposedly 'fair' block allocation scheme would fill all empty blocks in the cache before overwriting an existing cache block. However, as the block may reside in any location in the cache, or not at all, all cache tags must be simultaneously check for the presence of a block.

- Any of a selected set of positions. Known as *Set Associative*, this scheme is a mixture of Direct Mapping and Fully Associative, in that the set of locations in which a block may be placed is fixed (direct mapped), but any location within the set may be used to store the block (fully associative).

Generally, set associative cache provide a better hit rates than direct mapped caches, without the massive complexity demanded by fully associative caches [35]. However, as the caches for our system will generally be addressing matrices with very regular access patterns, set associative caches lose some of their appeal, as a set associative cache using a least recently used replacement (LRU) policy is no better than a direct mapped policy for addressing a well stored partitioned matrix. In fact, simple simulations indicate that, coupled with the added overhead due to the increased complexity of a set associative cache, a direct mapped cache provides *better* performance than a set associative cache. These simulations modelled matrix address patterns and maintained cache access data to monitor the frequency of cache hits and misses and their effect on system performance. For this reason, a direct mapped cache is recommended.

6.4.4 Latency After a Miss

The section on block size (Section 6.4.2) assumed that if a cache miss occurred, then the block in which the missed word belongs was fetched from memory, the missed word was supplied, and the remainer of the block also written to cache memory. Any subsequent reads to the one that caused the cache miss were stalled until the block refill had finished. A major saving can be made here using 'data streaming' for accessing data as it arrives during a refill, especially if large block sizes are used.

The idea of 'streaming' is very simple, and most advanced microprocessors use some form of instruction streaming. The concept behind instruction streaming is that it is very likely that instructions subsequent to the instruction that caused the cache miss and refill will also be in the fetched block, and in fact be in adjacent locations to the instruction fetched. Therefore, considerable time can be saved by monitoring the addresses within the block as they are loaded into the cache, and using the data as it is loaded, rather than waiting for the refill to complete before the next instruction is fetched.

As the matrix engine implements a complete matrix operation such as matrix multiplication or Gauss-Jordan elimination with a single instruction, the advantage of instruction streaming is not of much use. However, due to the large amount of data that is moved from main memory into cache memory in the order used by the matrix engine, some form of data streaming would provide a great improvement in performance.

As the matrix address access pattern is well defined, and a well ordered matrix extraction will typically contain accesses to large amounts of data stored in successive locations in memory, the data written into cache memory during a block refill is typically also required for a future access. Therefore, data streaming will allow the data to be read directly into the array simultaneously to the cache refill, thus potentially saving the 'block size minus one' cache accesses that may be required if streaming is not implemented.

6.4.5 Write Policy

The question of which write policy to use ultimately provides little choice. Of the two main policies, those of write-through and write-back, the latter is the only suitable approach. It only remains to explain why this is the case, and to consider details.

To see why a write-through policy is not feasible, it is necessary to consider its place in the matrix processor system and also the reason why it works in a conventional system.

On a store, a write-through cache writes the data to both the cache *and* main memory. Therefore, main memory always has an updated copy of the elements in the cache. Write-through succeeds because the percentage of time spent performing writes is small, and the cache acts in many ways like a large buffer as far as writes are concerned. However, if a very large number of successive writes occur, main memory can not keep up with the cache, and the processor will 'write-stall' until sufficient writes have been retired to main memory. This is just the case with the output data controller for the matrix processor - there are no operations *but* write (store) operations. Therefore, the main memory will continue to write-stall, and thus stall the entire engine for matched read-write operations.

A write-back cache suffers in part from a similar problem, but to nowhere near the same extent. However, many out-of-order or scattered operations can be handled in a much better manner using a write-back cache, and the main memory can support the block writes typical of write-back caches much more efficiently (see Section 6.5).

6.4.6 Implementation

The complete system is basically comprised of three identical Data Controller/Memory pairs plus a Memory/(I/O) pair. A Data Controller /Cache Memory pair is shown in Figure 6.4.

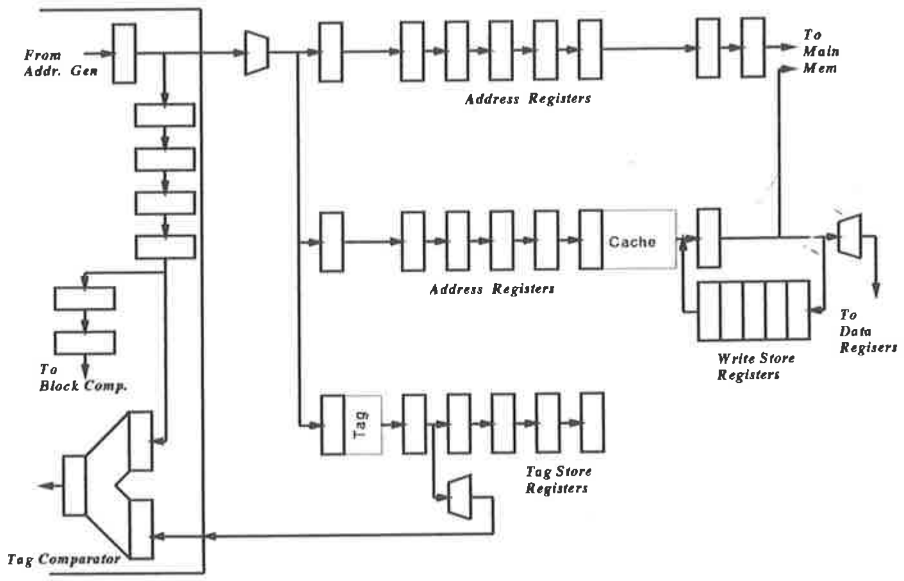


Figure 6.4: **Data Controller / Cache Memory pair**

The latches can be implemented using standard 74ASXX or 74VHCXX components. Octal latches such as Motorola's 74AS373 have maximum propagation times of approximately 5.0 nanoseconds, which leaves another 5.0 nanoseconds for propagation between latches in a 100 MHz system, which is ample for a well designed MCM or high speed system.

Although not strictly 'off-the-shelf', there was still an attempt to use readily available high speed Static RAM modules. The Static RAM (SRAM) modules are available from manufacturers such as Integrated Device Technologies (IDT), Electronic Design Inc. (EDI) and Motorola. None of the SRAM modules available in the data sheets [68, 40] come in a latching version with assess times of ten nanoseconds or less. Therefore, the cache memory will need to be 2-way interleaved, although it is only a matter of time until the timing constraints will allow a single bank of memory with cycle times operating at 100MHz.

The suggested SRAM module is the IDT7M4077, manufactured by Integrated Device Technologies, or similar. This is a $256k \times 32$ memory subsystem, with a read and write cycle times of fifteen nanoseconds. As 2-way interleaving is required, the minimum size of a cache will be 512×32bits = 2Mbytes. For a one megaword cache, two interleaved banks will be required. Therefore, as there are four memory ports, each with approximately four SRAM modules, sixteen memory modules will be needed in total.

To determine the cache tagging requirements, the cache size, address range and cache block sizemust be known. For convenience, it can be assumed that each data port can potentially access $2^{30} \approx 1.07 \times 10^9$ locations. The block size is determined at run-time but is in the range 16 to 1024 words, denoted by 'B'. The cache size is implementation dependent, but will contain approximately one million locations, denoted by 'C'. The depth of the tag SRAM is equal to the maximum number of blocks that can be contained in the cache SRAM. However, is the cache SRAMs are interleaved and the tag SRAM is not fast enough to be read for every access, then the tag RAM will need to be duplicated by the cache SRAM interleaving factor, 'I'. The length of a tag word is the total number of addressable locations divided by the number of locations in the cache. Using these criteria, the tag requirements can be calculated, and are presented in Table 6.1, assuming an interleaving factor of one.

Cache Size (kbytes)	Block Size (words)	Tag Depth (kbtyes)	Tag Length (bits)
16	16	64	10
16	64	64	8
16	256	64	6
16	1024	64	4
64	16	16	12
64	64	16	10
64	256	16	8
64	1024	16	6
256	16	4	14
256	64	4	12
256	256	4	10
256	1024	4	8
1024	16	1	16
1024	64	1	14
1024	256	1	12
1204	1024	1	10

Table 6.1: **Tag Requirements**

6.5 Main Memory

It is anticipated that the matrix engine will be used on very large problems in the fields of aerodynamics and fluid flow, among others, which involves the solution of very large, dense matrices. Typically, none of the matrix elements will be zero. Practical problems require matrix sizes of the order of hundreds to thousands of elements in a row or column ($\sim 200 \times 200 \rightarrow 10,000 \times 10,000$), resulting in storage requirements of $40kWords \rightarrow 100MWords$ which equates to $160kbytes \rightarrow 400Mbytes$ for single precision or $320kbytes \rightarrow 800Mbytes$ for double precision. Added to this the bank-swapping nature of the equation solving routines (Gaussian elimination - see Section 3.4) which at least doubles the memory requirements, and the need for very large amounts of main memory becomes apparent.

To keep the cost of the main memory to a reasonable amount, Dynamic Random Access Memory (DRAMs) should be used. DRAMs have the advantage over Static Random Access Memory (SRAMs) that they provide a much denser memory system at a cheaper price. This is paid for by the slower access times for a DRAM system and by the added complexity due to the multiplexed nature of a DRAM chip.

In an attempt to match cache operations to main memory, it was decided to concentrate mainly on block data transfers. While this may not be the ideal for embedded systems engaged in high-speed Digital Signal Processing (DSP) or large control systems (Kalman filtering, Neural networks), such systems typically will not use a cache/DRAM hierarchy, operating instead entirely out of the static RAM used for the cache. In such a system, the Tag checking can simply be turned 'off', and the main memory comprising of dynamic RAMs omitted. Thus, the main memory design is intended only for a numerical system solver type architecture.

The majority of DRAM chips available in the market place today are of an asynchronous nature. These require that the system place the upper half of the address on the input address pins, after which an address strobe is asserted. Then the lower half of the address is place on the address pins and a second address strobe is asserted. The internal structure of a $1M \times 4$ static column DRAM chip is shown in Figure 6.5 [68]. The static column refers to the fact that, once a particular row of the memory array has been selected using the $\overline{RAS}$ signal and the address pins $A0 \ldots A9$, any column of the selected row can be accessed by changing only the input address pins; ie no column address strobe ($\overline{CAS}$) signal is required.

The data sheets provided by Motorola give the time to read from a $1M \times 4$ static column DRAM to be 60 nanoseconds for the first read and 35 nanosecond for any successive read that falls within the same DRAM page. Therefore, for a ten nanosecond cycle time, the main memory needs to be four words wide, or use an interleaving factor of four. With an interleaving factor of four, a 32-bit word and four memory banks, the fundamental width of memory will be $4^3 = 64$ bytes. If these are composed of $1M \times 4$ DRAM chips, the minimum memory size will be $64Mbytes$ of main memory. Although this may appear to be a large amount, it is small compared to other systems of comparable performance.

Recalling that the cache SRAM is two way interleaved, the main memory can be considered to be a 2-way 64-bit wide memory rather than a 4-way 32-bit wide memory. This eases the data switching requirements at the main memory to cache interface, as less data needs to be pushed on to a smaller bus.

6.5.1 Implementation

As there is a very large pipeline for the cache memory system, the row address can be ready at the address inputs of the DRAM when a miss is signalled. Therefore, the row address strobe ($\overline{RAS}$) can be asserted simultaneously with the cache miss flag.

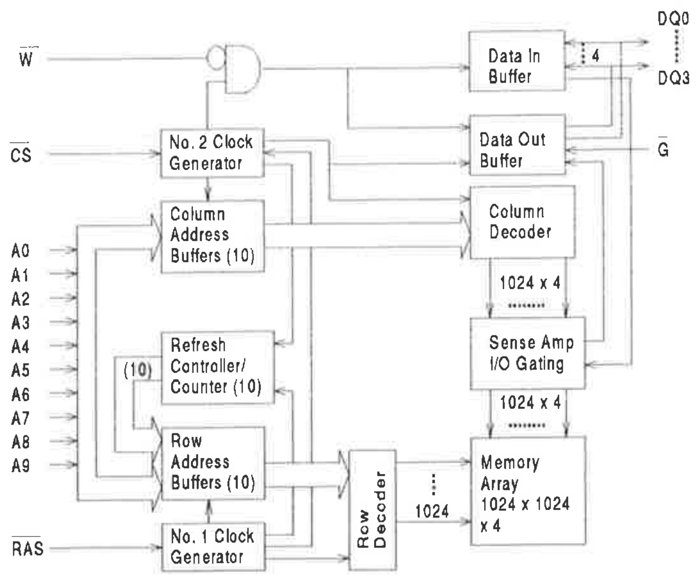


Figure 6.5: **Internal Structure of** $1M \times 4$ **Static Column DRAM**

A block diagram of the DRAM configuration is shown in Figure 6.6. The input address is partially applied to multiplexer Mux0, and then loaded into the 4-bit synchronous counters, which are used as block counters. In this way, an arbitrary sized block starting address can be created from any address within the block. The starting address is passed to the second multiplexer, Mux1, which separates the address into row and column addresses, the higher address bits first. Note that the row address (upper address bits) can be transmitted through multiplexer Mux1 while the block starting address is being calculated and loaded into the block counters. The row address is then latched into latches 0 to 3, and applied to all the DRAM arrays simultaneously. The $\overline{RAS0}$ to $\overline{RAS3}$ lines can then all be asserted.

Once the row address has been applied to the DRAM arrays, the column address (lower address bits) can be applied to latches 0 to 3 via multiplexer Mux1, and then the chip select signals ($\overline{CS0\text{-}3}$) asserted[5]. For all successive column addresses (block addresses), the latches and chip select signals are used in turn.

If the memory operation is a memory read, then the data output from the DRAMs is passed to latches 6 & 9 via multiplexers Mux2 & Mux3, and then to the SRAMs in the cache. As the cache SRAMs are 2-way interleaved (Section 6.4), then DRAM banks 0 & 2 feed SRAM bank 0, and DRAM banks 1 & 3 feed SRAM bank 1. While data is being applied to the output latches, it is also applied to the 74AS280 parity checking chips, which will flag a parity error in the event that there has been a data storage error.

If the memory operation is a memory write, the data to be written is applied to DRAM arrays via latches 4,5,6 & 7. Note that, although the write hold time for the MCM54402A is fifteen nanoseconds [68] and the chip select pin ($\overline{CS}$) could be used to control which DRAM bank data is written to, the physical connection between the DRAM chips would disrupt memory reads. Therefore, separate latches are used. While data is held in the input latches, it is also applied to the 74AS280 parity checking chips to generate a parity bit for each byte.

The initial timing of a burst read is shown in Figure 6.7. A burst write is similar, except that the data flows in the opposite direction.

[5]Recall that static column DRAMs require no $\overline{CAS}$ signal, and instead replace this signal with a chip select

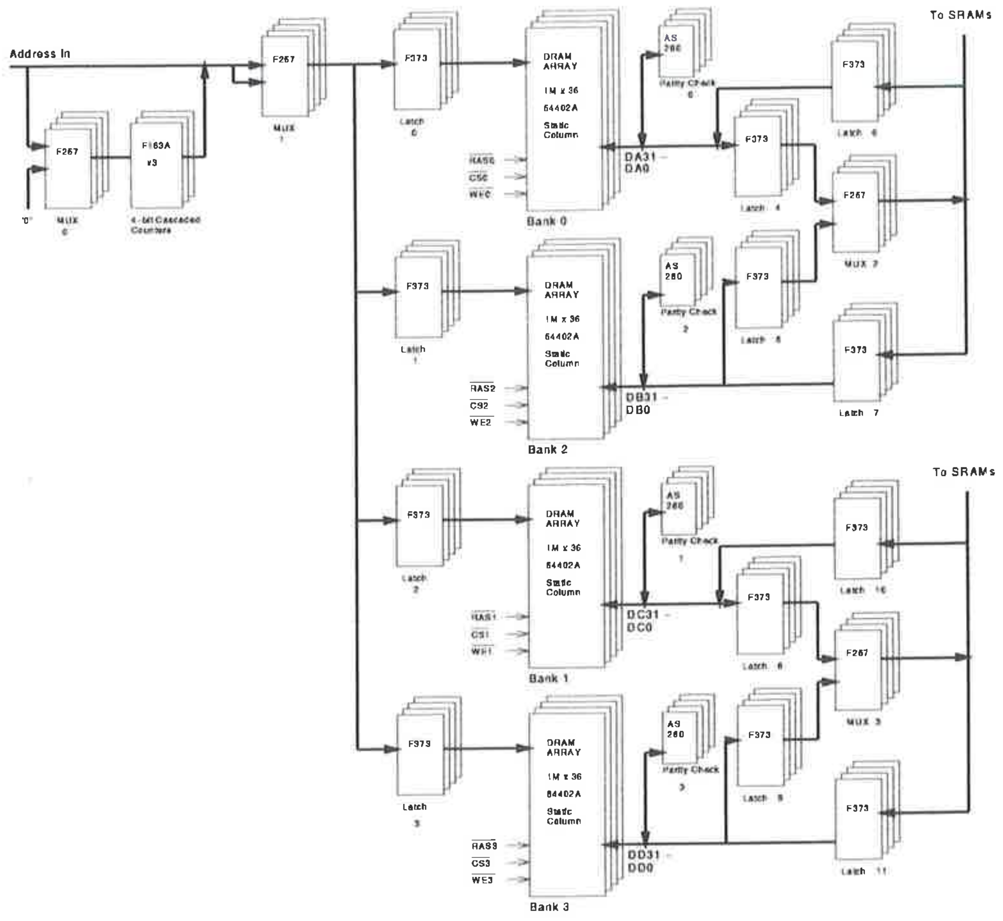


Figure 6.6: **Block Diagram of Four Way Interleaved DRAM Subsystem**

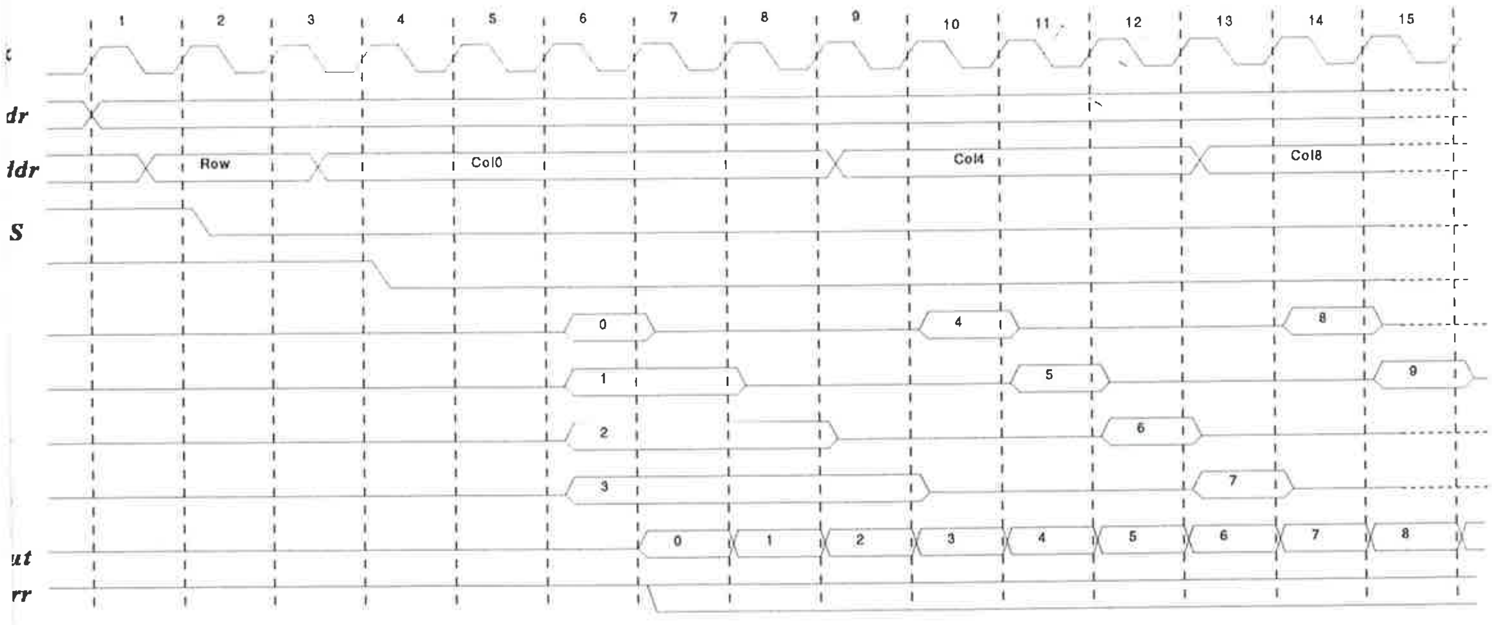


Figure 6.7: **Initial Timing of Burst Read**

6.5.2 Synchronous DRAMs

An alternative to using the asynchronous DRAM chips described above is to use the relatively new synchronous DRAM chips that are just becoming available from manufacturers such as Samsung Inc. etc. These DRAMs, denoted SDRAM for Synchronous Dynamic Random Access Memory, are similar to a core DRAM chip with some extra registers added for timing and burst control. A block diagram of the Samsung KM48SH2000-6 is shown in Figure 6.8 [13].

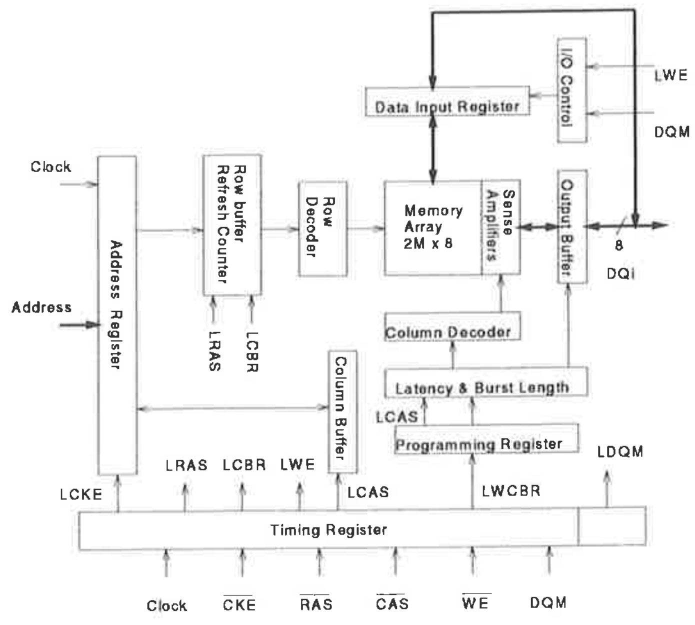


Figure 6.8: **Synchronous DRAM Internal Block Diagram**

The Samsung KM48SH2000-6 is a synchronous DRAM design configured as $2M \times 8$bits that can burst from the input address at 2-, 4-, or 8-bit, or full-page (512-bit) block sizes. It has a $\overline{CAS}$ latency of two cycles, and a maximum clock frequency of 100MHz. After an initial latency, the SDRAM chip cycles at the clock rate, ie provides a new byte every 10 ns. Once a burst has been completed, the outputs automatically enter a high impedance state. Additionally, memory chip precharging can commence two clock cycles before the burst is complete. Therefore, with the faster versions of the SDRAM that require two cycles of precharge, a new cycle can commence immediately the previous one has finished.

Using the $2M \times 8$ chips running at 100MHz, no interleaving is required to support ten nanosecond accesses. Therefore, the minimum memory increment size per bank is $2M \times 32 = 8$ Mbytes for a 32-bit word. Then for four memory banks, $4 \times$ 8Mbytes = 32Mbtyes is required, which is similar to the size required for asynchronous DRAMs. The difference between the options is the reduction in added complexity as the synchronous chips contain all the necessary counters, and there is no need for multiplexing. There will also be fewer memory chips using synchronous DRAMs, as they have a 16Mbit density instead of a 4Mbit density, although this is only an implementation detail.

control line ($\overline{CS}$)

6.5.3 Bank Size

The main memory of the system should be large enough to hold a very large matrix without requiring any access to (slower) Input/Output devices. The main memory can either be composed of one segment of thin DRAM chips, such as $4M \times 1$bit, or several segments of fewer wide DRAM chips, such as $1M \times 4$bit. This is shown graphically in Figure 6.9

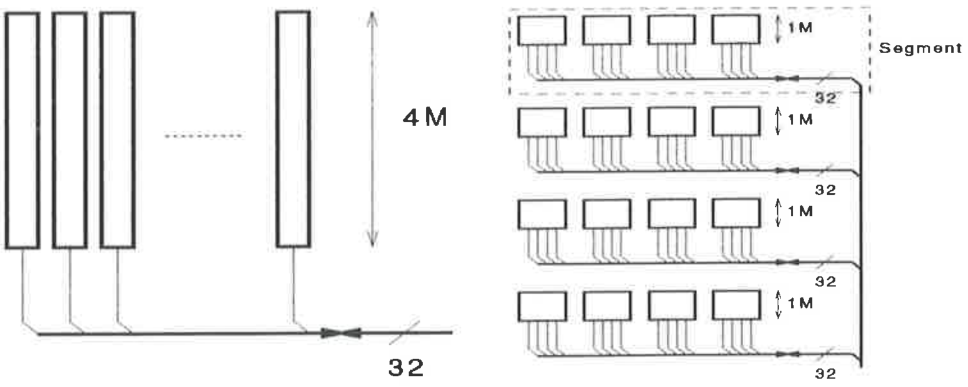


Figure 6.9: **Memory Bank Configuration using a) Thin & b) Wide DRAMS**

The structure using thin DRAMs (Figure 6.9a) is much simpler to implement as no selection between bank segments is required. However, the configuration using several segments composed of wide DRAMs has several advantages, despite the more complex control required. The first, and most obvious, is the possibility to interleave the segments or use a wide bus to supply data at a higher bandwidth. As alluded to in Chapter 5, an increase in bandwidth can be accompanied by the square of the increase in computational rate, depending on implementation. Thus, two way interleaving can quadruple the computational rate, 4-way interleaving will increase the rate by a factor of 16, etc.

The second advantage comes in the expansion of system memory. As memory control hardware generally requires equal size banks, the smaller segments will allow a smaller increase in main memory.

The third advantage is linked to the cache system, in particular the fact that the cache will be very large and of a similar size to a realistic segment size (eg 1MWord). This can be used to an advantage when cache write-backs are required. Given that the cache will be direct-mapped, and assuming a cache size equal to a segment size, then if a block in the cache is displaced and the data in the block needs to be written back to main memory, the incoming data (the replacement block) must be coming from a different segment. Therefore, the read from one segment can be initiated while the the write back to another segment is still completing, as in Figure 6.10.

Typically, the access time from the last write for a 70nsec static column 1Meg×4 dynamic RAM is approximately 65 nsec[68]. If the new block is in a different DRAM page, which is highly likely with the large block sizes available, then a RAS precharge period of approximately 40nsec will be required. Additional delays in the change from read to write will account for approximately 15nsec, for a total delay of 120nsec.

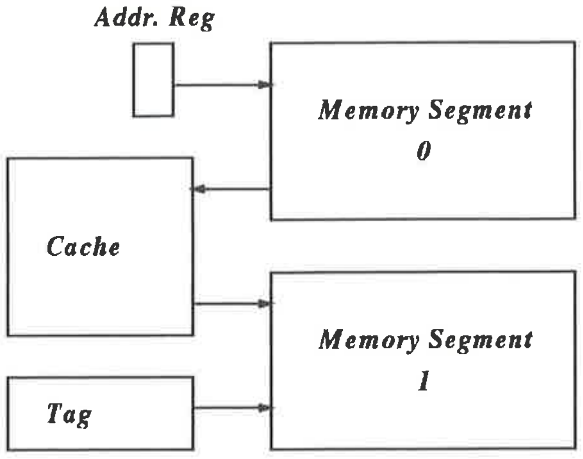


Figure 6.10: **Initiating a Read Simultaneously to Completing a Write**

6.6 Bus Exchangers

The requirement in the memory system that large banks of memory be swapped very rapidly presented a difficulty, in that standard off-the-shelf CMOS and AS-TTL chips were not fast enough. As we have access to very fast and reasonably priced 0.7 μm and 0.5 μm CMOS processes via TIMA-CMP in France, it was decided to design our own chips, which we designated 'Bus Exchangers', shortened to 'Bus_X'.

The Bus Exchangers must be able to connect any of the three matrix engine ports or the external port to any of the four memory subsystems. As there are two input ports and one output port to the matrix engine, the Bus Exchangers need to be bi-directional, although the directionality of the switches need only be determined at a memory swap. Additionally, they must be able to latch the transmitted data, as well as behaving as a transparent switch. A modification of the latch structure in Figure 6.11a) can be used to implement the latch/transparent features, which we shall refer to as the 'driver' cell (shown in Figure 6.11b)).

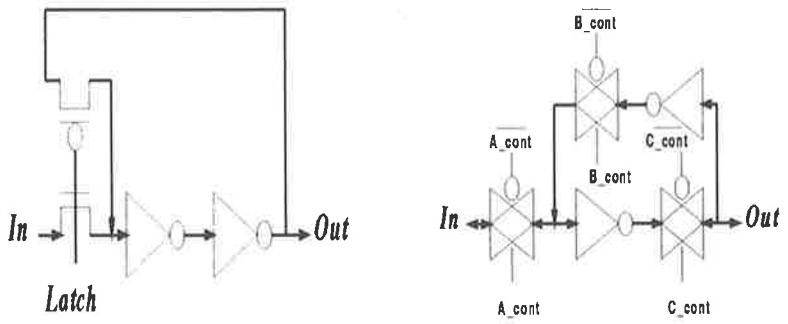


Figure 6.11: a) **Latch Cell** b) **Driver Cell**

The drivers need to push data through four p-type and four n-type pass transistors (configured as four transmission gates) across the chip and through two transmission gates toward the the outputs, so they need a relatively large driving capacity. To achieve this, three inverters are configured in parallel for each inverter. The transmission gates are paired to provide the two levels of transmission control while using minimal area. The VLSI layout of the driver cell is shown in Figure 6.12.

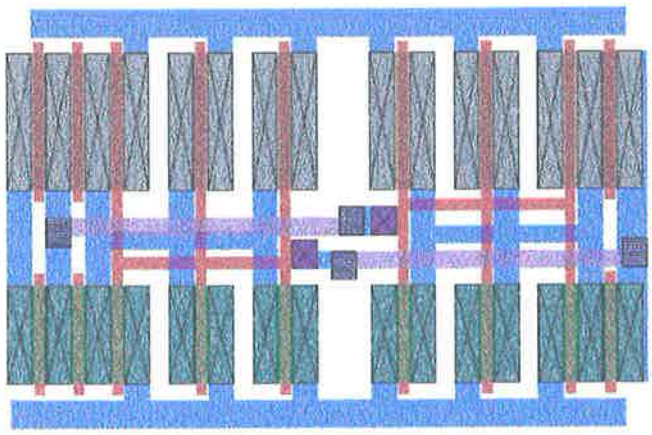


Figure 6.12: **VLSI Layout of the Driver/Transmission Cell**

If it is assumed that a read is data applied to '*In*' and a write is data applied to '*Out*', where '*In*' comes from an external source and '*Out*' is from the switches, then there are three possible modes of operation:

- Latched Read
  Data is applied to '*In*', where upon it is latched as well as being passed through to the output, '*Out*'. A_Cont is pulsed high, B_Cont is pulsed low and C_Cont is always high.

- Transparent Read
  Similar to a Latched Read, except that a change in the input is reflected by a similar change in the output, '*Out*', delayed by the propagation delay of the device. A_Cont is always high, B_Cont is low and C_Cont is high.

- Write
  Data passes directly from '*Out*' to '*In*'. A_Cont is high, B_Cont is high and C_Cont is low.

6.6.1 Control

The bus exchangers must produce some internal control signals from the input control signals. The inputs to the BusX units for control are:

- $r/\overline{w}$
  Read or write control signal. This signal controls the direction of data flow, either from one of four inputs to the single output (multiplexer) or from one input to one of four outputs (demultiplexer).

- $a0$,$a1$
  Used to determine which of the four banks the memory access is to.

- $t/\overline{l}$
  The control signaling whether the mux/demux is operating in the transparent mode or the latch mode.

- Ck
  The input clock. The clock *must* be inhibited in the event that a stall occurs when the BusX is operating in *latched* mode. Otherwise, incorrect data may be latched into the BusX.

The internal control signals can be broken into:

- Direction (r/w)
  The read/write signal determines the direction of data flow through the Bus Exchanger. This signal can be a buffered version of the input read/write signal.

- Select (Sel0, Sel1, Sel2, Sel3)
  These four signals are the decoded 'select one of four' signals. They are decoded from the input select signals and their complements ($a0$,$\overline{a0}$,$a1$, $\overline{a1}$). They can be decoded using four NOR gates, according to the equations:

$$
\begin{aligned}
Sel0 &= \overline{a_1 + a_0} && (6.5)\\
Sel1 &= \overline{a_1 + \overline{a_0}} && (6.6)\\
Sel2 &= \overline{\overline{a_1} + a_0} && (6.7)\\
Sel3 &= \overline{\overline{a_1} + \overline{a_0}} && (6.8)
\end{aligned}
$$

- Driver Control
  The signals a_cont, b_cont and c_cont in Figure 6.11 must be decoded for each of the drivers. As the connection to the bus must only be made once, the transmission gate controlled by a_cont must disconnect the driver from the bus unless the correct select signal (SelX) is present. a_cont is the only control signal that depends on the driver select signals. The control signals are:

$$
\begin{aligned}
\text{a\_cont} &= sel.\,(r.\,(t + l\Delta) + w) \\
&= \overline{\overline{sel.\left(r.\overline{(t + \Delta)}\right)}} && (6.9)\\
\text{b\_cont} &= w + l\nabla \\
&= \overline{\overline{\left(w + \overline{(\delta + t)}\right)}} && (6.10)\\
\text{c\_cont} &= r && (6.11)
\end{aligned}
$$

  where '*sel*' is the appropriately decoded select signal, t & l are the decoded transparent and latch mode signals, and Δ & ∇ are rising and falling pulses respectively.

Note that the control signals above are for data being applied to the drivers from off-chip for a read. The opposite driver will require a similar set of control signals, except with the 'r' and 'w' signals exchanged.

6.6.2 Read/Write Path

For data passing from one of four inputs to a single output, a 4-to-1 multiplexer is needed, while a 4-to-1 demultiplexer is required for data going the other way. As the Bus Exchangers are

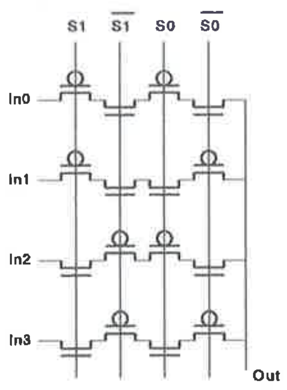


Figure 6.13: **Four-to-One Multiplexer/Demultiplexer**

bidirectional, data must be able to flow both ways through the mux/demux combination, and so transmission gates were used, rather than combinatorial logic. Such a 4-to-1 bidirectional mux/demux is shown in Figure 6.13.

The VLSI layout for this configuration is simple, and shown in Figure 6.14. Due to the regularity of the design, the VLSI section shown is in fact the transmission mux/demux for two bits, one moving from left to right and the other moving from right to left. The red vertical lines representing polysilicon are the mux/demux control signals. There are two pairs of complementary control signals running adjacently, denoted $\{a_0, \overline{a_0}\}$ & $\{a_1, \overline{a_1}\}$. An output signal from a driver will only appear on the metal2 line (purple) if the appropriate transmission gates are conducting. Otherwise, the signal will be blocked.

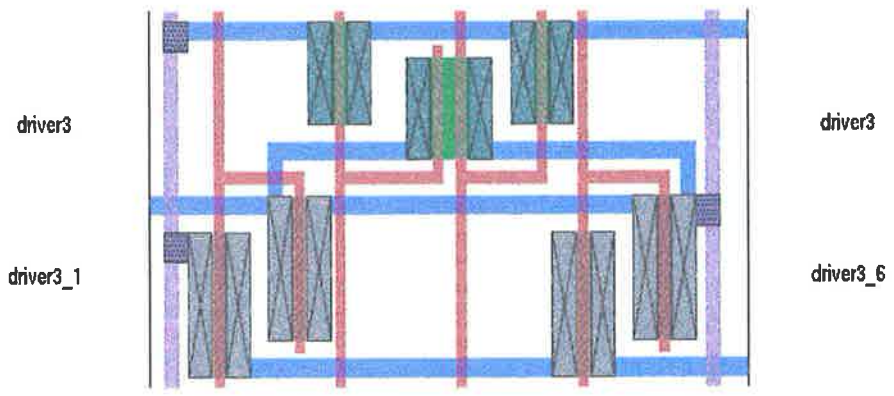


Figure 6.14: **VLSI Section of Four-to-One Mux/DeMux**

6.6.3 The Complete Bus Exchanger

Eight of the multiplexer/driver units were combined with a single control unit to produce a byte wide Bus Exchanger unit. The full Bus Exchanger unit is shown in Figure 6.15. The full unit consumes 0.585mm × 0.366mm = 0.214mm^2 of silicon in 0.7μm es2 CMOS, and uses 1191 transistors.

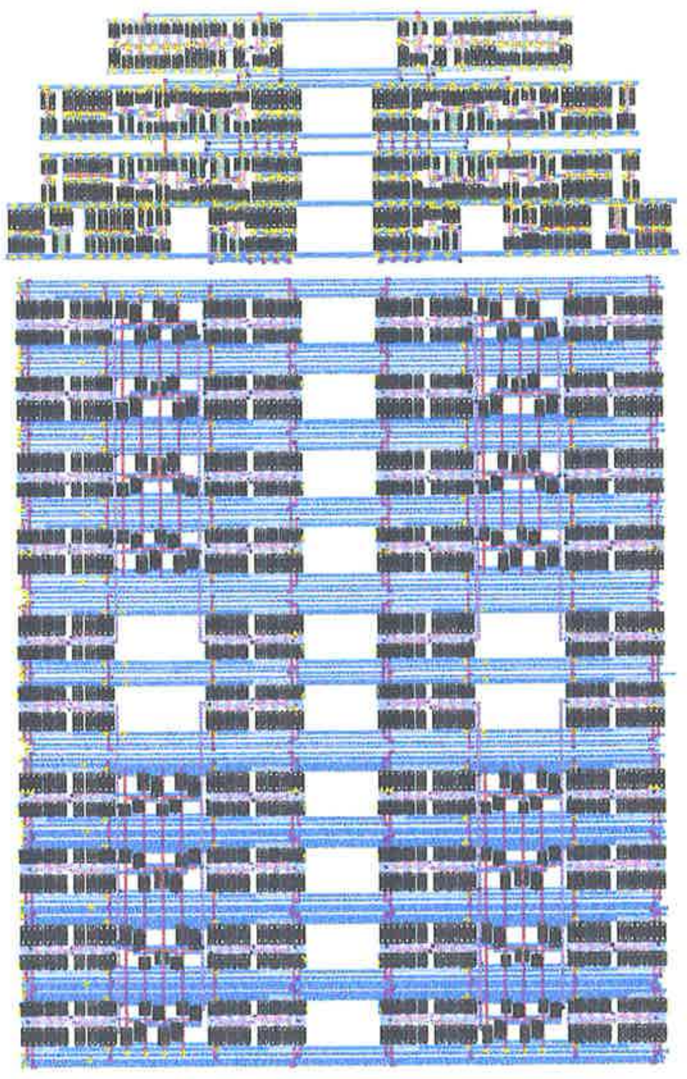


Figure 6.15: **Bus Exchanger VLSI Layout**

The Bus Exchanger was simulated using HSpice for several configurations. As each bit path is exactly the same as all the others, then simulations of one will produce the same results as simulations of any of the others. Therefore, although it was checked that each bit worked correctly, the results of only one bit path are shown except for the cases when the results are the same for each bit and can be shown simultaneously.

The first test case was when the Bus Exchanger was operating in 'latch' and 'read' mode, and the 'zeroth' input was always selected (a_1&a_0 both 0 volts). The resulting simulation results are shown in Figure 6.16. The plots show the outputs *a_out* to *h_out* following the inputs *a0_in* to *h0_in* approximately six nanoseconds after the rising edge of the clock pulse. The outputs are uneffected by various zero volt and five volt inputs applied to the other inputs of the Bus Exchanger.

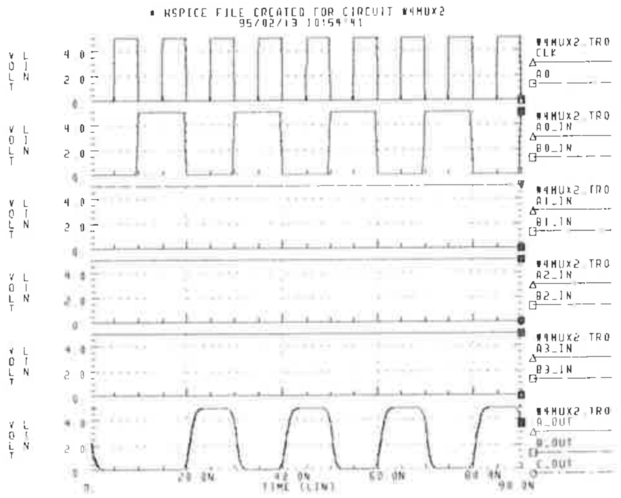


Figure 6.16: **Latched Read With Constant Select Signals**

Next, one of the select signals (*a_1*) was varied part way through the simulation, so that the input was changed from the 'zeroth' input to the 'second' input (eg. *a0_in* to *a2_in*). The results are shown in Figure 6.17. These show the output *a_out* lagging the input signal *a0_in* by approximately six nanoseconds after the clock for the first half of the simulation period, followed by the output becoming the input signal *a2_in* for the remainder of the simulation.

With the data flowing in the opposite direction (nominally a data write), the unselected outputs are blocked from the actual output, and the data flows from the 'output' side (eg. *a_out*) to the 'input' side (eg. *a0_in*). This case was simulated for data applied to pins *a_out* to *h_out* and the select signal *a_1* varied from 0 volts to 5 volts half way through the simulation. The resulting simulation plot is shown in Figure 6.18. The pins that are not used as outputs (*a1_in, a3_in, b1_in*, etc) are all at approximately two and a half volts, although of course they can be pulled to whatever logic level is required by another Bus Exchanger driving them. The signals that are driven (*a0_in, a2_in, b0_in*, etc.) follow the driving signal (*a_out, b_out*, etc) only with the correct select signal.

If the Bus Exchanger is run in 'transparent' mode (instead of 'latched' mode), then the simulations shown in Figures 6.19 to 6.20 are produced. These are the equivalent simulations to those in Figures 6.17 to 6.18 respectively, except for the change in latch/transparent mode. These plots show the outputs of the Bus Exchangers following the inputs by approximately three

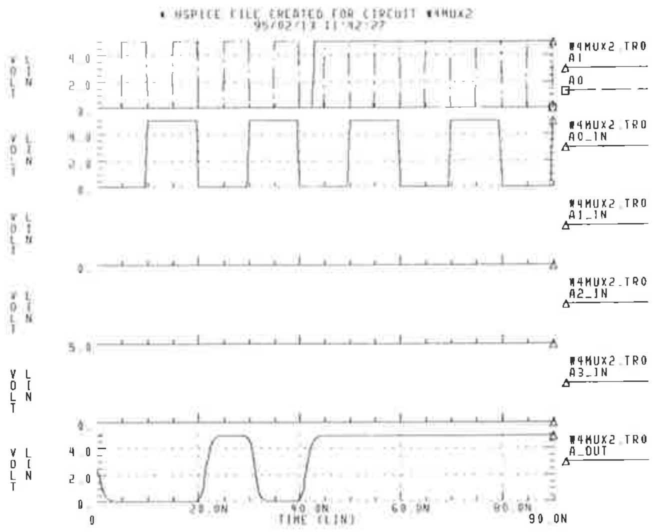


Figure 6.17: **Latched Read With Varying Select Signals**

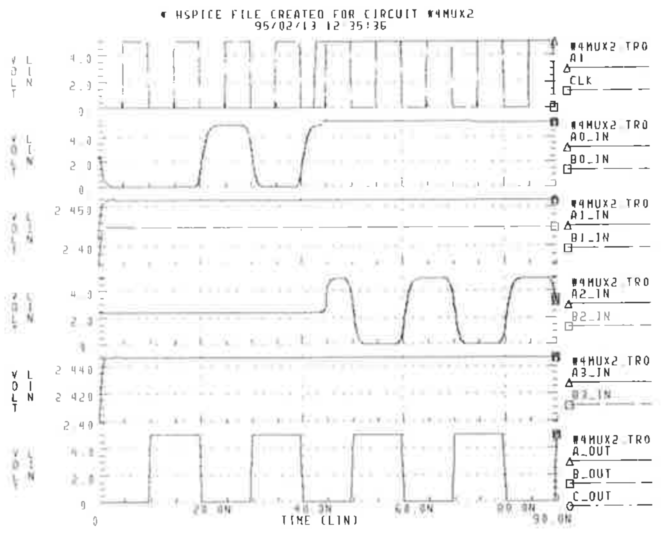


Figure 6.18: **Latched Write With Varying Select Signals**

to four nanoseconds in either direction, and independently of the clock signal.

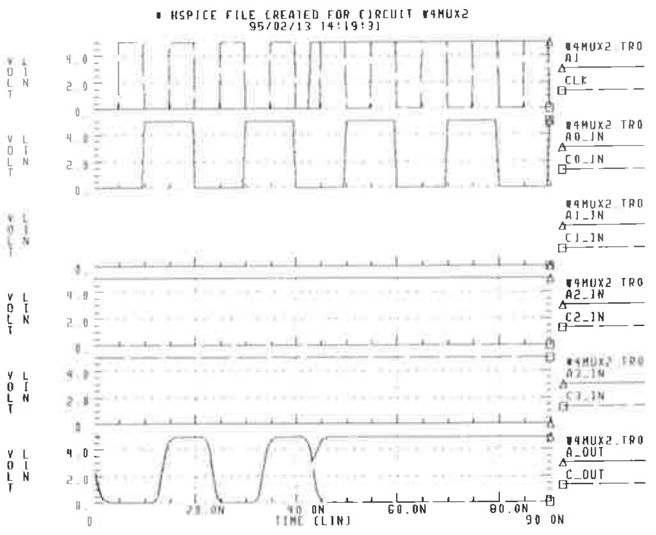


Figure 6.19: **Transparent Read With Varying Select Signals**

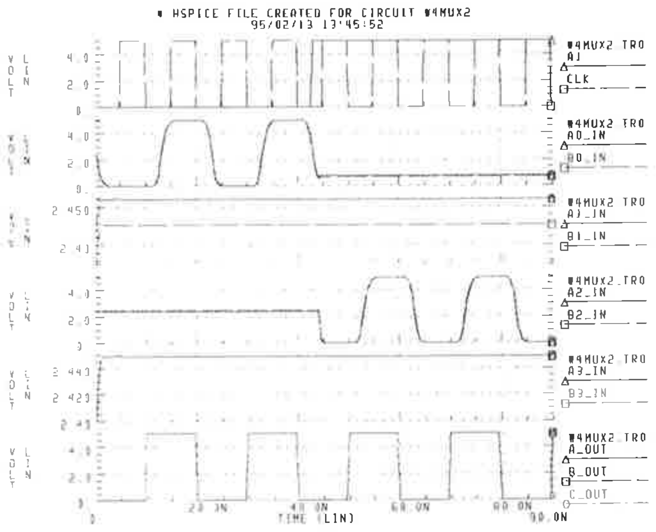


Figure 6.20: **Transparent Write With Varying Select Signals**

6.7 Loading Double-length Words

If double precision (64-bit) arithmetic is required, either four cells in the processing array can be merged or data can be cycled through a single cell four times. In either case, the required bandwidth in terms of the number of operands for the processing array will be one quarter of that when single precision (32-bit) data is being processed. As each operand is fetched with a single address, and a double precision operand is twice the size of a single precision operand, the address generator will operate at one quarter of its single precision rate, and one half of the bandwidth in terms of bytes will be required for double precision.

For a two-way interleaved memory system when fetching 32-bit operands, the memory system need no longer be interleaved when fetching 64-bit operands. This will reduce the complexity of the 64-bit operation mode, and remove the possibility of bank conflicts that may occur for an interleaved memory system.

Chapter 7

Performance Estimates

The following performance estimates are based on the memory structures described in Chapter 6. Some performance figures estimate the operational speed of a 'cut-down' system, typically one without the main dynamic memory, and hence no cache tag checking. Unless otherwise stated, it is assumed that the array used has a variable bandwidth, ie, the processing elements are all the same speed. The array is assumed to be of size $p \times p$, with C MWords of cache memory and M MWords of main memory. The bandwidth of the cache to main memory is B MWords per second, with τ ns per load, and a latency for a cache refill of L cycles ($= L\tau$).

7.1 Matrix Multiplication

For a matrix multiplication running directly from the cache SRAM, the most fundamental calculation is based on a partitioned matrix product with no delays due to memory. If the two matrices, **A** & **B**, that are to be multiplied together are of dimensions $R \times S$ and $S \times T$ respectively, then the final product matrix will be of size $R \times T$, divided into $\lceil \frac{R}{p} \rceil \times \lceil \frac{T}{p} \rceil$ partitions, each of size $p \times p$. The input matrices will be divided into $\lceil \frac{R}{p} \rceil$ & $\lceil \frac{T}{p} \rceil$ partitions respectively, of size $p \times S$ & $S \times p$. The time to perform the product is simply calculated by

$$T_{simp.\_mult} = \left\lceil \frac{R}{p} \right\rceil \times \left\lceil \frac{T}{p} \right\rceil \times p \times S \times \tau \tag{7.1}$$

The floating point performance in terms of Mflops (Million Floating Point Operations per Second) is simply the number of floating point operations done divided by the time taken as given in Equation 7.1.

$$Perf_{simp.\_mult} = \frac{2 \times R \times S \times T}{\left\lceil \frac{R}{p} \right\rceil \times \left\lceil \frac{T}{p} \right\rceil \times p \times S \times \tau} \tag{7.2}$$

The performance plot for this simple estimate is shown in Figure 7.1, below. In this plot, the array size is assumed to be fixed at 1024 processing elements (32×32, $p = 32$), the load/store time per word (τ) is ten nanoseconds, and the matrices multiplied together are assumed to be square ($R = S = T$).

For comparison purposes, significant benefits are obtained if either the array size is increased or the memory load/store time is reduced. Figure 7.2 shows the performance if the array dimension is doubled ($p = 64$), and Figure 7.2b) is the performance if the load/store time is halved ($\tau = 5$ns), but with the original array width of 32. Figure 7.3 is the performance if both the array size is doubled and the load/store time is halved (constant bandwidth).

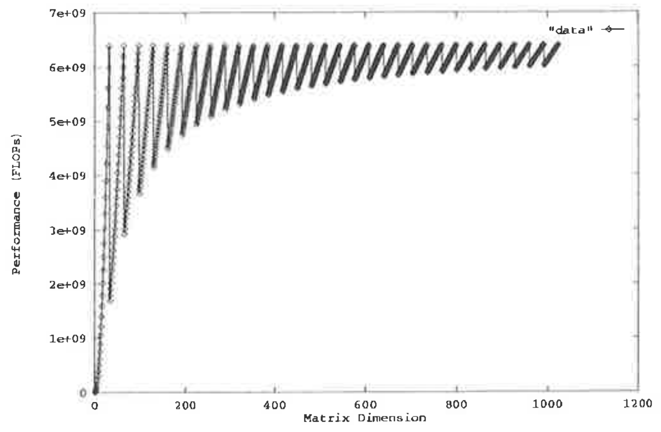
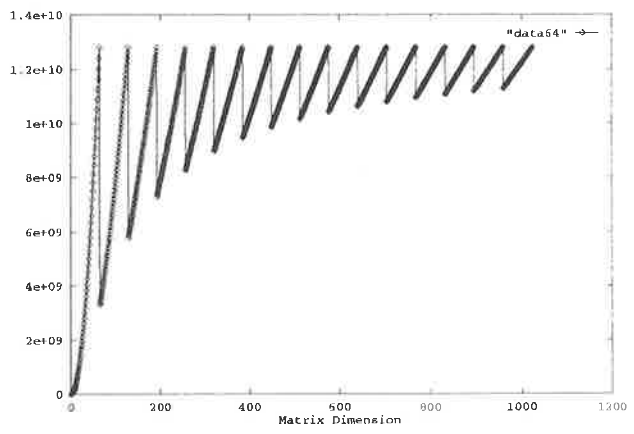


Figure 7.1: **Performance Figures for Simple Multiplication Model (MFlops)**



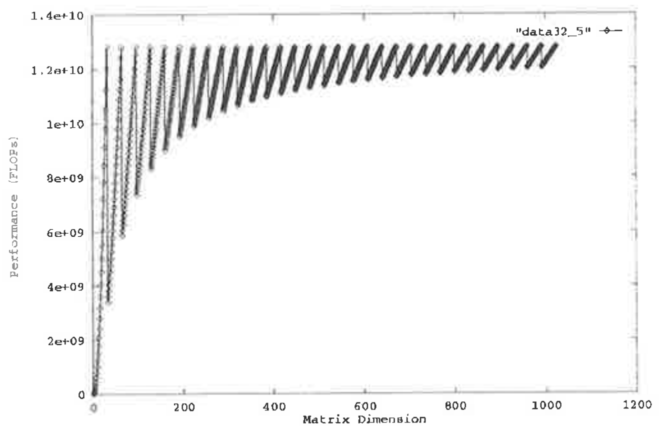


Figure 7.2: **Performance for a)Array Dimension = 64 b) Load/Store Time = 5ns (MFlops)**

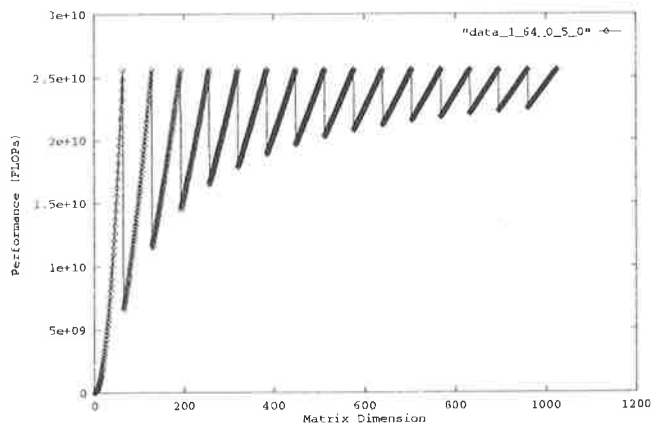


Figure 7.3: **Performance for Array Dimension = 64 and Load/Store Time = 5ns (MFlops)**

The significance of Figure 7.3 is that it shows the squared nature of the performance of the array for a constant bandwidth. This is an example of the concepts of address bandwidth that lead to Equation 5.4.

If the data must be loaded from main memory, then cache refill time must be included in the equations. There are two cases to be considered for a cache refill, the cases with and without data streaming. As described in Chapter 6, data streaming is when data loaded from main memory is available directly to the processor simultaneously to the data being written into the cache. Without data streaming, the complete data block must first be written into cache memory and then the desired data read from cache memory. If it is assumed that the cache is of size C MWords and a cache block is of size D words, then for a direct mapped cache in which the data is stored in the same order that it is required (ie in partitioned form), then a few assumptions can be made that will provide a simple estimate for system performance.

It is easiest to break the time required for a calculation into two categories, those when there is an integral number of array partitions in a matrix ($R \bmod p = 0$) and those when a full array boundary is not filled ($R \bmod p \neq 0$). Consider reading data into the array for the case of an integral number of partitions ($R \bmod p = 0$). Then the time taken to load all the data into the cache will be the latency plus the fill period, ie

$$T_{\text{Load}} \approx R^2 \frac{L}{D} + R^2 \tau$$

The data that is loaded into the cache will be used immediately, so it will need to be retrieved from cache memory. This will take another $R^2\tau$ cycles, after which the diagonal blocks of the solution (C_{ii}) will have been calculated. This will leave $\left(\frac{R}{p} - 1\right)\frac{R}{p}$ partitions to be calculated, each requiring approximately Rp cache loads. Therefore, the time required for computation after the data has been loaded into cache will be

$$\begin{aligned} T_{\text{Comp}} &\approx R^2\tau + \left(\frac{R}{p} - 1\right)\frac{R}{p} Rp\tau \\ &\approx R^2\tau + \left(\frac{R}{p} - 1\right) R^2\tau \end{aligned}$$

which totals

$$T_{Main\_Mem} = \frac{R^2}{D}(L + 2 \times D \times \tau) + R^2\left(\frac{R}{p} - 1\right)\tau \tag{7.3}$$

The second case to consider is for $R \bmod p \neq 0$. Then there will be $\left\lfloor \frac{R}{p} \right\rfloor$ full partitions, which will take the same load and diagonal computation time as before, ie

$$T_{\text{Load}} \approx \left\lfloor \frac{R}{p} \right\rfloor \frac{Rp}{D}(L + 2 \times D \times \tau) + pR\left\lfloor \frac{R}{p} \right\rfloor\left(\left\lfloor \frac{R}{p} \right\rfloor - 1\right)\tau$$

Added to this is the time it takes to compute the unfilled edges of the matrix. The edge width will be the total width minus the width of all the filled partitions, ie $R - p\left\lfloor \frac{R}{p} \right\rfloor$. The size of the data contained in this portion of the matrix will be $R\left(R - p\left\lfloor \frac{R}{p} \right\rfloor\right)$, which must be loaded and multiplied by all partitions (filled and unfilled). As there will be edges at two sides, there will be a factor of two in the computation times. The extra time is

$$T_{\text{edge}} \approx R\left(\frac{R - p\left\lfloor \frac{R}{p} \right\rfloor}{D}\right)(L + D \times \tau) + 2R \times \left\lceil \frac{R}{p} \right\rceil \times p \times \tau$$

These times are summarised in Equation 7.4.

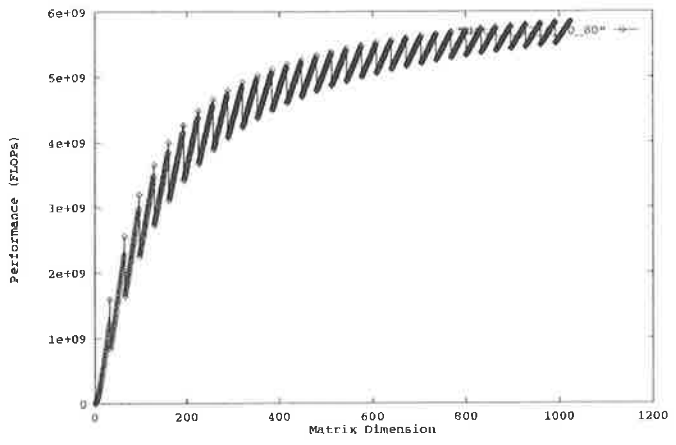
$$T_{Main\_Mem} \approx \begin{cases} \frac{R^2}{D}(L + 2 \times D \times \tau) + R^2\left(\frac{R}{p} - 1\right)\tau & \text{if } R \bmod p = 0 \\ \left\lfloor \frac{R}{p} \right\rfloor \frac{Rp}{D}(L + 2 \times D \times \tau) + pR\left\lfloor \frac{R}{p} \right\rfloor\left(\left\lfloor \frac{R}{p} \right\rfloor - 1\right)\tau + \\ R\left(\frac{R - p\left\lfloor \frac{R}{p} \right\rfloor}{D}\right)(L + D \times \tau) + 2R \times \left\lceil \frac{R}{p} \right\rceil \times p \times \tau & \text{otherwise} \end{cases} \tag{7.4}$$

So the performance of the system without data streaming is

$$Perf_{Main\_Mem} \approx \begin{cases} \frac{2 \times R^3}{\frac{R^2}{D}(L+2 \times D \times \tau) + R^2\left(\frac{R}{p}-1\right)\tau} & \text{if } R \bmod p = 0 \\ \frac{2 \times R^3}{\left\lfloor \frac{R}{p} \right\rfloor \frac{Rp}{D}(L+2 \times D \times \tau) + pR\left\lfloor \frac{R}{p} \right\rfloor\left(\left\lfloor \frac{R}{p} \right\rfloor - 1\right)\tau + R\left(\frac{R-p\left\lfloor \frac{R}{p} \right\rfloor}{D}\right)(L + D \times \tau) + 2R \times \left\lceil \frac{R}{p} \right\rceil \times p \times \tau} & \text{otherwise} \end{cases} \tag{7.5}$$

For a fixed array size of 1024 processing elements arranged as (32 × 32) with a constant bandwidth of 100 MWords/sec, then increasing the block size from 4 to 512 results in an improvement in the overall speed of the system as the block size increases, especially for matrices of small order. However, there is improvement in performance as the block size increases above sixteen is small. Figures 7.4 a) through d) show the performance of the processor as the block size is increased from 4 to 16 to 64 to 512. The other very noticeable feature is the performance advantage obtained when the matrix can be divided into an integral number of block of the same width as the processing array. The latency for a cache miss is assumed to be 80 nanoseconds.

If streaming is added to the functionality of the processor, then data can be used immediately it is available from main memory, and a separate write to cache phase is not required. In effect, this removes a factor of $D \times \tau$ from the memory load part of the time equations, which will reduce the effects of memory latency (the penalty will be mitigated across a greater number of loads). Then Equations 7.4 and 7.5 becomeEquations 7.6 and 7.7 respectively.



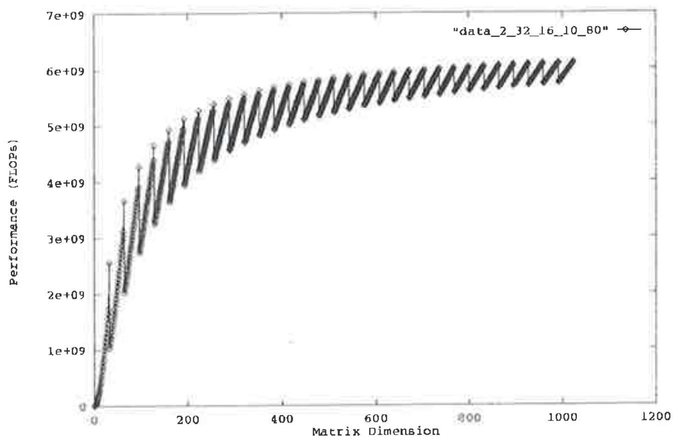
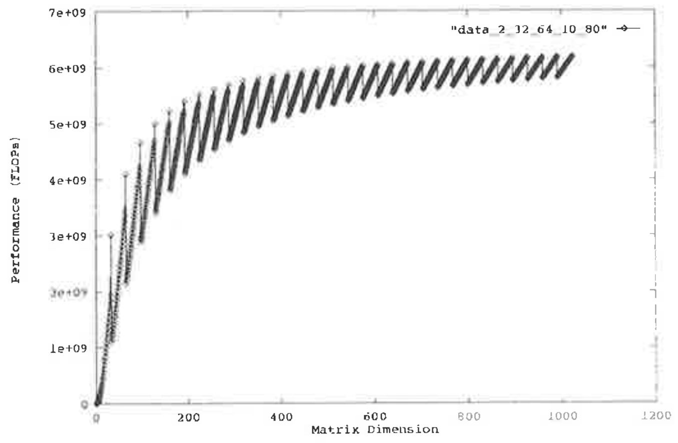


Figure 7.4: **a) Block Size = 4** **b) Block Size = 16**



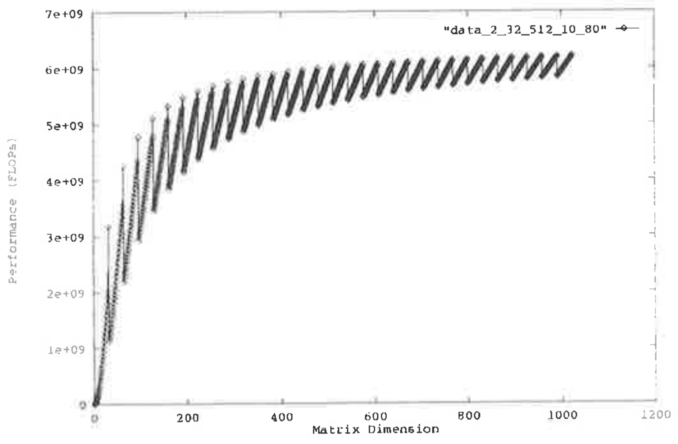


Figure 7.4: **c) Block Size = 64** **d) Block Size = 512**

Figure 7.4: **Performance Estimates for Matrix Multiplication Without Streaming**

$$T_{Main\_Mem} \approx \begin{cases} \frac{R^2}{D}(L + D \times \tau) + R^2\left(\frac{R}{p} - 1\right)\tau & \text{if } R \bmod p = 0 \\ \left\lfloor\frac{R}{p}\right\rfloor \frac{Rp}{D}(L + D \times \tau) + pR\left\lfloor\frac{R}{p}\right\rfloor\left(\left\lfloor\frac{R}{p}\right\rfloor - 1\right)\tau + R\left(\frac{R - p\lfloor\frac{R}{p}\rfloor}{D}\right)L + 2R \times \lceil\frac{R}{p}\rceil \times p \times \tau & \text{otherwise} \end{cases} \tag{7.6}$$

$$Perf_{Main\_Mem} \approx \begin{cases} \frac{2 \times R^3}{\frac{R^2}{D}(L+D\times\tau)+R^2\left(\frac{R}{p}-1\right)\tau} & \text{if } R \bmod p = 0 \\ \frac{2 \times R^3}{\left\lfloor\frac{R}{p}\right\rfloor \frac{Rp}{D}(L+D\times\tau)+pR\left\lfloor\frac{R}{p}\right\rfloor\left(\left\lfloor\frac{R}{p}\right\rfloor-1\right)\tau+R\left(\frac{R-p\lfloor\frac{R}{p}\rfloor}{D}\right)L+2R\times\lceil\frac{R}{p}\rceil\times p\times\tau} & \text{otherwise} \end{cases} \tag{7.7}$$

The plots for the performance with streaming included are shown in Figures 7.5 a) through d). These show the improvement in performance with increasing block size, with significant improvements up to a block size of 64, and with more improvement as block size increases further. This performance improvement relies heavily on proper placement of data in memory to achieve its full promise. However, if the data is located poorly in memory, the streamed version will do no worse than the unstreamed version.

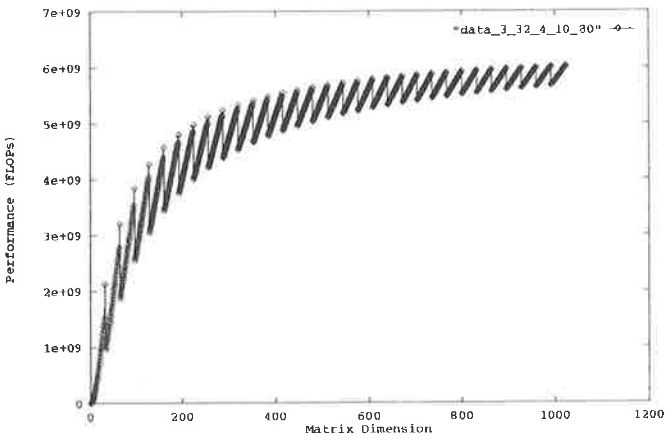
7.2 Gauss Jordan Elimination and Inversion

As mentioned previously in Section 3.4, one of the advantages of Gauss-Jordan elimination is that a matrix inversion uses *exactly* the same routine, with the column range extended to an augmented matrix including the identity matrix, **I**. The important trick to achieving maximum performance for Gauss-Jordan elimination is to very carefully partition the algorithm to fit both the array size and the cache memory size. If the inverse is sought, the identity matrix need not be explicitly included, as the columns of the identity matrix can be included as the inverse grows and the input matrix reduces. The performance estimates here assume that the data controller can stream data and that the data is stored in block rows in main memory.

7.2.1 Gaussian Elimination for Matrices Smaller than the Cache Size

If the input matrix (the one that is to be reduced using Gauss Jordan elimination) fits entirely into the cache, the only load from main memory (cache refill) will be the initial load. The procedure given in Section 3.4.1 can be implemented directly, with no blocking due to cache size.

The first step is to determine the inverse of the pivot block $\mathbf{A}_{11}$. The block must be loaded into cache from main memory and then operated on to produce the block inverse. Using the block inversion algorithm from Section 3.4.1, five iterations are required to produce an inverse, with each interation requiring three passes through the processing array, for a total of fifteen passes through the array. Added to this will be a factor due to the final scalar division of the initial estimate being completed entirely after all the data in the row has been fetched, the penultimate scalar division will commence with one load still to occur, etc. The time to perform



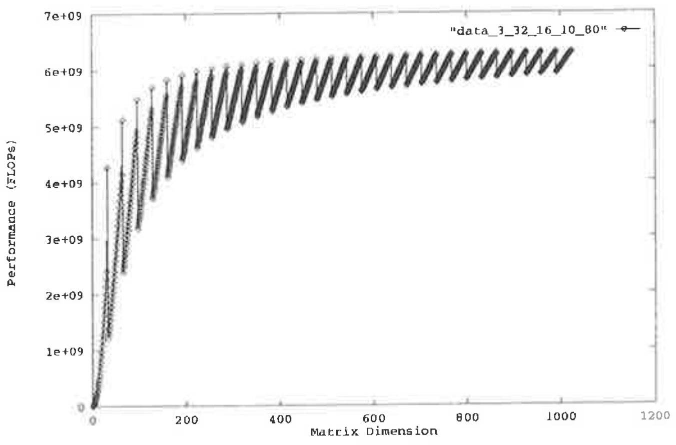
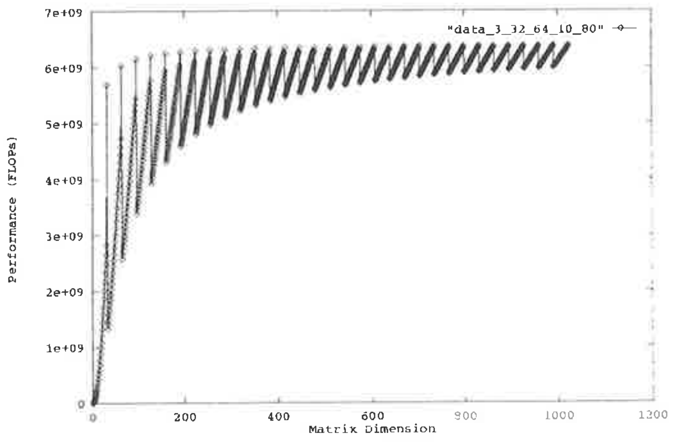


Figure 7.5: **a) Block Size = 4**

b) Block Size = 16



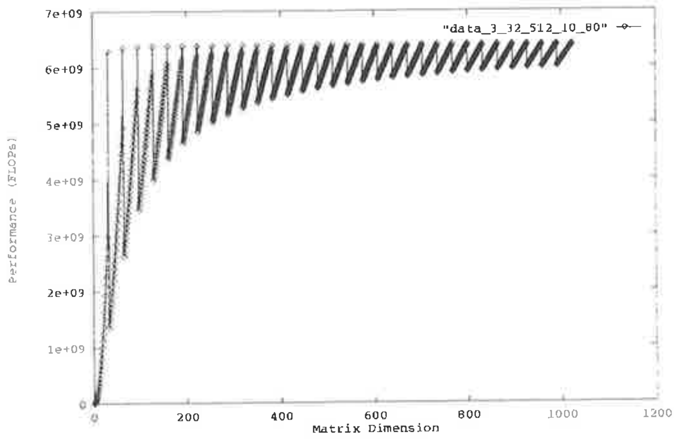


Figure 7.5: **c) Block Size = 64**

d) Block Size = 512

Figure 7.5: **Performance Estimates for Matrix Multiplication With Streaming**

the first block inversion will also include the time to load the block from main memory, and can be characterised by

$$T_{1^{st}pivot} \approx 15p^2\tau + \frac{T^2_{scalar\ divide}}{2}\tau + \begin{cases} \left\lceil \frac{R^2}{D} \right\rceil \times L & \text{if } R < p \\ \left\lceil \frac{p^2}{D} \right\rceil \times L & \text{otherwise} \end{cases} \tag{7.8}$$

where $T_{scalar\ divide}$ is the time it takes to perform a scalar inversion operation, and is in terms of cycles. Once the first iteration of Gauss-Jordan elimination has completed, then all the data will be in the data caches, so the time taken for any successive pivot inversions is simply:

$$T_{succ.\ pivot} \approx 15p^2\tau + \frac{T^2_{scalar\ divide}}{2}\tau \tag{7.9}$$

If the matrix to be solved is smaller than the size of the processing array, then the pivot inversion stage is all that is required. In general, this will not be the case, and the pivot row will need to be normalised and all other rows updated. As a column oriented scheme will make memory accesses simpler (Section 3.4.3), then the row normalisation and the matrix update can be performed together. Bearing in mind that there for $(n-1)$ updates, the 'k^{th}' of which is of length $(n-k)$, then the average length of an update will be $\frac{n(n-1)}{2}$, and so the time taken for the complete Gauss-Jordan elimination will be

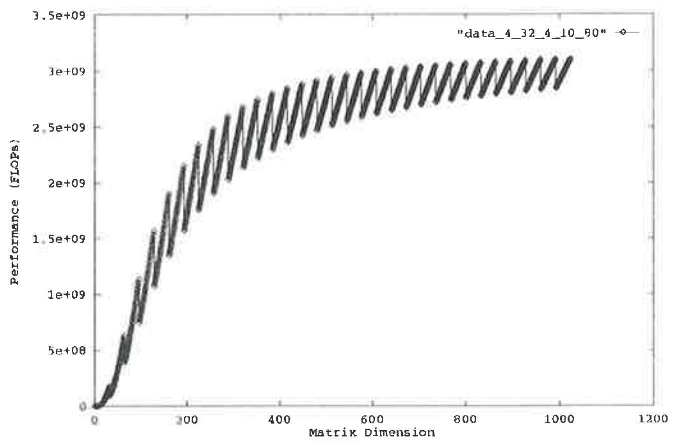
$$T_{Gauss\text{-}Jordan} \approx \begin{cases} \left\lceil \frac{R^2}{D} \right\rceil \times L + 15p^2\tau + \frac{T^2_{scalar\ divide}}{2}\tau & \text{if } R < p \\ \left(\left\lceil \frac{p^2}{D} \right\rceil + \left\lceil \frac{R(R-p)}{D} \right\rceil \right) \times L + \left(33p^2 + T^2_{scalar\ divide}\right) \times \tau & \text{if } p \le R < 2p \\ \left(\left\lceil \frac{p^2}{D} \right\rceil + \left\lceil \frac{R(R-p)}{D} \right\rceil \right) \times L + \left(15p^2 + \frac{T^2_{scalar\ divide}}{2}\right) \times \tau + \\ \left(2\left\lceil \frac{R}{p} \right\rceil - 1\right)\left(\left\lceil \frac{R-p}{p} \right\rceil + \frac{1}{2}\left(\left\lceil \frac{R}{p} \right\rceil - 1\right)\left(\left\lceil \frac{R}{p} \right\rceil - 2\right)\right) \times p^2\tau + \\ \left(15p^2\tau + \frac{T^2_{scalar\ divide}}{2}\right)\left\lceil \frac{R-p}{p} \right\rceil & \text{otherwise} \end{cases} \tag{7.10}$$

The number of *required* operations is approximately $R^3 - 1.5R^2 + 0.5R$, so the performance of the array in MegaFLOPs when performing Gauss-Jordan elimination is

$$Perf_{\mathrm{GJsolve}} \approx \frac{R^3 - 1.5R^2 + 0.5R}{T_{Gauss\text{-}Jordan}} \tag{7.11}$$

Figure 7.6 shows the performance estimates for solving a matrix of size less than 1025 rows for various cache block sizes.

If the inverse of the matrix is required, then the same procedure is used, except that the updated matrix does not reduce in size. Equation 7.10 can be used, except that the factor $\frac{1}{2}\left(\left\lceil \frac{R}{p} \right\rceil - 1\right)\left(\left\lceil \frac{R}{p} \right\rceil - 2\right)$ in the second to bottom line becomes $\left\lceil \frac{R}{p} \right\rceil \left(\left\lceil \frac{R}{p} \right\rceil - 1\right)$, and a factor of $p^2\tau$ is added to the case of $R \le 2p$. Then an estimate of the time taken to perform a matrix



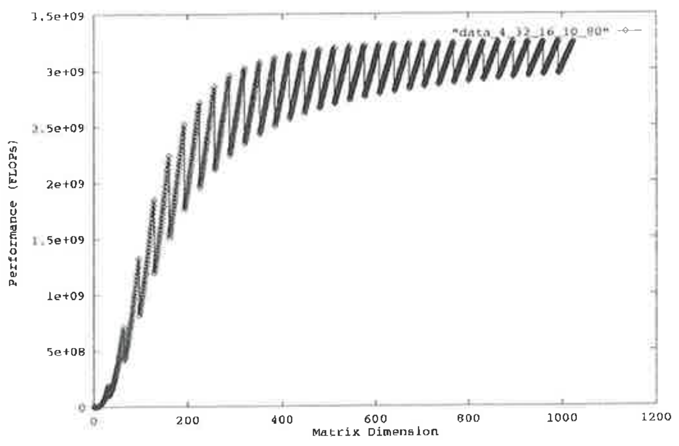
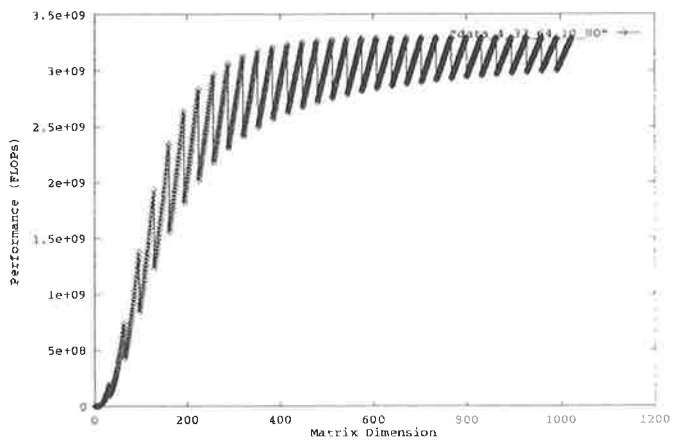


Figure 7.6: **a) Block Size = 4** **b) Block Size = 16**



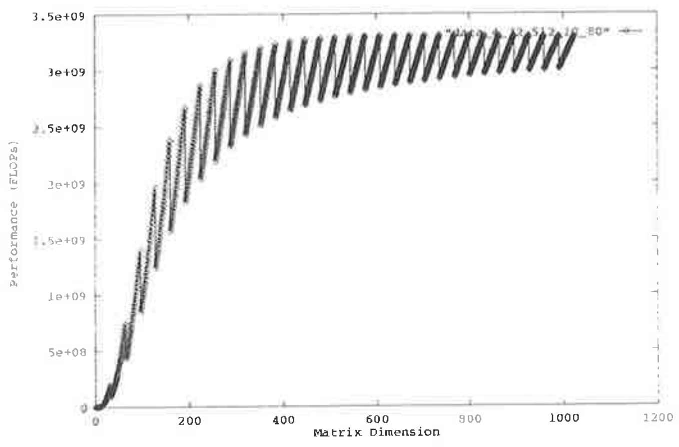


Figure 7.6: **c) Block Size = 64** **d) Block Size = 512**

Figure 7.6: **Performance for Gauss-Jordan Elimination on Matrices that Fit Into Cache**

inversion is

$$T_{Invert} \approx \begin{cases} \left\lceil \frac{R^2}{D} \right\rceil \times L + 15p^2\tau + \frac{T^2_{scalar\ divide}}{2}\tau & \text{if } R < p \\ \left(\left\lceil \frac{p^2}{D} \right\rceil + \left\lceil \frac{R(R-p)}{D} \right\rceil \right) \times L + \left(34p^2 + T^2_{scalar\ divide} \right) \times \tau & \text{if } p \le R < 2p \\ \left(\left\lceil \frac{p^2}{D} \right\rceil + \left\lceil \frac{R(R-p)}{D} \right\rceil \right) \times L + \left(15p^2 + \frac{T^2_{scalar\ divide}}{2} \right) \times \tau + \\ \left(2 \left\lceil \frac{R}{p} \right\rceil - 1 \right) \left(\left\lceil \frac{R-p}{p} \right\rceil + \left\lceil \frac{R}{p} \right\rceil \left(\left\lceil \frac{R}{p} \right\rceil - 1 \right) \right) \times p^2\tau + \\ \left(15p^2\tau + \frac{T^2_{scalar\ divide}}{2} \right) \left\lceil \frac{R-p}{p} \right\rceil & \text{otherwise} \end{cases} \tag{7.12}$$

Note that this is actually slightly conservative, as a full update is not required for the added rows. The given matrix is augmented a column at a time by a column of the identity matrix, which for the 'i^{th}' update is the column vector $(0, 0, \ldots, 0, 1, 0, \ldots, 0, 0)^T$, with the '1' in the i^{th} position. Therefore, rather that requiring two multiply-accumulate pairs for an update (one to load the array), and one multiply-accumulate pair for the normalise, the result of the normalise is the inverse of the pivot, which is already calculated, and the update is only a single multiply-accumulate pair. However, as the order of this saving is small, the difference on the overall performance estimate is small.

The number of required operations to invert a matrix using Gauss-Jordan elimination is $2R^3 - 4R^2 + 5R + 1$, and so the performance of the array when inverting a matrix is

$$Perf_{Invert} \approx \frac{2R^3 - 4R^2 + 5R + 1}{T_{Invert}} \tag{7.13}$$

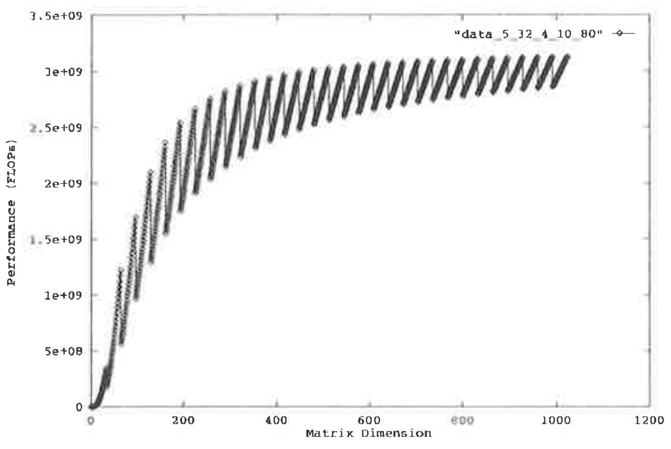
Figure 7.7 shows the performance estimates for inverting a matrix of size less than 1025 rows for various cache block sizes.

7.2.2 Gaussian Elimination for Matrices Larger than the Cache Size

If the matrix to be solved is too large to fit into the cache, then the matrix can be partitioned into equal sized blocks that do fit into the cache. Ideally, the blocks will be the same size as the cache. Thus, for a cache size of one Megaword,

- a matrix of size 1500×1500 would be partitioned into four blocks each of size 750×750
- a matrix of size 2000×2000 would be partitioned into four blocks each of size 1000×1000
- a matrix of size 3000×3000 would be partitioned into nine blocks each of size 1000×1000
- a matrix of size 10000×10000 would be partitioned into one hundred blocks each of size 1000×1000

The procedure is very simple, and is just an extension of the block Gauss-Jordan algorithm given in Section 3.4.1. The 'pivot' block is inverted using the algorithm described above in Section 7.2.1. The block thus inverted is multiplied by all the other blocks is the same row, with the performance estimated by Equation 7.7. All other blocks are updated as for a standard Gauss-Jordan elimination method, and then the procedure repeats. To estimate the performance of such a method, use the estimates in Equations 7.12 & 7.7. For a cache size of 'C' words, let



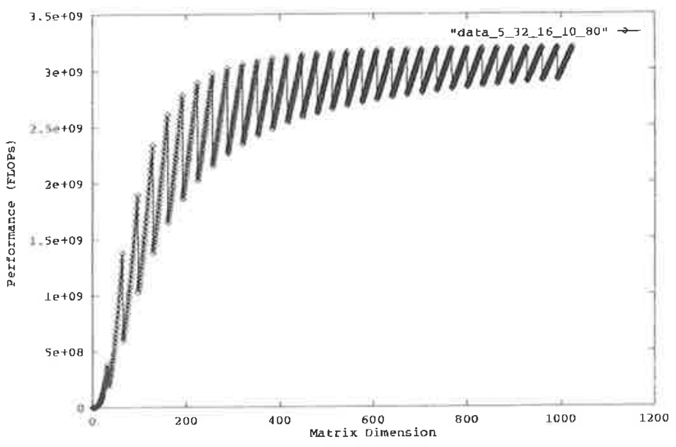
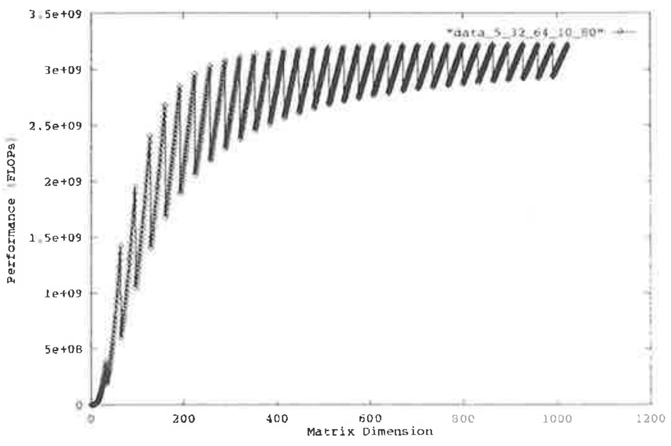


Figure 7.7: **a) Block Size = 4** **b) Block Size = 16**



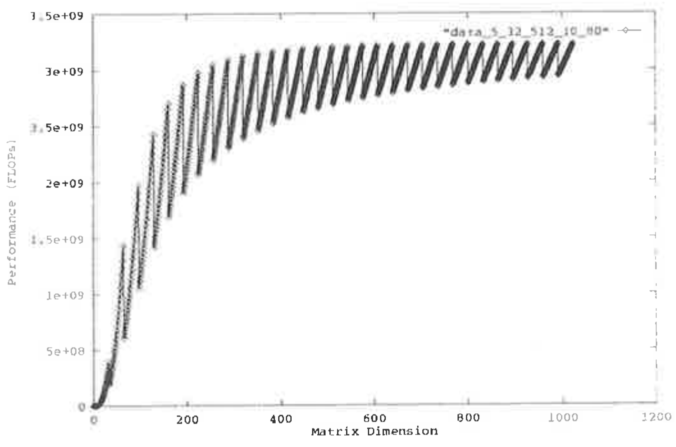


Figure 7.7: **c) Block Size = 64** **d) Block Size = 512**

Figure 7.7: **Performance for Gauss-Jordan Inversion on Matrices that Fit Into Cache**

$f = \sqrt{C}$, so the block dimension will be the largest number that is less than f while still being an integral factor of the full matrix dimension. Let the block dimension be denoted by ρ, and let there be ξ^2 blocks of size $\rho \times \rho$ in the matrix (ie $R = \xi\rho$).

$$T_{Large\ GJ} \approx \xi T_{Invert} + (2\xi - 1)(\xi - 1)\left(\frac{\xi}{2}\right) T_{Mult_{\rho\rho}} \tag{7.14}$$

The number of operations is again $R^3 - 1.5R^2 + 0.5R$, and so the performance of the system when solving a matrix that is larger than the cache size is

$$Perf_{Large\ GJ} \approx \frac{R^3 - 1.5R^2 + 0.5R}{T_{Large\ GJ}} \tag{7.15}$$

$$\approx \frac{R^3 - 1.5R^2 + 0.5R}{\xi T_{Invert} + (2\xi - 1)(\xi - 1)\left(\frac{\xi}{2}\right) T_{Mult_{\rho\rho}}} \tag{7.16}$$

Equation 7.16 is plotted for a range of block sizes in Figure 7.8. An interesting point to notice is that the performance actually peaks for a matrix a few times larger than the cache size, before declining to the previous Gauss-Jordan asymptote. The reason for this peak is linked to the performance difference between matrix multiplication and matrix inversion. As the method of adding two matrices is to load one matrix into the array (a multiply-accumulate pair with accumulator reset) and then multiply-accumulate the other matrix without resetting, Gauss-Jordan elimination can be expected to run at approximately half the speed of an equivalent matrix multiplication. However, the normalisation phase does not require an addition, so operates at multiplication speed. Therefore, if the normalisation phase is of a similar order to the update phase, the processing array will be running at the speed of a matrix multiplication for a significant period. As the update phase is of an order that is the square of the order of the normalisation phase, the peak speed will be when there are two to four blocks in a dimension. This is shown clearly in the plots.

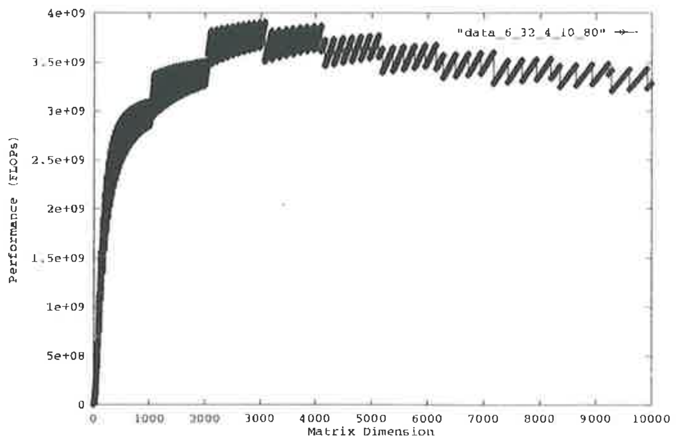
7.2.3 Conclusion

The plots given in this section show that the matrix processor configured with modest dimensions (~ 32) and address generation times ($\sim 10ns$) is capable of speeds in excess of three GigaFLOPs for Gauss-Jordan elimination and inversion. As the plots in Figure 7.8 show, the size of a problem that can be solved at this speed is largely limited not by the processing array size or rate, but by the memory size.

An interesting point of reference is the Linpack1000 and Linpack10000 benchmarks [23]. Versions of these benchmarks can be summarised as the number of floating point operations *required* divided by the time taken to perform a 1000×1000 and $10,000 \times 10,000$ matrix reduction using any method. The important point about the *required* operations is that any extra operations are not included in the operations count. As this is the case for the plots given in this chapter, the Linpack1000 and Linpack10000 benchmarks will have ratings in excess of three GigaFLOPs. It is interesting to note that it would take less than one second to solve a set of 1000 linear equations, while a set of 10,000 linear equations would be solved in approximately six minutes.

7.3 Discrete Fourier Transform

The procedure outlined in Section 3.5 leads to some very simple estimates for the performance of an embedded array both with and without a multi-level memory system. In general, an



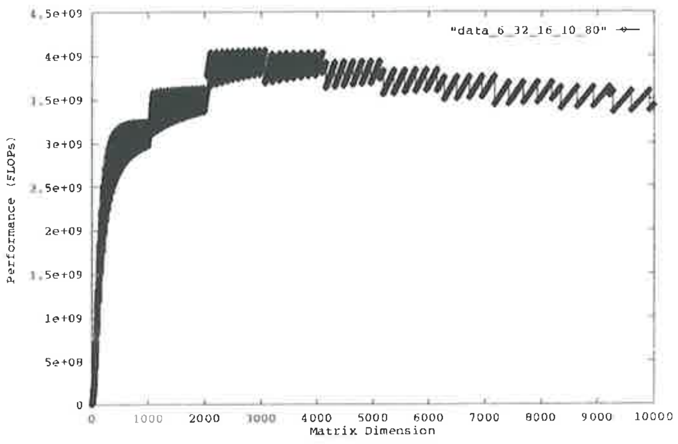
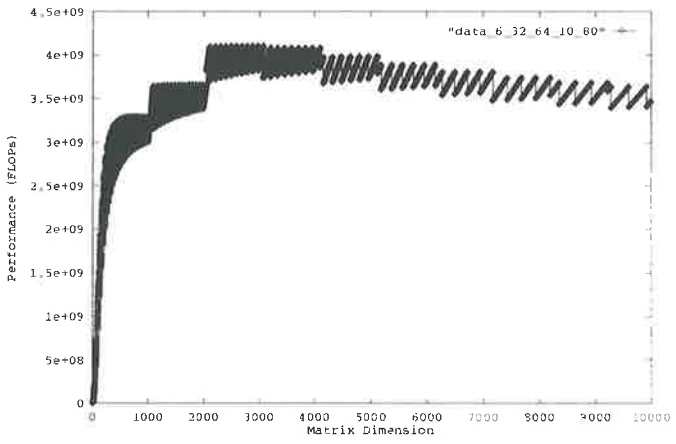


Figure 7.8: **a) Block Size = 4** **b) Block Size = 16**



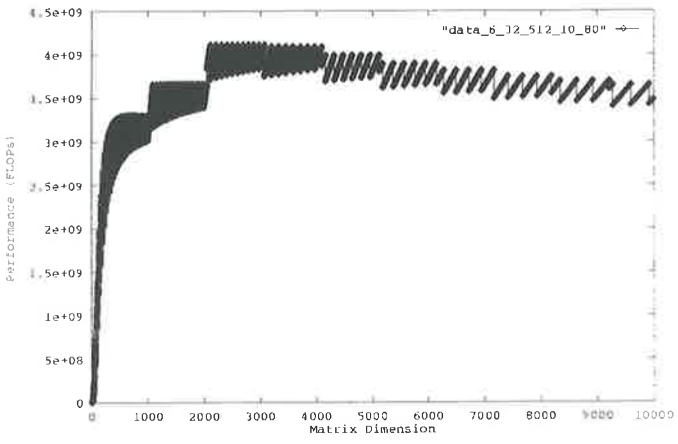


Figure 7.8: **c) Block Size = 64** **d) Block Size = 512**

Figure 7.8: **Performance for Gauss-Jordan Inversion on Matrices that Do Not Fit Into Cache**

embedded system will not have a multi-level memory, although this may be necessary if very large size DFTs are required.

The performance of a two dimensional DFT is simply the number of floating point operations divided by the time for two multiply-accumulate operations and one unload operation. The inclusion of the unload is due to the fact that the second matrix product is dependent on the result of the first. However, if the size of the matrices is larger than the size of the processing array, the unloading of each partition can be 'hidden' within successive partition products, and the calculation time becomes the time for the two matrix products.

If a higher $n-$dimensional DFT is performed, then the unload operation is again hidden by successive matrix products, as all premultiplications can be completed before the postmultiplications are started. Therefore if the size of the first dimension to be calculated is N_1, then the time to complete the first dimensional computation will be

$$2N_1 \times T_{matrix\ product} + T_{unload}$$

Here, the unload time is due to the very final unload before the output bank can be swapped back. If the dual ported RAMs in the feedback path are available, then the temporary output data can be returned to the input without any bank swapping, and no unload delay is required. The time to complete each dimension is then:

$$T_{per\ dimension} = 2N_i \times T_{matrix\ product} \tag{7.17}$$

As the length of a transform is the product of the n dimension lengths which are mutually prime, the time taken for only selected transform lengths will be estimated, and the performance in *equivalent* MegaFLOPs provided. The equivalent MegaFLOP rating is calculated by dividing the number of necessary operations as required for a standard FFT algorithm by the time taken to perform the calculation. The text by Press *et. al.* [73] was used to determine an approximate operations count, which was found to be approximately $6.5N \log N$ for a transform of length N. The processing array is assumed to be square, of dimension $p \times p$. The times are shown in terms of the cycle time τ_c, which is the time is takes to load a single data element into the array from memory, or from the cache in the case of a multi-level memory system. Note that the Fourier coefficients will not need to be reloaded into cache memory in a multi-level memory system as the input data must be, so the load time will always be dependent on the input data and not on the Fourier coefficients.

7.3.1 Discrete Fourier Transform Without Multi-level Memory System

If only a single level memory system is used, the procedure and estimates are very simple. It can be assumed that the four banks of memory are still used, but there is not cache tagging to be checked. The Fourier coefficients reside in one of the memory banks, while the input data is deposited into another of the banks. A third bank is needed for temporary storage during processing and for storing the output data, while the input data for the next iteration of the DFT can be loaded into the fourth bank in preparation.

Two Dimensional Transform

If the length of the transform is less than p^2, then the DFT can be completed in one iteration. All that is required is a matrix product, an unload, and then another matrix product. Due to the constant bandwidth property of the array, the time to perform the two dimensional transform is approximately

$$T_{2D} \approx 3 \times p^2 \times \tau_C \tag{7.18}$$

if no consideration is made for matrix products on matrices that are smaller than the processing array. If such consideration is made, the time reduces to

$$T'_{2D} \approx (2 \times N_1 + N_2) \times p \times \tau_C \tag{7.19}$$

where N_1 and N_2 are the two dimensions of the transform, and $N1 > N2$.

A selection of two dimensional transform lengths and their computational time in cycles is given in Table 7.1 for dimension lengths smaller than the processing array size, which is assumed to be of size 32×32 processing elements. The performance in MegaFLOPs is also given, assuming a cycle time of ten nanoseconds.

Transform length	N_1	N_2	Time (τ_C)	Performance (MFLOPs)
899	31	29	2912	1969
837	31	27	2848	1855
775	31	25	2784	1737
713	31	23	2720	1615
651	31	21	2656	1489
589	31	19	2592	1359
783	29	27	2720	1798
725	29	25	2656	1685
667	29	23	2592	1569
609	29	21	2528	1448
551	29	19	2464	1323
675	27	25	2528	1631
621	27	23	2464	1520
513	27	19	2336	1285
575	25	23	2336	1466
525	25	21	2272	1357
475	25	19	2208	1243

Table 7.1: **2 Dimensional DFT Performance**

The data is plotted in Figure 7.9. This plot shows the performance improvement available by using two similar sized matrices instead of a larger and a smaller matrix that produce a similar transform length. The far right hand side shows the peak performance for the array, which is slightly less than 2000 MFLOPs.

If the transform length is greater than the product of the maximum dimension of the two matrices allowed by the processing array size (without partitioning), two options are available. Either

1. Use a higher dimensional DFT
   This is dealt with in Section 7.3.1

or

2. Partition the two input matrices to fit within the processing array

If the partitioning option is chosen, then all four dimensions of the address generator are required for automatic partitioning. The time to calculate a partitioned two dimensional transform

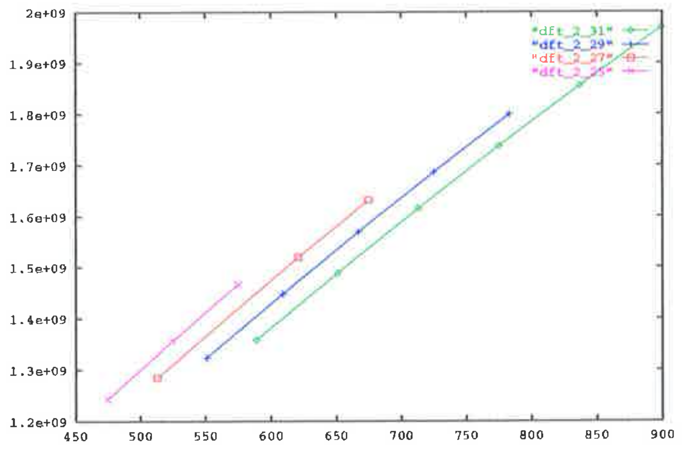


Figure 7.9: **Performance of Two Dimension DFT** - FLOPs vs DFT length

is the time to perform the two partitioned multiplies plus the time to unload the last partition calculated in the first multiplication. Table 7.2 shows the performance details of selected two dimensional partitioned DFTs with each input matrix being less than four times the size of the processing array, while the values are shown graphically in Figure 7.10.

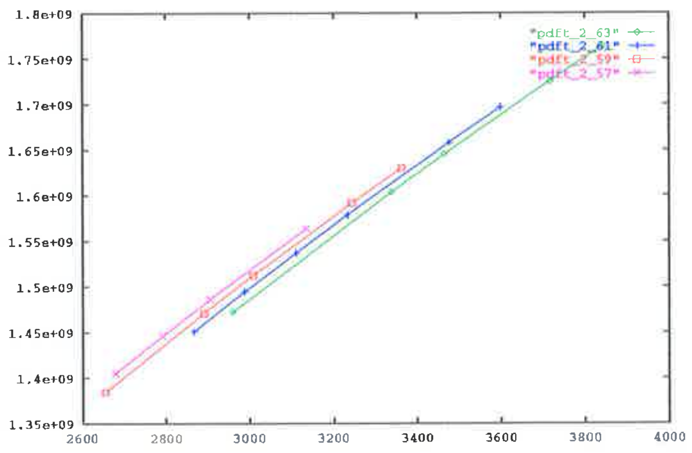


Figure 7.10: **Performance of Partitioned Two Dimension DFT** - FLOPs vs DFT length

Some performance estimates for selected larger-sized matrices are given in Table 7.3, and are plotted in Figure 7.11.

Three and Four Dimensional Transform

The alterative to using a partitioned two dimensional transform to perform larger DFTs is to use a higher dimensional Prime Factor Mapping. The decision to use add another dimension must

Transform length	N_1	N_2	Time (τ_C)	Performance (MFLOPs)
3843	63	61	16864	1763
3717	63	59	16608	1725
3465	63	55	16096	1645
3339	63	53	15840	1603
2961	63	47	15072	1472
3599	61	59	16288	1696
3477	61	57	16032	1658
3233	61	53	15520	1578
3111	61	51	15264	1537
2989	61	49	15008	1494
2867	61	47	14752	1450
3363	59	57	15712	1629
3245	59	55	15456	1591
3009	59	51	14944	1512
2891	59	49	14688	1470
2655	59	45	14176	1384
3135	57	55	15136	1563
2907	57	51	14624	1486
2793	57	49	14368	1446
2679	57	47	14112	1405

Table 7.2: **Partitioned 2 Dimensional DFT Performance**

Transform length	N_1	N_2	Time (τ_C)	Performance (MFLOPs)
8835	95	93	55136	1365
15875	127	125	130016	1107
24327	159	153	250592	919
24963	159	157	253792	934
36099	191	189	438752	810
101123	319	317	2036192	537
228483	479	477	6884192	384
407043	639	637	16333792	302
614431	799	769	31360992	245
636803	799	797	31920992	250

Table 7.3: **Partitioned 2 Dimensional DFT Performance**

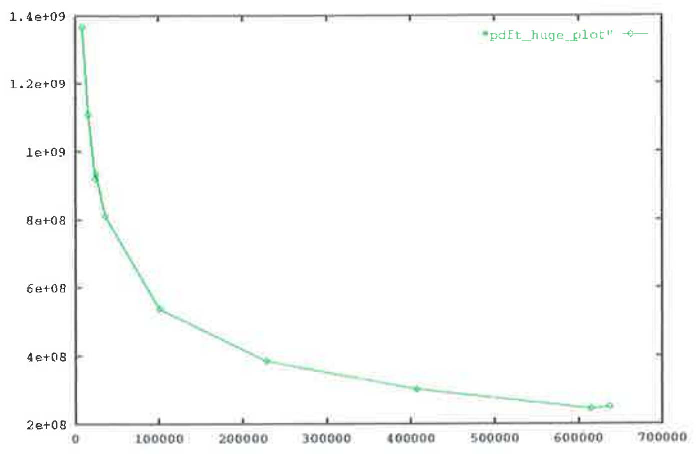


Figure 7.11: **Performance of Partitioned Two Dimension DFT** - FLOPs vs DFT length

be taken with care, as adding a new dimension of small matrix dimension will result in $2(n-1)$ inefficient product groups for an n-dimensional mapping. To illustrate, consider increasing a two dimensional mapping to three dimensions, and let the original two sizes be 31 and 29 on a 32×32 processing array. Then using a three dimensional mapping with dimension sizes 31,29 & 4 will require 31 (29×4) and 29 (31×4) matrix products, which would be very inefficient on the 32×32 processing array. However, as the size of the dimensions increases, the efficiency increases, and a higher dimensional mapping will have a higher performance than a partitioned lower dimensioned mapping.

The time to perform a multidimensional DFT is similar to the time to perform a two dimensional mapping, except that more calculations can be achieved before the the array must stall for an unload operation. In fact, if proper use is made of the dual ported RAMs, the unload stall can be avoided altogether. Table 7.4 shows the performance figures for selected three dimensional DFTs, which are also plotted in Figure 7.12

The time taken for a four dimensional transform can be calculated using the estimate for a three dimensional transform. The performance estimates are shown in Table 7.5, and graphically in Figure 7.13

7.4 The Kalman Filter

The Kalman filter described in Section 3.6 can be directly implemented on the matrix processing array with the bank-switched memory architecture that all previous performance estimates have used. The important performance measurement for the Kalman filter is the time it takes for a complete update, as this is in effect the repetition rate.

Two example systems can be estimated, one running entirely from static RAM (effectively running out of Cache RAM with no tag checking), and the other using a multilevel memory subsystem. Additionally, the size of the problem for each memory configuration can be varied such that the problem size is smaller than the processing array (no partitioning required) or the problem size is larger than the processing array (partitioning is required). Each case is considered separately. The processing array is assumed to be of size $p \times p$, with an access cycle time of τ nanoseconds and a memory latency of L where appropriate. The problem size is

Transform length	N_1	N_2	N_3	Time (τ_C)	Performance (MFLOPs)
24273	31	29	27	161216	1425
22475	31	29	25	153536	1375
20677	31	29	23	145856	1321
18879	31	29	21	138176	1261
14725	31	25	19	117696	1126
13175	31	25	17	110528	1060
19575	29	27	25	139712	1298
15225	29	25	21	118976	1155
9135	29	21	15	86976	898
4959	29	19	9	62912	628
12075	25	23	21	101312	1050
10925	25	23	19	95168	1001
9975	25	21	19	89536	962
4675	25	17	11	56768	653
2275	25	13	7	37824	436

Table 7.4: **Three Dimensional DFT Performance**

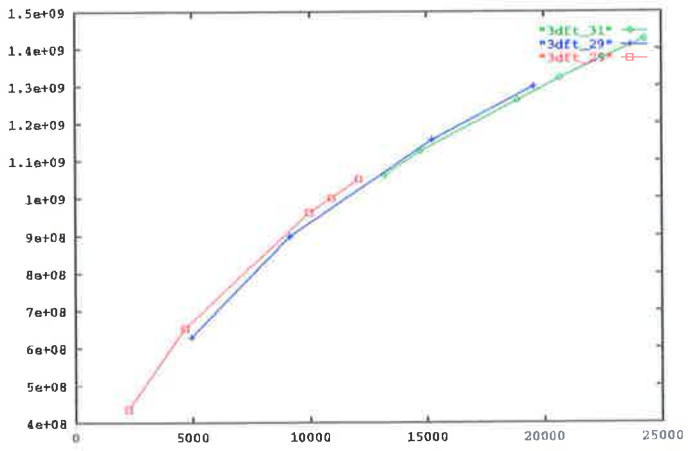


Figure 7.12: **Performance of Three Dimension DFT** - FLOPs vs DFT length

Transform length	N_1	N_2	N_3	N_4	Time (τ_C)	Performance (MFLOPs)
606825	31	29	27	25	25485440	297
558279	31	29	27	23	24231808	285
461187	31	29	27	19	21769088	259
471975	31	29	25	21	21732864	266
382075	31	29	25	17	19390976	237
292175	31	29	25	13	17108480	201
365769	31	27	23	19	18105728	242
276675	31	25	21	17	14859520	218
96255	31	23	15	9	8010880	129
229425	25	23	21	19	12370304	214
101745	21	19	17	15	6775680	162
62985	19	17	15	13	4757120	137

Table 7.5: **Four Dimensional DFT Performance**

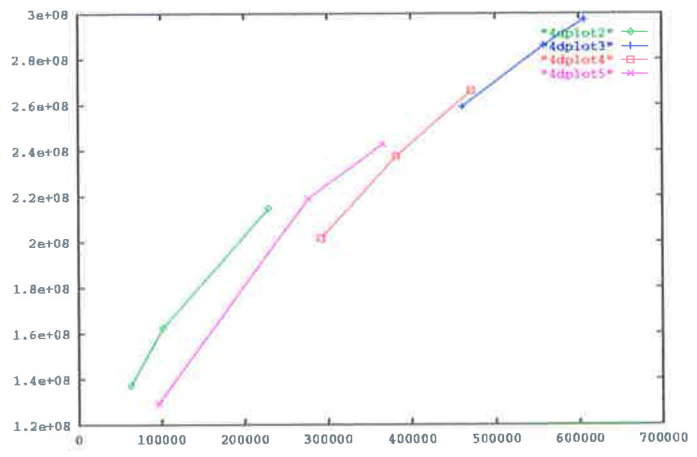


Figure 7.13: **Performance of Four Dimensional DFT** - FLOPs vs DFT length

assumed to be of length ϱ.

7.4.1 Small Problem Size Running From SRAM

This is the case of $p \geq \varrho$. As the processing array has a constant bandwidth, then the processor must stall after applying a wavefront until the time for a full width product has been completed, ie if $p-1$ operands are applied as a wavefront to the $p \times p$ processing array, the processing array must stall for one cycle, if $p-2$ operands are applied, the processor must stall for two cycles, etc. This only applies in one dimension of the applied matrices. If the matrix product **AB** is performed and **A** is of dimension $r \times s$ and **B** is of dimension $s \times t$, then the application of both **A** & **B** will take ps cycles, and the unloading of the result will take $\max(pr, pt)$ cycles. Therefore, if all matrices are assumed to be square and of dimension ϱ, non-square matrices use an equivalent time if the larger of the two dimensions is made the dimension of the square matrix.

Proceeding in the same order as is provided in Table 3.1, the time to compute the matrix $\mathbf{K}(k)$ can be estimated as follows:

$$T_1 \approx p\varrho \times \tau \tag{7.20}$$
$$T_U \approx 2 \times p\varrho \times \tau \tag{7.21}$$
$$T_{Delay} \approx p^2 \times \tau \tag{7.22}$$
$$T_{inv.} \approx 15p^2\tau + \frac{T^2_{scalar\ divide}}{2}\tau \tag{7.23}$$
$$T_{end} \approx p\varrho \times \tau \tag{7.24}$$

where the time for inversion is from Equation 7.12. Summing these to determine the time to calculate $\mathbf{K}(k)$ produces Equation 7.25.

$$\begin{aligned} T_{\mathrm{K}} &\approx T_1 + T_U + 2T_{Delay} + T_{inv.} + T_{end} \\ &\approx \left(4p\varrho + 2p^2 + \frac{T^2_{scalar\ divide}}{2}\right)\tau \end{aligned} \tag{7.25}$$

Similarly, the determination of $\mathbf{P}(k+1|k)$ requires five matrix products and one accumulate which takes the same time as a multiply, which all can take place without stalls. The estimate for the $\mathbf{P}(k+1|k)$ update is then given by Equation 7.26.

$$T_{\mathrm{P}} \approx 5 \times p\varrho \times \tau \tag{7.26}$$

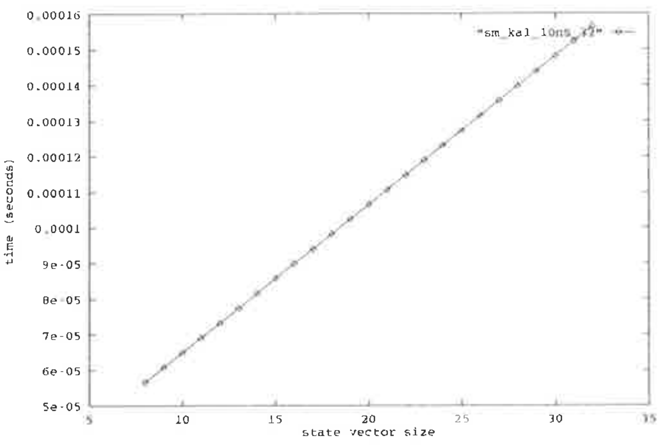
The final part is the state estimate vector update $\hat{\mathbf{x}}(k+1|k)$, which requires four matrix-vector products. As the processing array has a constant bandwidth, the matrix-vector products effectively consume the same time as a matrix-matrix product. Therefore, the $\hat{\mathbf{x}}(k+1|k)$ update consumes time according to Equation 7.27.

$$T_{\hat{x}} \approx 4 \times p\varrho \times \tau \tag{7.27}$$

Summing all components provides an estimate of the time the Kalman filter takes in this simple implementation. The resultant time is given in Equation 7.28.

$$\begin{aligned} T_{Kalman1} &= T_{\mathrm{K}} + T_{\mathrm{P}} + T_{\hat{x}} \\ &\approx \left(4p\varrho + 2p^2 + \frac{T^2_{scalar\ divide}}{2}\right)\tau + 5 \times p\varrho \times \tau + 4 \times p\varrho \times \tau \\ &\approx \left(13p\varrho + 2p^2 + \frac{T^2_{scalar\ divide}}{2}\right)\tau \end{aligned} \tag{7.28}$$

This result is shown plotted in Figures 7.14a) to d) for array sizes of 32 and 64 and cycle times of ten and twenty nanoseconds. The plots show the time on the vertical scale and the number of elements in the state vector $\hat{\mathbf{x}}(k|k-1)$ on the horizontal axis. The plots show the time increasing linearly with vector size, as the matrix processor multiplies matrices in approximately linear time. The number of multiplications required simply introduces a constant factor to the time for computation.



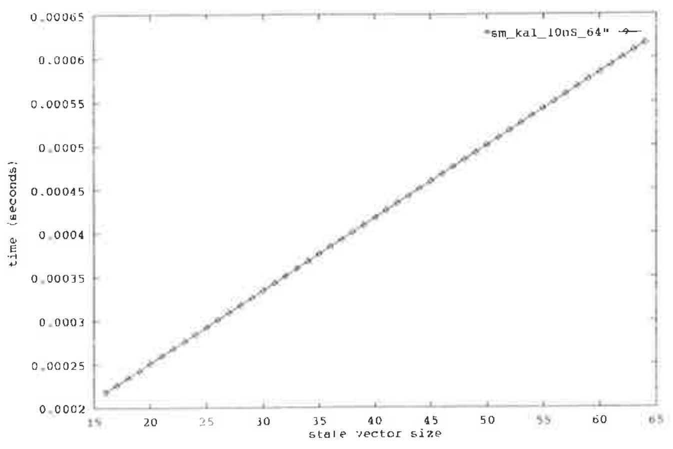
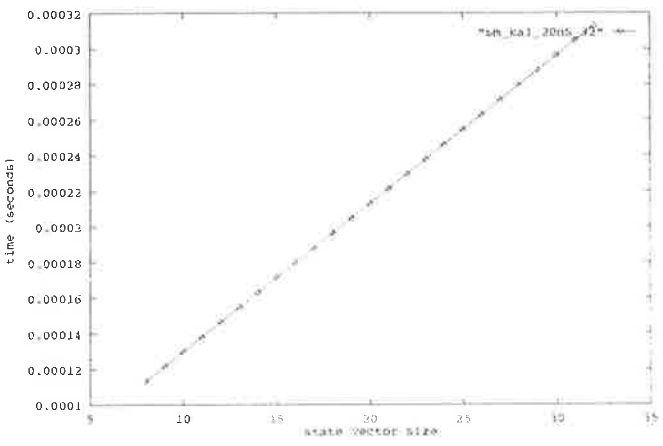


Figure 7.14: **10ns Cycle Time** a) **Array Size = 32** b) **Array Size = 64**



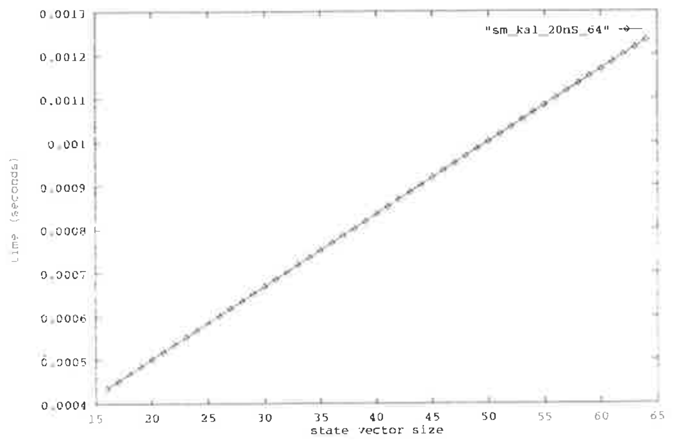


Figure 7.14: **20ns Cycle Time** c) **Array Size = 32** d) **Array Size = 64**

Figure 7.14: **Simple Kalman Filter for State Vector Smaller Than Array Dimension**

7.4.2 Large Problem Size Running From SRAM

Now the problem must be partitioned into matrices that fit the processor array size. The time to perform the multiplication of two matrices that are larger than the size of the processing array is equivalent to the sum of the time to calculate all the full matrix partitions plus the time to calculate all the partial matrix partitions.

Calculating the Kalman filter in the same order as provided in Table 3.2, the approximate times required for the computation can again be broken into the component calculations of $\mathbf{K}(k)$, $\mathbf{P}(k+1|k)$ & $\hat{\mathbf{x}}(k+1|k)$. The time estimation for the computation of $\mathbf{K}(k)$ proceedes as follows.

To calculate the temporary matrix $\mathbf{L}$, each of the partitions $\mathbf{L}_{ij}$ must be calculated in turn. The time for these calculations has been determined in Equation 7.1, and is

$$T_{\text{mult}} = \left\lceil \frac{R}{p} \right\rceil \times \left\lceil \frac{T}{p} \right\rceil \times p \times S \times \tau \tag{7.29}$$

The calculation of the temporary matrix $\mathbf{U}$ also requires an addition at the start of each partition calculation, which will add approximately p^2 cycles to each partition calculation time. The time to calculate $\mathbf{U}$ can then be calculated from Equation 7.30.

$$\begin{aligned} T_{\text{U}} &= \left\lceil \frac{R}{p} \right\rceil \times \left\lceil \frac{T}{p} \right\rceil \times \left(p \times S + p^2\right) \times \tau \\ &= \left\lceil \frac{R}{p} \right\rceil \times \left\lceil \frac{T}{p} \right\rceil \times (S+p)p \times \tau \end{aligned} \tag{7.30}$$

The inversion procedure is again similar to that determined earlier, except simplified for the less complex memory system. The time to invert each $p \times p$ block is again

$$T_{\text{inv}} \approx \left(15p^2 + \frac{T^2_{\mathit{scalar\ div.}}}{2}\right) \times \tau \tag{7.31}$$

and so the total time for the inversion of each of the pivot blocks is simply the time for one inversion multiplied by the number of partitions, as shown in Equation 7.32.

$$T_{\text{all piv}} \approx \left(15p^2 + \frac{T^2_{\mathit{scalar\ div.}}}{2}\right) \times \tau \times \left\lceil \frac{\varrho}{p} \right\rceil \tag{7.32}$$

In each column, there will be a maximum of $\left\lceil \frac{\varrho}{p} \right\rceil$ partitions, so there will be $2\left\lceil \frac{\varrho}{p} \right\rceil - 1$ updates per column. The total number of updates for the complete inversion process is then the average number per iteration multiplied by the number of iterations. This is expressed in Equation 7.33.

$$\begin{aligned} T_{\text{full inv}} &\approx T_{\text{all piv}} + T_{\text{update}} \\ &\approx \left(15p^2 + \frac{T^2_{\mathit{scalar\ div.}}}{2}\right) \times \tau \times \left\lceil \frac{\varrho}{p} \right\rceil + \left(2\left\lceil \frac{\varrho}{p} \right\rceil - 1\right) \frac{\left(\left\lceil \frac{\varrho}{p} \right\rceil - 1\right)^2}{2} p^2 \tau \end{aligned} \tag{7.33}$$

After the inversion, the matrix $\mathbf{K}(k)$ can be determined with one extra multiplication, which takes time according to the previously determined Equation 7.29. Note that no delay slots are required, as the feedback of one partition can now be hidden by the calculation of another partition.

The calculation of $\mathbf{P}(K+1|k)$ is simply three partitioned multiplies, as for Equation 7.29, and then two inplace multiplies and an inplace accumulate. The time for the inplace multiplications are each approximately equal to the time to perform a multiply-accumulate operation, as the only difference is the ordering of the multiplications of the partitions. The time required for the inplace accumulate is simply the time required for an 'array sized' multiplication, as it is, in effect, multiplying an 'array sized' matrix by the identity matrix $\mathbf{I}$. Therefore, the time to update $\mathbf{P}(K+1|k)$ can be approximated by the time to perform five partitioned multiplies plus one array-sized multiplication. This is summarised in Equation 7.34.

$$\begin{aligned} T_{\text{P update}} &\approx 3T_{\text{Part. mult.}} + 2T_{\text{Part. mult-acc}} + T_{\text{Part. acc}} \\ &\approx (5S+p)\left\lceil \frac{R}{p} \right\rceil \times \left\lceil \frac{T}{p} \right\rceil \times p \times \tau \end{aligned} \tag{7.34}$$

When calculating the updated state vector $\hat{\mathbf{x}}(k+1|k)$, we recall that the matrix-vector product operation must use the full bandwidth of the array. Therefore, the matrix-vector product will consume the same time as a matrix-matrix product with the vector replaced by a matrix of the same width as the processing array. The time to perform each of the three matrix-vector products involved in the calculation of $\hat{\mathbf{x}}(k+1|k)$ can therefore be approximated by the time in Equation 7.35, while the vector accumulate takes the time shown in Equation 7.36.

$$T_{\text{vect mult}} \approx \left\lceil \frac{\varrho}{p} \right\rceil \times p\varrho \times \tau \tag{7.35}$$

$$T_{\text{vect acc}} \approx p \times \varrho \times \tau \tag{7.36}$$

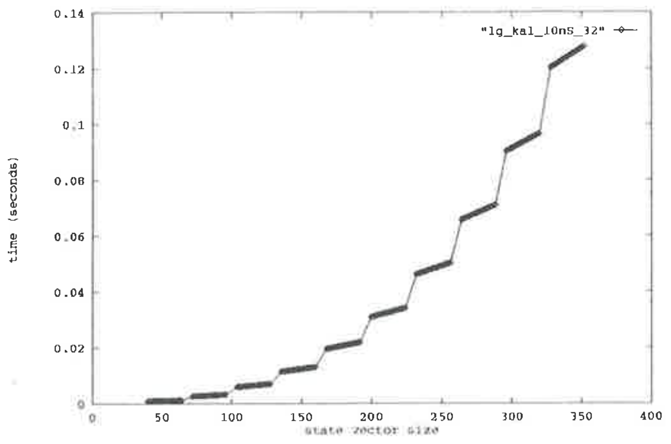
The total time for a large Kalman filter with a simple memory system is then the sum of all the parts, as shown in Equation 7.37, for the case $R = S = T = \varrho$.

$$\begin{aligned} T_{\text{Large Kalman}} &= T_K + T_P + T_{\hat{x}} \\ &\approx \left(\left\lceil \frac{\varrho}{p} \right\rceil \times \left\lceil \frac{\varrho}{p} \right\rceil \times (2\varrho+p)p \times \tau + \left(15p^2 + \frac{T^2_{\mathit{scalar\ div.}}}{2}\right) \times \tau \times \left\lceil \frac{\varrho}{p} \right\rceil + \right. \\ &\quad \left. \left(2\left\lceil \frac{\varrho}{p} \right\rceil - 1\right) \frac{\left(\left\lceil \frac{\varrho}{p} \right\rceil - 1\right)^2}{2} p^2 \tau + \left\lceil \frac{\varrho}{p} \right\rceil^2 \times p \times \varrho \times \tau \right) + \\ &\quad \left(\left\lceil \frac{\varrho}{p} \right\rceil \times \left\lceil \frac{\varrho}{p} \right\rceil \times (5\varrho+p)p \times \tau \right) + \left(\left(3\left\lceil \frac{\varrho}{p} \right\rceil + 1\right) \times p\varrho \times \tau \right) \\ &\approx \left(\left\lceil \frac{\varrho}{p} \right\rceil^2 p(8\varrho+2p) + \left(3\left\lceil \frac{\varrho}{p} \right\rceil + 1\right) p\varrho + \right. \\ &\quad \left. \left(15p^2 + \frac{T^2_{\mathit{scalar\ div.}}}{2}\right) \left\lceil \frac{\varrho}{p} \right\rceil + \left(2\left\lceil \frac{\varrho}{p} \right\rceil - 1\right) \frac{\left(\left\lceil \frac{\varrho}{p} \right\rceil - 1\right)^2}{2} p^2 \right) \tau \end{aligned} \tag{7.37}$$

Equation 7.37 was evaluated for a range of vector sizes, implemented on processing arrays of size 32 and 64 processing elements per side and with cycle times of ten and twenty nanoseconds. The resultant times are plotted in Figures 7.15a) to d).

7.4.3 Kalman Filter With a Multi-level Memory Sub-system

If a multilevel memory subsystem is used, then the time estimates for the various parts of the Kalman filter can be obtained directly from the equations derived in Sections 7.1 & 7.2. The ordering of operations in Tables 3.1 & 3.2 still applies, as do assumptions above concerning the number of equivalent cycles used in the estimates of Sections 7.4.1 & 7.4.2. As the equations in Sections 7.1 & 7.2 assume data is initially stored in main memory, then no cache reuse is assumed between successive operations. Therefore, the performance figures estimated here can be a lower bound on the performance. As cache reuse increases (the case where the cache is



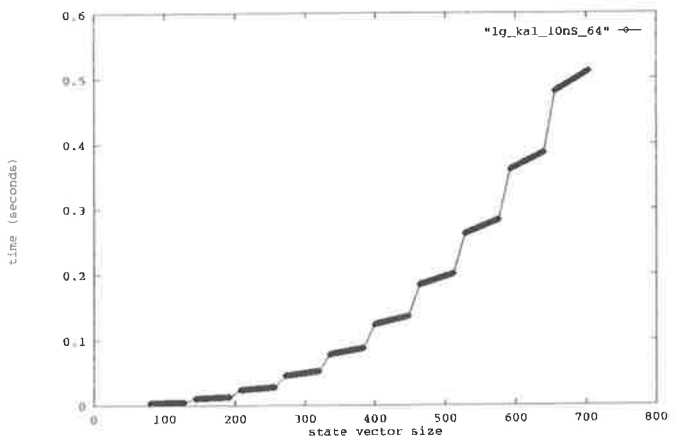
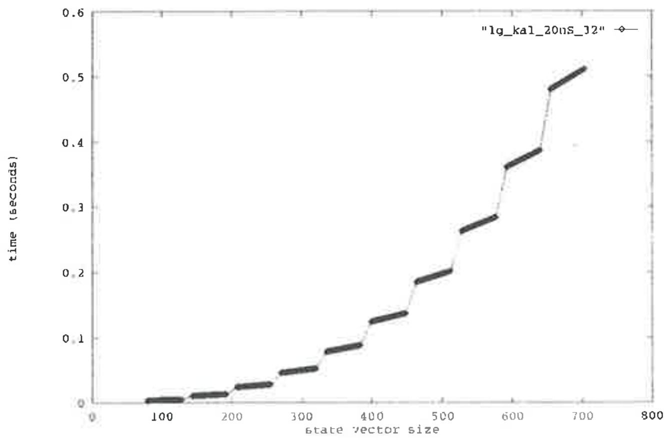


Figure 7.15: **10ns Cycle Time** a) **Array Size = 32** b) **Array Size = 64**



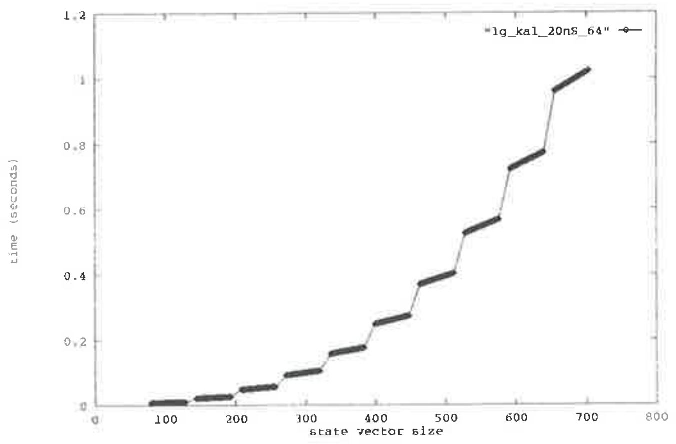


Figure 7.15: **20ns Cycle Time** c) **Array Size = 32** d) **Array Size = 64**

Figure 7.15: **Simple Kalman Filter for State Vector Larger Than Array Dimension**

large enough to hold more data), the performance will approach the case of the system running directly from SRAM.

As before, the calculation of '$\mathbf{K}(k)$' will require two matrix products, one in-place accumulate and one matrix inversion. The matrix products will take time determined by Equation 7.6 for the full size of the matrices, while the time required per partition for the inplace accumulate uses the same equation except for the matrix size is the same size as the processing array (= matrix partition size). The time for the matrix inversion can be obtained directly from Equation 7.12. The time taken to calculate $\mathbf{K}(k)$ can be estimated to be as shown in Equation 7.38.

$$T_{\mathrm{ML\ K}} \approx 2T_{\varrho \times \varrho \text{ mult}} + \left\lceil \frac{\varrho}{p} \right\rceil^2 \times T_{p \times p \text{ mult}} + T_{\varrho \times \varrho \text{ invert}} \tag{7.38}$$

Similarly, the calculation of '$\mathbf{P}(k+1|k)$' requires five multiplications and one in-place accumulate. Therefore, the time required can be approximately determined using Equation 7.39.

$$T_{\mathrm{ML\ P}} \approx 5T_{\varrho \times \varrho \text{ mult}} + \left\lceil \frac{\varrho}{p} \right\rceil^2 \times T_{p \times p \text{ mult}} \tag{7.39}$$

To calculate the updated state vector '$\hat{\mathbf{x}}(k+1|k)$', the computation time taken by a matrix-vector product needs to be estimated. As the matrix in the matrix-vector product will be the dominant factor for loads, then we can estimate the load & calculation time, assuming streaming, to be as shown in Equation 7.40.

$$T_{\text{mat./vect. prod}} \approx \left(\frac{\varrho^2}{D} L + p\varrho \left\lceil \frac{\varrho}{p} \right\rceil \times \tau \right) \tag{7.40}$$

Similarly, the time to accumulate a vector with the contents of the processing array is simply the time required to load a partition of the input vector, as shown in Equation 7.41.

$$T_{\text{vect acc}} \approx \left\lceil \frac{p}{D} \right\rceil L + p^2 \times \tau \tag{7.41}$$

Therefore, as the updating of the state vector requires three matrix-vector multiplies and one vector accumulate, the time required to calculate $\hat{\mathbf{x}}(k+1|k)$ is as shown in Equation 7.42.

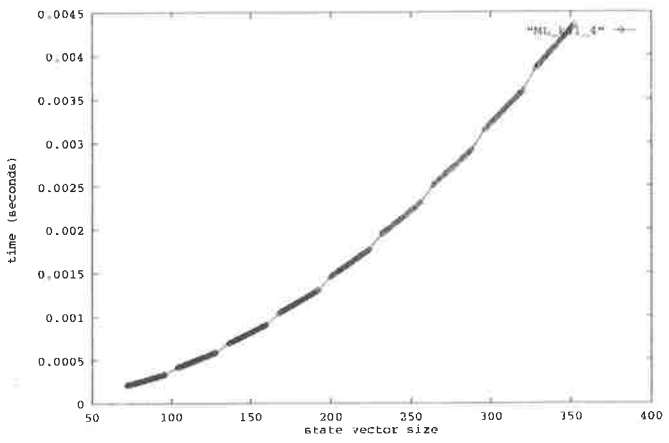
$$T_{\mathrm{ML}\ \hat{\mathbf{x}}} \approx 3T_{\text{mat./vect. prod}} + T_{\text{vect acc}} \tag{7.42}$$

The time to complete a cycle of the Kalman filter is then the sum of all the updates, as shown in Equation 7.43.

$$T_{\text{ML cycle}} = T_{\mathrm{ML\ K}} + T_{\mathrm{ML\ P}} + T_{\mathrm{ML}\ \hat{\mathbf{x}}} \tag{7.43}$$

Equation 7.43 was evaluated for a range of state vector sizes, using block sizes varying between 4 and 512 on a 32×32 processing array with a ten nanosecond cycle time and 100 nanosecond main memory latency. The results are shown in Figures 7.16a) to d).

As is block size increases, the time to compute the Kalman filter can be seen to decrease. However, there is minimal significant difference between block sizes of 64 and 512. It is stressed again that these estimates are *lower bounds*. The practical performance should be much better, as there is a large amount of data reuse and also of data cycling from the output back to an input.



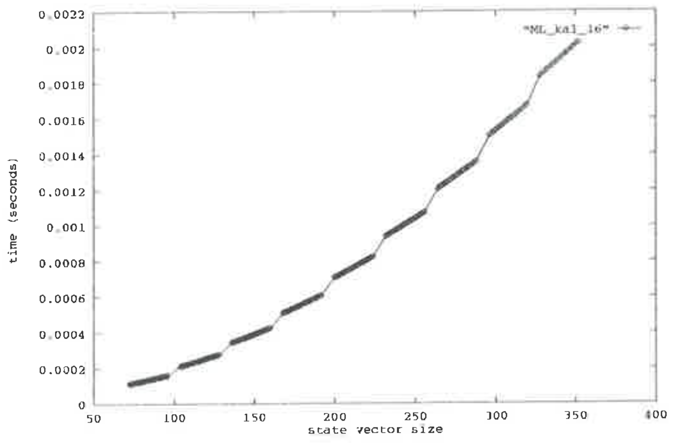
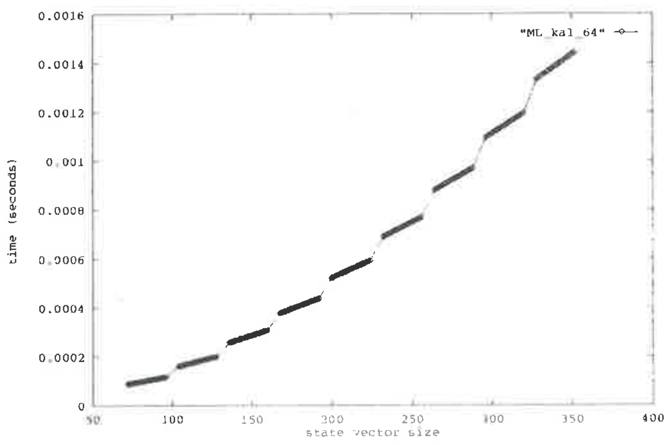


Figure 7.16: a) **Block Size = 4** b) **Block Size = 16**



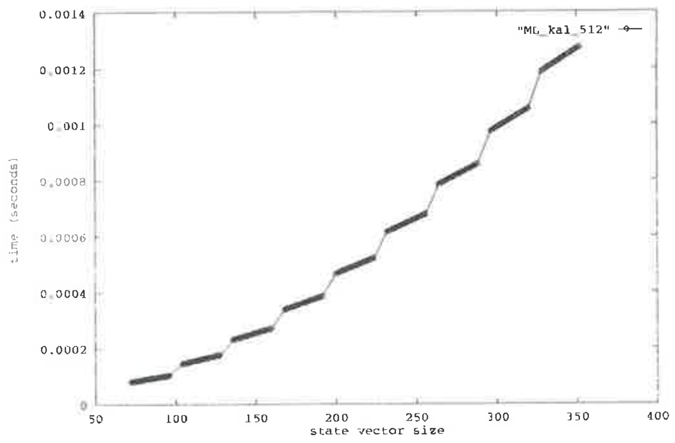


Figure 7.16: a) **Block Size = 64** b) **Block Size = 512**

Figure 7.16: **Lower Bound on Kalman Filter With Multilevel Memory**

7.5 Conclusion

The performance estimates shown in this chapter provide a significant insight to the potential performance that is available using systolic techniques coupled with clever addressing and interface units. Very few computer systems can provide sustained performance in the GigaFLOP range, in fields as varied as numerical systems to digital signal processing and control.

Chapter 8

Future Projects

8.1 A Multi-Processor Teraflop Engine

Although many in engineering see this field as a continual push forward toward an ultimate goal of infinitely fast processors along the most direct path, some observers liken the process more to a corkscrew than an arrow. We have seen processor and memory limits leapfrog forward, with a new topology suggested at each improvement. However, history shows that some ideas keep resurfacing on a regular basis whenever the relationship of technologies is at the correct point[1]. The architecture suggested here is just such a case, nothing new, but a familiar architecture revisited.

The discussion that follows is not an attempt to define a complete implementation of a full multiprocessor system. Indeed, the design work required for such a task would require several man-years. It is meant as an indication of the approximate performance that would be available given such a system, and also some suggested enhancements that could be used to extract the most from the base system.

8.1.1 The Hypercube

The interconnection of nodes within a multiprocessor is the subject of much debate at present, with many different topologies being suggested as the 'ideal'. Current commercial topologies include the mesh, fat-tree, torroidal mesh and hypercube from Intel, Thinking Machines, Cray and nCube, to name a few [29].

Not long ago, the hypercube, or n-cube, was touted as the ultimate interconnect, with commercial machines from Intel, Thinking Machines and nCube and experimental machines being developed at Caltech, CMU and other establishments. The hypercube is a 'rich' structure that contains many others within it, and as such can fully utilise its own structure or be reconfigured to simpler topologies such as mesh, ring or tree structures. As such, the hypercube matches or includes many configurations that naturally occur in mathematical, scientific and engineering problems. Indeed, Edelman of the University of California, Berkeley, stated recently [25], "The hypercube with full concurrent communication to every node's nearest neighbors is the only mathematically elegant communications network that has ever been devised."

One of the niceties of hypercubes is that they grow by replication. Given two identical $m-$cubes (each with 2^m nodes), an $(m+1)-$cube is formed by linking their corresponding nodes in a one-to-one fashion. This leads to the node numbering concept that, for an $m-$cube labelled with the numbers 0 to $2^m - 1$, the nodes are connected in such a way that the binary representation of connected nodes differ in exactly one bit. An example labelling for a $3-$cube

[1]RISC/CISC is an example of this, as is Bipolar/BiCMOS

is given in Figure 8.1. Alternative labels allows a ring of arbitrary length l, where $4 \leq l \leq 2^m$, or a mesh of size $2^u \times 2^v$, where $u + v = m$, to be mapped onto an $m-$cube [76].

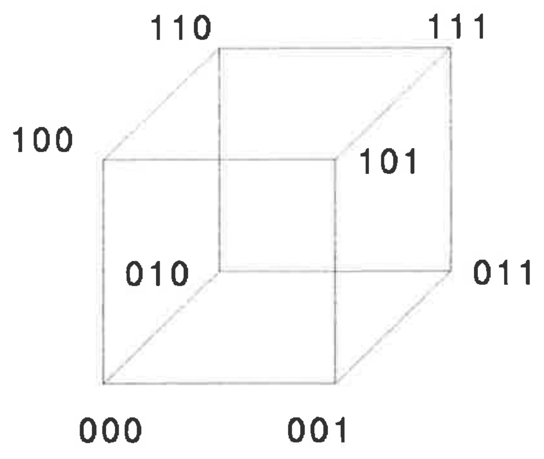


Figure 8.1: **Node Labelling for a 3-Cube**

Why Not Use It?

The move away from the hypercube architecture is generally due to the difficulty in scaling the hypercube above a dimension of about ten (ie $2^{10} = 1024$ nodes). With a hypercube of size 2^n, each node has n connections, one to each of its nearest n neighbours. Thus, the number of interconnections grows rapidly with machine size, posing a dilemma: if the interconnects are all bit-serial copper links, the speed is often inadequate. On the other hand, if the interconnects are parallel, byte wide or larger, then a large number of semiconductor pins are required to drive them, and the system becomes unwieldy and difficult to make reliable. As Michael Meirer of Thinking Machines said [93]. "Unless you have all the hardware drivers and communication channels on chip, you cannot implement them. You end up with too much hardware; the reliability and sheer size become unmanageable."

The hypercube was viable in the 1980's because the computational units were simple enough (eg. Thinking Machines CM-2) or slow enough (eg Intel i/PSC) that a bit-serial or lower-bandwidth communication strategy could be used. As the number of nodes and the node performance increased, so did the number of interconnections and the communication requirements. New topologies such as the mesh and fat tree were introduced (actually *re*introduced), as local communication with these topologies is machine-size independent. Therefore, the only way to reintroduce the hypercube is to make the interconnections sufficiently simple that the gains in topology greatly outway the connection complexity.

8.1.2 Fibre-Optical Interconnection

Current multiprocessors using metal interconnects increase the node-to-node bandwidth by transmitting data with widths of bytes (8 bits), words (32-bits) or long words (64-bits). Therefore, the interconnection is physically large, and the capacitance that must be driven and discharged, proportional to the length of the line and the number of lines, will consume large quantities of power, increase delays and reduce throughput.

Fibre optics, on the other hand, typically benefit from:

- much lower attenuation for a similar length of line
- a propagation speed independent of signal length
- lower signal and clock skew
- higher immunity to crosstalk, electromagnetic interference and ground loops
- flexibility in 3-dimensional space (not constrained largely to two or three perpendicular directions)

Fibre-optic interconnections used in communication systems achieve speeds between 1 and 1.5 Gigabits per second on a standard serial connection, and 2 Gigabits per second is available. Therefore, a bidirectional internode connection rate of 320 Mbytes per second is quite feasible using four strands of optical fibre [79]. As this design is for a moderate number of nodes (16 to 1024), limited by other factors such as power dissipation, problem size, etc., each node will have at most ten (= $\log_2 1024$) connections to other nodes. As each connection contains four optical strands, the total number of strands leaving a node will be at most 40. It is interesting to compare that a mesh connected system with four neighbours to each node (North, South, East, West) using standard copper interconnects would likely have 32 or 64 copper tracks or strands between nodes, a similar amount to an optically interconnected hypercube.

The interconnection bandwidth and internode latency determine the time required to transmit data from one node to another. However, broadcasting the same data to all nodes in a hypercube is not a simple matter, and is discussed in Appendix C. Ultimately, the time to broadcast 'L' words over a hypercube with 2^m nodes can be minimised to that in Equation 8.1

$$T_{PR}(\mu_{opt}) = \left(\sqrt{\frac{L\tau_c}{m}} + \sqrt{m\beta}\right)^2 \qquad (8.1)$$

where β is the node-to-node communication latency and τ_c is the time to transmit one word to a nearest neighbour node. For an optically connected hypercube, both m and β are implementation dependent, while L is problem size dependent and τ_c is technology dependent.

8.1.3 Utilising the Hypercube - the Proposed Model

In the proposed architecture, the memory is distributed among the nodes and processors, ie there is no shared memory, even within the node. However, due to the logical labelling of a hypercube's nodes, the memory system can be viewed as physically distributed but globally and logically shared. A custom VLSI design implementing a directory search engine utilising an algorithm similar to the Stanford DASH architecture [50] is suggested, as this allows the scalability of a message-passing machine to be combined with the model of a shared-memory architecture for ease of programming.

The suggested node configuration is one that is based on a hybrid of the Intel Paragon XP/S and a higher performance vector supercomputer. The Paragon XP/S contains up to five i860 RISC processors, allocating four processors to the computational task of problem solving and the fifth dedicated to the task of managing the message passing and node connections [93, 94]. A high end multiprocessor vector supercomputer typically uses a relatively small number of very powerful vector processors (typically one to sixteen processors, each capable of between 500 Mflops and 8000 Mflops - see Table 2.4). The hybrid system is a combination of a modest number of matrix engines with attached scalar units (typically three) and the associated communication hardware (another scalar processor plus custom VLSI directory/coherency manager and node link hardware) at each node. The nodes are then replicated into a hypercube, to become an MPP[2] of very high performance nodes.

The advantage of this approach is that the complete system can achieve high performance with a relatively modest number of nodes. Despite claims by Burkhardt [93], of Kendall Square Research, and others, that their massively parallel processor machines are only limited by economics, the latency that must exist for long communication paths as the system grows will result in detrimental communication overheads for problems that require a large amount of data movement between nodes. Therefore, by keeping the number of nodes and hence the critical path between nodes relatively small the communication overheads will be much smaller for similar performance.

As mentioned above, each node will consist of approximately three matrix engines, each with attached scalar processor, for computation. A minimum of nine banks of memory will be required for the matrix processors, so increasing the number of banks to sixteen will allow ample extra banks for swapping of data between computation processors and between computation-communication processors. The interface between the memory and the processors can be either an extension of the memory structure used for a single matrix array, as shown in Figure 8.2a), or a full cross-connection allowing any matrix array pipe to access any memory, as shown in Figure 8.2b).

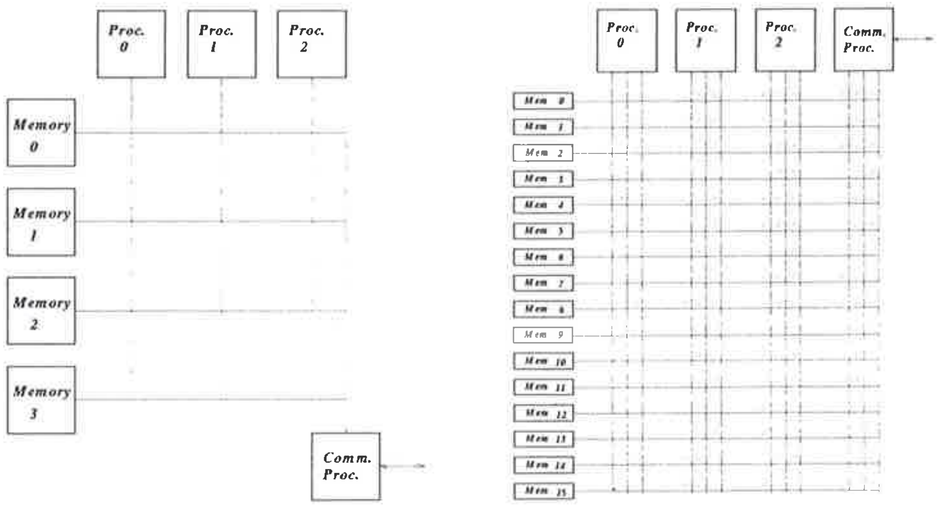


Figure 8.2: **a) Extended Simple Memory Structure b) X-Connect Memory Structure**

8.1.4 Algorithms

Multiplication on the Hypercube

Several authors have investigated the efficiency and performance of matrix multiplication on a hypercube architecture [29, 28, 76, 42, 74, 78]. The act of splitting the problem across several processors necessitates the partitioning of the problem, which will generally be rectangular. In fact, it has been found that, for a standard scalar processor architecture at the nodes, square partitioning (a subset of rectangular partitioning) yields the optimum results. For this case,

[2]Massively Parallel Processor

various 'roll and rotate' algorithms have been developed to exploit the special structure of the hypercube [29, 28, 74, 86].

Due to the structure of the matrix engines at each node, which perform substantially better for long 'strips' of products, a rectangular structure would be preferable. As such, the matrix is partitioned to allow large 'inner products of outer products'. Consider the matrix product $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A}$ is a $P \times Q$ matrix, $\mathbf{B}$ is a $Q \times R$ matrix, and so $\mathbf{C}$ is a $P \times R$ matrix. Now partition the matrix into 'processing array' sized strips, ie $\mathbf{A}$ is partitioned into ψ $p \times Q$ partitions and $\mathbf{B}$ is partitioned into ϱ $Q \times p$ partitions for processing on arrays of size $p \times p$. The matrix product can now be determined in terms of the partitioned pıoducts as shown in Equation 8.2.

$$\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1\varrho} \\ C_{21} & C_{22} & \cdots & C_{2\varrho} \\ \vdots & \vdots & \ddots & \vdots \\ C_{\psi 1} & C_{\psi 2} & \cdots & C_{\psi\varrho} \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_\psi \end{pmatrix} \times \begin{pmatrix} B_1 & B_2 & \cdots & B_\varrho \end{pmatrix} \tag{8.2}$$

This naturally suggests a ring structure with the partitions of one matrices rotating around the nodes.

Now distribute the partitions equally among the nodes. Ideally, there should be more partitions than processing arrays. If this is not the case, it may be more efficient to use fewer nodes, and run another problem on the unused nodes. Assume that there is at least one more partition at each node than there are processing arrays, ie for a system with three processing arrays per node, there are four or more partitions at each node. Therefore, one of the partitions can be transmitted while the others are being operated on. This is shown in Figure 8.3.

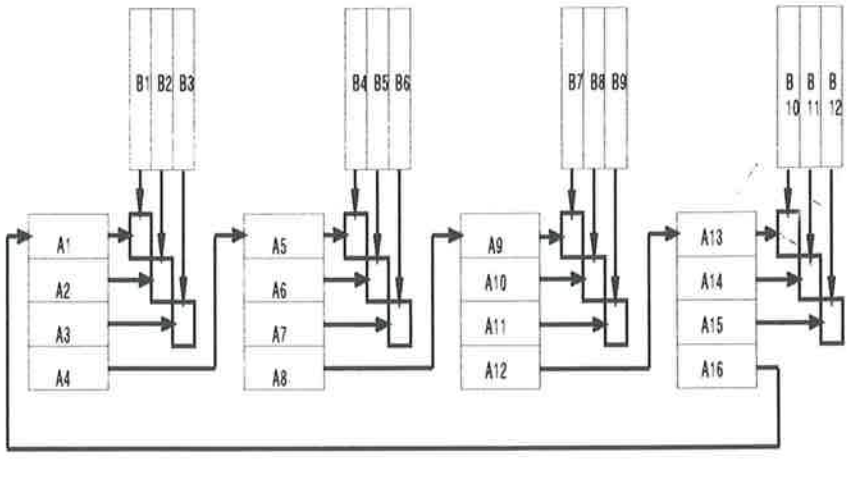


Figure 8.3: **Processing/transmission Diagram for 1st Partitioning**

The choice of matrix that is rotated is arbitrary, but careful selection will yield a better result than a poor choice due to processor load balancing. If one of the matrices is partitioned such that the number of processing array width partitions is equal to or just smaller than a multiple of three (the number of processing arrays at a node), then make this the stable partition. If all the partitions in the stable matrix are multiplied by a single partition in the rotation matrix, the processing times for all processing arrays will be approximately equal. Next rotate the second matrix (the rotating matrix) by one partition, and repeat the multiplication step. Repeat the procedure above until all partitions in one matrix have been multiplied by all partitions in the other matrix. This is shown graphically in Figure 8.4 for the multiplication of a matrix with ten partitions (**A**) with a matrix with twelve partitions (**B**) on a multiprocessor with two nodes.

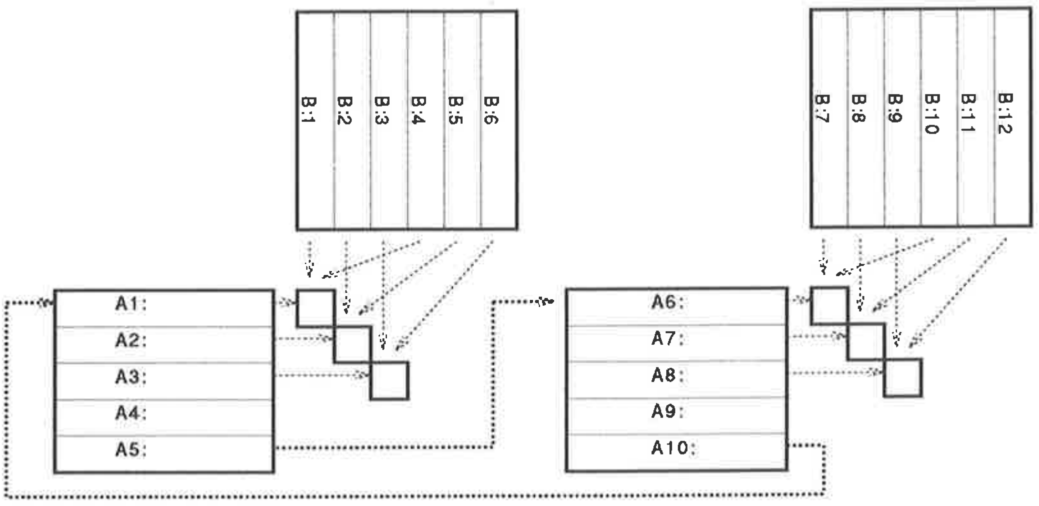


Figure 8.4: a) **First Multiplication Phase**

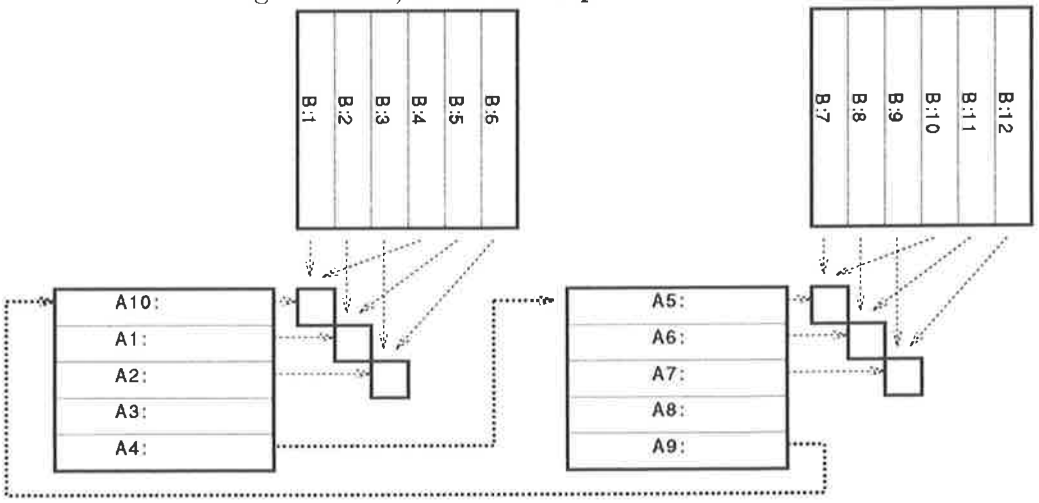


Figure 8.4: b) **Second Multiplication Phase After First 'Roll'**

As there are six partitions of the **B** matrix per node, then two partitions ($\frac{\textit{No. of partitions}}{\textit{processing arrays per node}}$) of **B** are multiplied by each partition of **A** for each 'roll'. Table 8.1 shows the cycle at which each multiplication is completed, with 'a' indicating the first multiplication in a cycle and 'b' indicating the second, and also the processing array in which the multiplication takes place (in '[ ]').

The bottleneck in the process can be either the computation or the data transmission, depending on factors such as the internode latency, the cache performance and the number of bands at each node. To estimate the performance of the system, the computation and transmission times can be calculated, and the maximum of the two used to estimate the cycle time. As the cycle time is basically the same for all matrix band products calculated simultaneously, the total

Result C		Matrix A partitions									
		1	2	3	4	5	6	7	8	9	10
	1	1a[1]	10a[1]	9a[1]	8a[1]	7a[1]	6a[1]	5a[1]	4a[1]	3a[1]	2a[1]
	2	2a[2]	1a[2]	10a[2]	9a[2]	8a[2]	7a[2]	6a[2]	5a[2]	4a[2]	3a[2]
	3	3a[3]	2a[3]	1a[3]	10a[3]	9a[3]	8a[3]	7a[3]	6a[3]	5a[3]	4a[3]
	4	1b[1]	10b[1]	9b[1]	8b[1]	7b[1]	6b[1]	5b[1]	4b[1]	3b[1]	2b[1]
Matrix	5	2b[2]	1b[2]	10b[2]	9b[2]	8b[2]	7b[2]	6b[2]	5b[2]	4b[2]	3b[2]
B	6	3b[2]	2b[3]	1b[3]	10b[3]	9b[3]	8b[3]	7b[3]	6b[3]	5b[3]	4b[3]
	7	6a[4]	5a[4]	4a[4]	3a[4]	2a[4]	1a[4]	10a[4]	9a[4]	8a[4]	7a[4]
	8	7a[5]	6a[5]	5a[5]	4a[5]	3a[5]	2a[5]	1a[5]	10a[5]	9a[5]	8a[5]
	9	8a[6]	7a[6]	6a[6]	5a[6]	4a[6]	3a[6]	2a[6]	1a[6]	10a[6]	9a[6]
	10	6b[4]	5b[4]	4b[4]	3b[4]	2b[4]	1b[4]	10b[4]	9b[4]	8b[4]	7b[4]
	11	7b[5]	6b[5]	5b[5]	4b[5]	3b[5]	2b[5]	1b[5]	10b[5]	9b[5]	8b[5]
	12	8b[6]	7b[6]	6b[6]	5b[6]	4b[6]	3b[6]	2b[6]	1b[6]	10b[6]	9b[6]

Table 8.1: **Parallel Multiplication Allocation Table**

time will be approximately the number of bands of the rotating matrix at a node multiplied by the number of stationary bands at a node and by the time it takes to multiply a single band divided by the number of node processing arrays. This has been summarised for a system with ξ nodes and three $p \times p$ processing arrays per node performing the matrix product $\mathbf{C} = \mathbf{A}.\mathbf{B}$ where $\mathbf{A}$ has dimensions $\psi p \times Q$ and $\mathbf{B}$ has dimensions $Q \times \varrho p$, in Equation 8.3. It is assumed that there are more matrix band partitions than there are processing arrays, and that matrix $\mathbf{B}$ has closer to, but less than or equal to, a multiple of three partitions per node than matrix $\mathbf{A}$.

$$T_{\text{parallel multiply}} \approx \max\left(T_{\text{transmit}}, T_{\text{parallel multiply}} \times \left\lceil \frac{\varrho}{3} \right\rceil\right) \times \frac{\psi}{\xi} \tag{8.3}$$

The transmission time is simply the inverse of the internode bandwidth multiplied by the size of a band partition and then added to the internode latency, β. The size of the band partition is the length of the partition, which is Q, multiplied by the width of the partition, which is p. Equation 8.4 summarises this.

$$T_{\text{trasmit}} = \frac{Qp}{\text{bandwidth}} + \beta \tag{8.4}$$

The computation time is derived from the memory load time, and is the cycle time plus any memory latencies. A full matrix band partition will only fit completely into a cache memory if the 'Q' dimension is less than the cache size divided by the band partition width p. For a one MWord cache and 32×32 processing array, the maximum length band partition that will fit into the cache is

$$\begin{aligned} Q_{\text{max cache}} &= \frac{1048576}{32} \\ &= 32768 \end{aligned}$$

In general, the matrices that will be multiplied on a multiprocessor will be larger than this, so cache reuse will be at a minimum. Therefore, cache data streaming is almost essential, as is careful placement of the data in memory. The computation time for a band partition can be approximated by Equation 8.5, where 'D' is the cache block size and 'τ' is the load/store cycle time.

$$T_{\text{compute}} \approx Q \times p \times \tau + \frac{Q \times p}{D} \times T_{\text{mem latency}} \tag{8.5}$$

Equation 8.3 can be expressed in more detail, as shown in Equation 8.6.

$$T_{\text{parallel multiply}} \approx \max\left(\left(\frac{Qp}{\text{bandwidth}} + \beta\right), \left(Qp \times \tau + \frac{Qp}{D} \times T_{\text{mem latency}} \times \left\lceil \frac{\varrho}{3} \right\rceil\right)\right) \times \frac{\psi}{\xi} \qquad (8.6)$$

This equation was evaluated for a range of selected matrix sizes on various sized multiprocessor configurations. Because there are three degrees of freedom in the sizes of matrices that can be evaluated, namely the P, Q & R values, the range of matrix sizes that can be simulated becomes very large. To simplify the simulation process, both the **A** and **B** matrices were chosen to be square, and also to be the same size. However, the performance should be similar in the event that this is not the case.

The performance in FLOPs (Floating Point Operations Per second) is shown in Figures 8.5a) through c) for internode latencies of 100 nanoseconds, 1000 nanoseconds and 10000 nanoseconds respectively. They show that, once the matrix becomes a size large enough to be effectively handled by the multiprocessor system (minimal idle processors), the processing arrays achieve a sustained speed that no longer increases with matrix size. Note that the estimate assumes that a band will always be fetched from main memory, so that it underestimates the performance for small matrices.

Gauss-Jordan Elimination

The procedure for Gauss-Jordan elimination on a multiprocessor is the same as for a single processor, except that the update phase is processed using all the processing nodes. To implement the algorithm efficiently, the input matrix **A** must be partitioned and distributed evenly among the nodes. The most obvious partitioning is to divide the matrix into as many columns as there are nodes. However, as Gaussian decomposition zeros a column at a time, the node containing the first column is only used once, the node containing the second column is only used twice, etc., which is a very inefficient use of the computational power available due to processor utilisation mismatch.

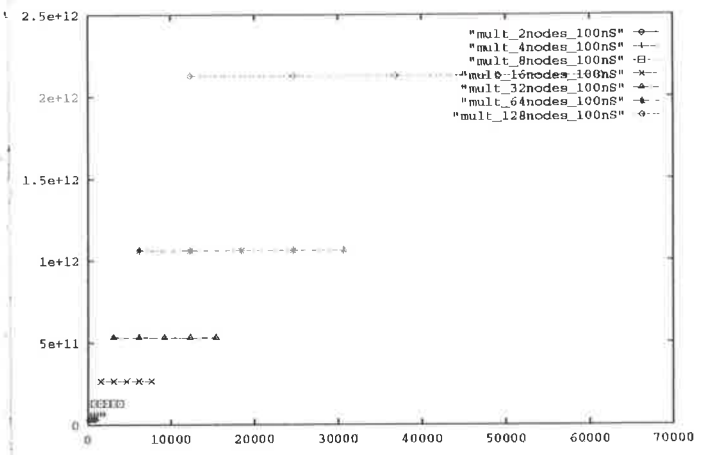
Instead, distribute the rows among the nodes, and then operate on the columns within the rows stored at a node. Gauss-Jordan elimination for the multiprocessor becomes the routine shown in Figure 8.6, which provides good processor utilisation matching. The $P \times R$ matrix is divided into 'ψ' $p \times R$ row partitions, which each contain 'ϱ' $p \times p$ column partitions.

If the rows are distributed among the nodes in groups the same size as the number of processing arrays at a node, the number of required nodes will increase only when the efficiency of doing so justifies the inclusion. Therefore, on a multiprocessor with ξ nodes and three processing arrays at each node, row partitions $1, 2, 3, 3\xi+1, 3\xi+2, 3\xi+3, \ldots$ are placed at node 0, partitions $4, 5, 6, 3\xi+4, 3\xi+5, 3\xi+6, \ldots$ at node 1, etc. The node allocation function for the 'i^{th} row partition of a matrix will be

$$\text{alloc}(i) = \left\lfloor \frac{i-1}{3} \right\rfloor \bmod \xi \quad 1 \leq i \leq m \qquad (8.7)$$

Performance Estimation for Gauss-Jordan Elimination

A simple estimate of the time to complete a Gauss-Jordan elimination on a hypercube can be made. Like the estimates for matrix multiplication, the example performances are provided only for selected sizes of problems. However, the sizes are less critical in this case, as there will always be processor mismatch due to the reduction nature of Gaussian elimination.



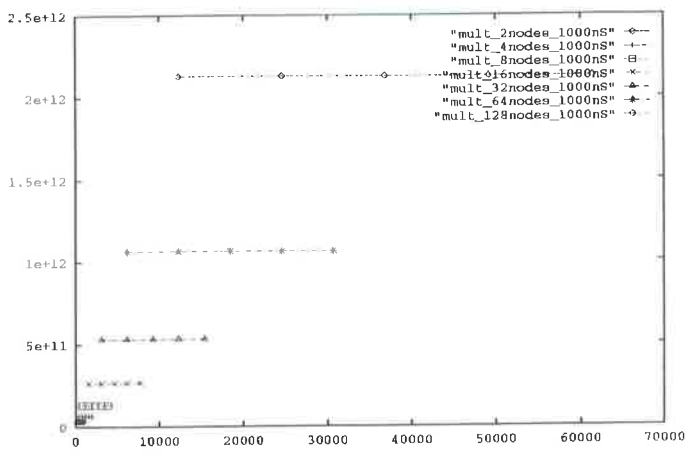


Figure 8.5: a) **100ns Internode Delay** b) **1000ns Internode Delay**

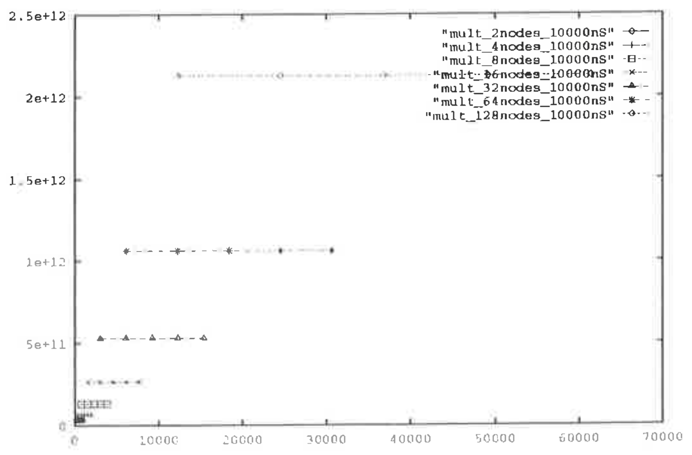


Figure 8.5: c) **10000ns Internode Delay**

Figure 8.5: **Performance Estimates for Multiprocessor Multiplication**

```
for $k = 1$ to $\psi$
    $\hat{A}_{kk} = A_{kk}^{-1}$
    for j=k+1 to $\varrho$
        $\hat{A}_{kj} = A_{kj}\hat{A}_{kk}$
    **broadcast** pivot row
    for **each processing array**
        for **each row** $\hat{i}$ *in a partition*
            for $j = k + 1$ to $\varrho$
                $\hat{A}_{\hat{i}j} = A_{\hat{i}j} - A_{\hat{i}k}\hat{A}_{kj}$
```

Figure 8.6: **Distributed Row-wise Gauss-Jordan Elimination**

The time to perform a Gaussian decomposition such as Gauss-Jordan elimination can be estimated from the time to complete all the jobs in the serial task list. The jobs within the task list may be done in parallel, but at several points throughout the procedure, all the processors must wait on a single processor. Such a case in Gauss-Jordan elimination is when the processors in the node in which the pivot row is held must wait for the inverse of the pivot block to be calculated before the pivot row can be normalised. Additionally, all other processing nodes must wait on the pivot row to be normalised before the other rows can be updated. Therefore, the inherent serial nature of the algorithm constrains the maximum parallel efficiency and hence performance speed up.

The time estimate is based on the serial procedure:

1. Invert the pivot block A_{kk}
   All processors except the one performing the inverse are idle at this stage.

2. Normalize the pivot row
   All processors within a node can be active for this operation as the row can be distributed throughout the node. All processors at other nodes are idle.

3. Broadcast the normalized pivot row
   This is a communication phase, the time for completion of which is dependent on the broadcast parameters in Equation 8.1. The broadcast need not be included in its entirety as the broadcast can commence as soon as the first pivot row element has been calculated, while the update phase can start as soon as the pivot row has been received at a node. However, for the sake of argument, the inclusion of the broadcast time in the serial procedure will cause a minimum bound on the expected performance, and hence a more exact calculation can only get better.

4. Update all blocks in all other rows at each node
   This part is the most parallel part of the algorithm, as all processors are in use all the time. This is the main advantage of Gauss-Jordan elimination for a parallel architecture over other matrix reduction schemes, as there will always be a constant number of rows being operated on. In fact, work at Caltech has shown that Gauss-Jordan elimination, traditionally considered the least efficient algorithm for matrix solution, can solve a system faster than many other algorithms because of its very good processor usage properties[36]. If there is only a single row in a node, then the row can be divided among the processing arrays at a node, and all processors operate on the same row partition. If the number of row partitions at a node is a multiple of the number of processing arrays in a node (nominally three), then each processing array can exclusively operate on a row partition.

5. Repeat the procedure above until all columns are reduced

The time to invert a block stored in main memory will be the same as Equation 7.8. The following normalisation phase will be one third the number of column partitions in a row partition multiplied by the time for a single block size multiply. The time to perform the block size multiply is the time for the node with the largest number of row partitions (there could be several with the same number) to update its rows. The update time is twice the multiply accumulate time for a block sized multiply multiplied by the number of blocks stored at a node divided by three. The factor of two is present because the original block must be loaded into the array before it is updated. The factor of three is due to there being three processing arrays at a node. The total time is the time for all iterations on a decreasing sized matrix.

Because the matrix size is reducing in one dimension only (the number of rows operated on per column is always constant; only the number of columns reduces), the decreasing matrix size can be replaced by a constant sized matrix with the dimension of the average of the decreasing matrix dimensions for the sake of time calculations. Equation 8.8 summarizes the times for a $P \times R$ matrix on a multiprocessor with ξ nodes, each with three processing arrays of size 32×32. The number of row partitions is denoted by ψ and the number of column partitions is denoted by ϱ, such that $P = 32\psi$ and $R = 32\varrho$.

$$\begin{aligned}
T_{\textit{Par. Gauss-Jordan}} &= \left(T_{\text{pivot}} + T_{\text{normalise}} + T_{\text{broadcast}} + T_{\text{update}}\right) \times (\textit{no. of row partitions}) \\
&= \left(T_{\text{pivot}} + T_{\text{block mult}} \times \frac{(\varrho-1)}{2}/3\right. \\
&\quad \left. + \left(\sqrt{\frac{L\tau_c}{m}} + \sqrt{m\beta}\right)^2 + 2\left\lceil\frac{\psi}{\xi}\right\rceil T_{\text{block mult}} \times \frac{(\varrho-1)}{2}/3\right) \times \psi \\
&= \left(T_{\text{pivot}} + \left(\sqrt{\frac{L\tau_c}{m}} + \sqrt{m\beta}\right)^2 + \right. \\
&\quad \left. T_{\text{block mult}} \times \frac{(\varrho-1)}{6}\left(1 + 2\left\lceil\frac{\psi}{\xi}\right\rceil\right)\right) \times \psi
\end{aligned} \tag{8.8}$$

The number of necessary operations is still the same as for a single processing array, ie $ops \approx n^3 - 1.5n^2 + 0.5n$ for an $n \times n$ matrix ($P = R = n$ in this case). This leads to an expression for system performance, which is given in Equation 8.9.

$$Perf_{\textit{Par. Gauss-Jordan}} \approx \frac{n^3 - 1.5n^2 + 0.5n}{T_{\textit{Par. Gauss-Jordan}}} \tag{8.9}$$

Equation 8.9 was evaluated for systems with between four and 128 processing nodes, each with three processing arrays, with varying internode latencies. The results for internode latencies of 100ns, 1000ns and 10000ns are shown in Figures 8.7a) to 8.7c) respectively.

8.2 Wavelet Processor

This is a simple description of a systolic array suitable for performing a compact wavelet transformation. The impetus for this came while investigating applications for the matrix engine, due to the matrix multiplication nature of a wavelet transform. However, although some forms of the wavelet transform are suitable to implement on the matrix engine [52], the compact wavelet transforms deserve a more specialised architecture.

Similarly to the Fourier Transform, a Wavelet Transform translates a vector of components in one domain (eg time) into an equally sized vector in another domain (eg frequency). Unlike the Fourier Transform, for which the translation operators are sines and cosines, which are localized in frequency but infinite in time, the operators for the Wavelet Transform are localised in both time and frequency.

The translation matrix for the compact Daubechies transform is a cyclic, nearly-band matrix. The dimension of the matrix *must* be a power of two. Such a matrix is shown in Figure 8.8 (from [73]) for the case of four coefficients. The coefficients are chosen such that the odd rows of the transform matrix produce a 'smoothing' effect on the input vector, while the even rows attempt to zero, or 'decimate', the response [19].

The coefficient matrix can be modified to remove the cyclic wrap-around condition, or rather to help visualise the method of coping with the fact that the last wavelet coefficients are affected

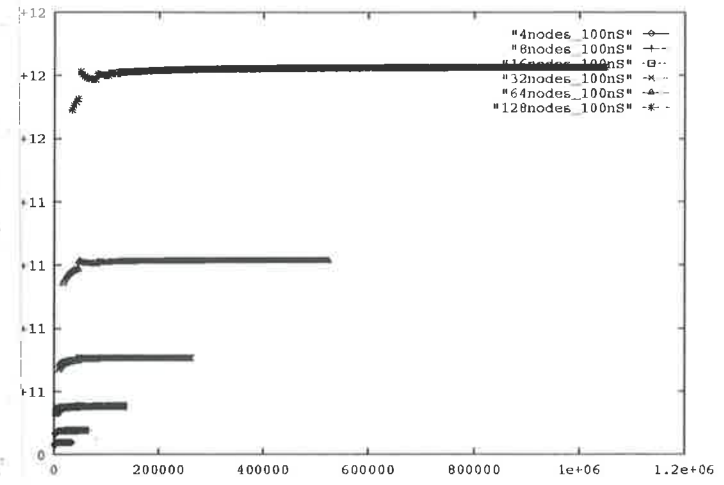
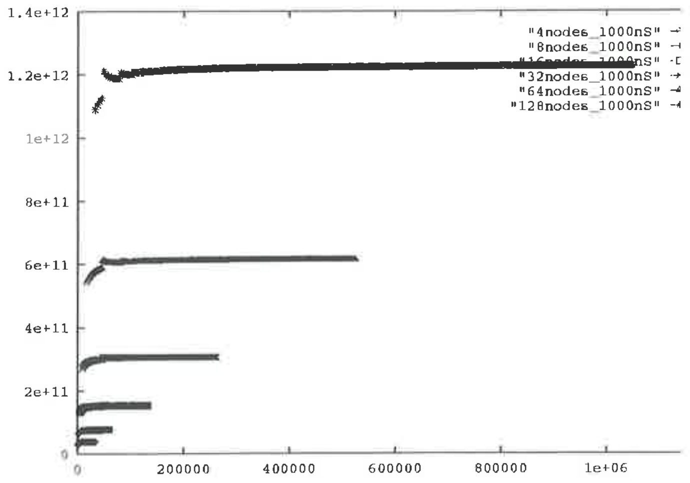


Figure 8.7: a) **Latency = 100ns**



b) **Latency = 1000ns**

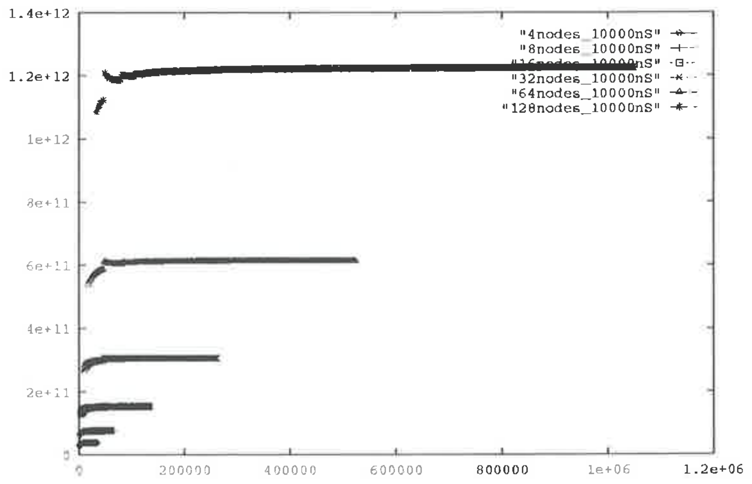


Figure 8.7: c) **Latency = 10000ns**

Figure 8.7: **Performance Estimates for Multiprocessor Gauss-Jordan Elimination**

$$\begin{pmatrix}
c_0 & c_1 & c_2 & c_3 & & & & & & & \\
c_3 & -c_2 & c_1 & -c_0 & & & & & & & \\
 & & c_0 & c_1 & c_2 & c_3 & & & & & \\
 & & c_3 & -c_2 & c_1 & -c_0 & & & & & \\
\vdots & \vdots & & & & & \ddots & & & & \\
 & & & & & & & c_0 & c_1 & c_2 & c_3 \\
 & & & & & & & c_3 & -c_2 & c_1 & c_0 \\
c_2 & c_3 & & & & & & & & c_0 & c_1 \\
c_1 & -c_0 & & & & & & & & c_3 & -c_2
\end{pmatrix}$$

Figure 8.8: **Four Coefficient Compact Wavelet Transformation Matrix**

by both ends of the data vector. The new matrix becomes the one shown in Equation 8.10. According to Press *et. al.* [73], it is possible to eliminate the wrap-around completely, leaving a purely band-diagonal coefficient matrix without changing the size of the matrix, although they don't go into any detail on this point.

$$\begin{pmatrix} s_1 \\ d_1 \\ s_2 \\ d_2 \\ \vdots \\ s_{n/2} \\ d_{n/2} \end{pmatrix} = \begin{pmatrix}
c_0 & c_1 & c_2 & c_3 & & & & & & & & & \\
c_3 & -c_2 & c_1 & -c_0 & & & & & & & & & \\
 & & c_0 & c_1 & c_2 & c_3 & & & & & & & \\
 & & c_3 & -c_2 & c_1 & -c_0 & & & & & & & \\
\vdots & \vdots & & & & & \ddots & & & & & & \\
 & & & & & & & c_0 & c_1 & c_2 & c_3 & & \\
 & & & & & & & c_3 & -c_2 & c_1 & c_0 & & \\
 & & & & & & & & & c_0 & c_1 & c_2 & c_3 \\
 & & & & & & & & & c_3 & -c_2 & c_1 & -c_0
\end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{n-1} \\ x_n \\ x_1 \\ x_2 \end{pmatrix} \tag{8.10}$$

One property is that the transforms of the odd rows are stored in the top half of the output vector and the even rows are stored in the bottom half. Thus the top half contains the 'smooth' components and the bottom half contains the 'zero'ed components. The 'smooth' component of the output vector are then recirculated again in a pyramidical fashion, smoothing to half the components and decimating the other half until only one 'smooth' component remains. The text by Press *et. al.* provides a good graphical description of the transform and permute procedure, similar to Figure 8.9 below.

Of course, there will be no actual 'permute' stage, as this is simply a storage routine. The address pattern for loading will be

$$prev\_addr + 1$$

while the pattern for storing will be

$$\begin{array}{rl} prev\_addr + \frac{n}{2} & \text{for even rows} \\ prev\_addr - \frac{n}{2} + 1 & \text{for odd rows} \end{array}$$

or even more simply, two counters can be used, the first starting at one, the second at $\frac{n}{2} + 1$.

Although the compact Daubechies transform does not map efficiently onto the systolic matrix array (due to the large number of zero entries), a different systolic array can exploit the repetitive

$$\begin{pmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \\ x8 \end{pmatrix} \xrightarrow{transform} \begin{pmatrix} s1 \\ d1 \\ s2 \\ d2 \\ s3 \\ d3 \\ s4 \\ d4 \end{pmatrix} \xrightarrow{permute} \begin{pmatrix} s1 \\ s2 \\ s3 \\ s4 \\ d1 \\ d2 \\ d3 \\ d4 \end{pmatrix} \xrightarrow{transform} \begin{pmatrix} s1' \\ d1' \\ s2' \\ d2' \\ d1 \\ d2 \\ d3 \\ d4 \end{pmatrix} \xrightarrow{pemute} \left(\begin{array}{c} s1' \\ s2' \\ d1' \\ d2' \\ \hline d1 \\ d2 \\ d3 \\ d4 \end{array}\right) \xrightarrow{transform} \left(\begin{array}{c} s1'' \\ d1'' \\ \hline d1' \\ d2' \\ \hline d1 \\ d2 \\ d3 \\ d4 \end{array}\right)$$

Figure 8.9: **Pyramidical Transform and Permute Procedure**

nature of the algorithm, the use of constant matrix coefficients and the sparsity of the matrix. As the bandwidth of the matrix is always 'small' compared to the matrix size, then the dimension of the array can be reduced by one, ie to a linear array the same length as the number of coefficients.

If the array is preloaded with the coefficients, then the input vector can move from one end of the array to the other, with the 'smoothed' part of the output vector moving back to the input end and the 'decimated' part of the output vector moving in the same direction as the input vector. A cell from such an array would have the form shown in Figure 8.10.

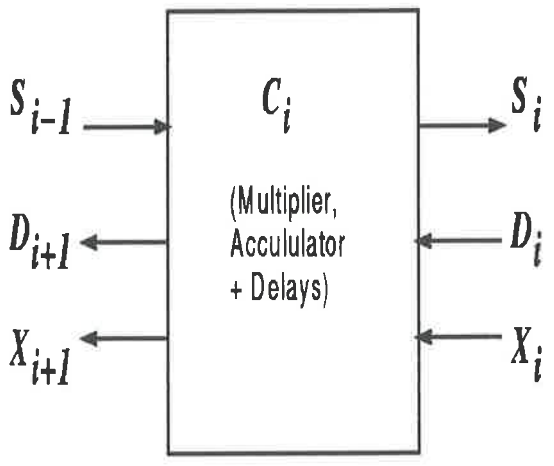


Figure 8.10: **Cell from a Linear Array for the Wavelet Transform**

A block diagram of a cell in the the linear array is given in Figure 8.11, showing the array, the input and output vectors, and the dual ported RAMs used for recycling the smoothed data.

Then if matrix coefficients are in ascending order from left to right and the input vector is applied from the right hand side, the inputs propagate to the left and ultimately all end in the output vector in the correct form. Because of the ordering in the systolic array of the matrix coefficients, the 'smooth' components will all be computed simultaneously. Therefore, coefficients 1 and 2 can be added, while all the others are delayed by suitable elements. The accumulated coefficient products are then shifted right, and accumulated with the next coefficient product, and all elements further to the right are delayed again. This procedure is repeated until the

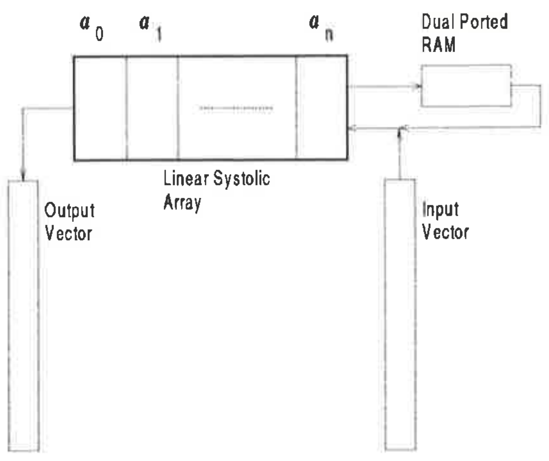


Figure 8.11: **Linear Array for n-dimensional Wavelet Transform**

'smooth' component emerges from the right hand edge of the array[3]

The formation of the decimated components also requires delays, but this time because some products are available before any other products.

Table 8.2 shows the time-step at which each product is calculated for the first ten time-steps, with four coefficients. The first set of products that are summed to produce the first 'smooth' coefficient, s_1, are all calculated in step 4. The required sum is

$$s_1 = C_0X_1 + C_1X_2 + C_2X_3 + C_3X_4$$

If the data is moved to the 'right', then the temporary sum $\dot{s}_1 = C_0X_1 + C_1X_2$ can be calculated immediately. The next temporary sum $\ddot{s}_1 = \dot{s}_1 + C_2X_3$ can be calculated in the next time step, so the product C_2X_3 needs one delay slot. The remaining product can be added one time step later, according to the equation $s_1 = \ddot{s}_1 + C_3X_4$, so two delay slots are required. The decimated components move to the left, in the same direction as the incoming data. The product C_3X_1 is calculated in the first time step, and must have the product C_2X_2 subtracted from it. C_2X_2 is calculated in the third time step, so the result of C_3X_1 must be delayed by two slots. The temporary sum $\dot{d}_1 = C_3X_1 - C_2X_2$ is available after the fourth time step, and must be added to the third product C_1X_3, which is available after the fifth time step, to produce the temporary product $\ddot{d}_1 = \dot{d}_1 + C_1X_3$. Therefore, the temporary sum $\dot{d}_1$ must only be delayed by a single time slot. Similarly, the final 'decimated' result $d_1 = \ddot{d}_1 - C_0X_4$ also requires that the temporary sum $\ddot{d}_1$ only be delayed by a single slot. The reason only the first product term requires two delay slots instead of one is that it not passed through an accumulator or adder. Therefore, if all the cells in the array were created exactly alike, and the first product was accumulated with 'zero', then only a single delay slot would be required.

With these requirements in mind, it is a simple matter to produce a systolic cell capable of performing a compact Daubechies wavelet transform. Such a cell is shown in Figure 8.12. An arbitrary number of these cells can be butted together to form a linear systolic array capable of

[3]Note that a binary tree would have been a more logical approach to implementing the accumulation of all the components that are available simultaneously. However, the binary tree does not lend itself to a simple, expandable systolic array (one branch structure will always be wasted) and will lose the compactness of a linear array.

Time (steps)	Cell 0 (data)	Cell 1 (data)	Cell 2 (data)	Cell 3 (data)
1	-	-	-	C_3X_1
2	-	-	C_2X_1	C_3X_2
3	-	C_1X_1	C_2X_2	C_3X_3
4	C_0X_1	C_1X_2	C_2X_3	C_3X_4
5	C_0X_2	C_1X_3	C_2X_4	C_3X_5
6	C_0X_3	C_1X_4	C_2X_5	C_3X_6
7	C_0X_4	C_1X_5	C_2X_6	C_3X_7
8	C_0X_5	C_1X_6	C_2X_7	C_3X_8
9	C_0X_6	C_1X_7	C_2X_8	C_3X_9
10	C_0X_7	C_1X_8	C_2X_9	C_3X_10

Table 8.2: **Wavelet Products versus Time**

performing a wavelet transform of any degree within the confines of the algorithm (there must be an even number of coefficients).

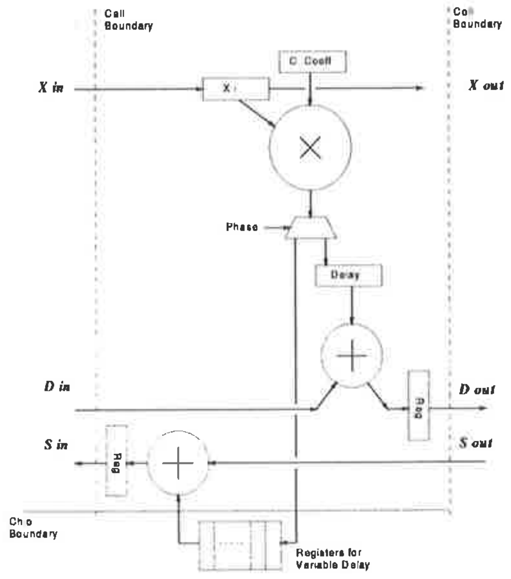


Figure 8.12: **Single Cell for Wavelet Implementation**

To 'undo' the wavelet transform and reproduce the original input vector, $\tilde{x}$, then the vector of wavelet values, denoted $\tilde{w}$, needs to be multiplied by the inverse of the wavelet coefficient matrix, $\mathbf{C}$,ie

$$\tilde{x} = \mathbf{C}^{-1}\tilde{w}$$

Recalling that the compact Daubechies wavelet transform is a transformation on an orthonormal basis, ie all the rows and columns in $\mathbf{C}$ are orthogonal, then the inverse matrix $\mathbf{C}^{-1}$ is simply the transpose of the same matrix $\mathbf{C}^{-1} = \mathbf{C}^{T}$. As the transposed matrix is nearly identical to the original matrix in form, exactly the same hardware can be used for the inverse transform as was used for the original transform.

8.2.1 Still Picture Compression

Because the Wavelet Transform produces large coefficients for regions of large change, or high contrast, and small coefficients for regions of small change, or low contrast, compression by maintaining only the large components will result in minimal noticeable loss in image quality.

Additionally, as the Wavelet Transform is localised in both space and time, then the effects of omitting a coefficient are localised as well. This is different to omitting Fourier or cosine coefficients, which are not localised, and hence produce 'ringing' when components are omitted. For this reason, Fourier based image compression generally uses 'blocking', whereby the image is divided into many small blocks (typically 6×6 to 16×16 pixels) and the lossy compression contained within the blocks. This in itself is another cause of poor image compression, as the blocks themselves become noticeable for large compression ratios.

8.2.2 Moving Picture Compression

The MPEG standard (Moving Pictures Experts Group) includes both inter- and intra-frame compression. These compressions are between successive images and within an image respectively. Therefore, the amount of computational power required is much greater than for single image compression. However, as the images will not, in general, be analysed as critically as a still image, the compression ratios can be much higher, considerably reducing bandwidth [2].

Similarly, by comparing the difference between successive compressed wavelet transformed images, a system can be constructed that uses wavelet compression on moving pictures with a highly reduced required bandwidth and a less noticeable loss of quality

Variable Quality Image Compression

As the number of coefficients used for the Wavelet Transform is an arbitrary even number and the systolic array for processing the transform is linear, the number of coefficients is dynamically reconfigurable. Additionally, the threshold of compression can be changed between successive images. Therefore, it is relatively simple to create a system that transmits a reduced quality image for the majority of its use, while being capable of transmitting high quality images with an increased delay if required. Such a system would be of great use in applications such as video telephones and mobile video communication, where the bandwidth is limited. The standard image of reduced quality could be tailored to use the full available bandwidth for real time motion, while a high quality image (such as a photograph or detailed diagram) could be sent with a delay.

This section on a systolic wavelet processor is intended to demonstrate the ease with which many algorithms can be converted into systolic arrays, not on the image compression and coding itself. For more information on compression, see [73, 52, 53] and other authors.

8.3 Conclusion

The ability to parallelize many matrix operations has been used to show that a multiprocessor version of the matrix engine is possible, with estimated performance figures in the region of tens to hundreds of GigaFLOPS for Gauss-Jordan elimination and approaching a TeraFLOP for matrix multiplication on an appropriately sized machine for a large enough problem size. These figures open a whole realm of modelling in fields such as airflow and fluid-flow that was previously infeasible, such as the automated optimisation of aircraft, vehicle or ship profiles involving the repeated solution of matrices of the order of $20,000 \times 20,000$ to $1,000,000 \times 1,000,000$.

The systolic process was also extended to the design of a linear systolic array for the computation of the Daubechies Discrete Wavelet Transform (DWT). This transform shows great potential for use in lossy image compression, and as such a systolic array capable of perform the computations simply and with a high bandwidth would be of great use. Applications such as pay-TV (cable-TV) and video conferencing would benefit greatly by a reduction in the amount of data that is required to be transmitted with minimal loss in quality, and that can be modified dynamically to match a required image quality.

Chapter 9

Summary and Conclusion

In this thesis, the implementation of a systolic processor interface has been introduced and solutions proposed for the many difficulties encountered in such a task. The design task has included several separate but related tasks, including

1. The VLSI design of an extended version of Marwood's Address Generator.
2. The construction and development of algorithms to be implemented on the matrix engine.
3. A discussion of arithmetic components used to optimise the speed of the system.
4. The design of a memory subsystem to support the high computational rate of the processing array.

For comparison purposes, consider the SCAP processor particulars that were presented in Section 2.2. Although the SCAP system has now been updated and is using a slower technology than that used here, a point by point comparison is still interesting. This is shown in itemised form below.

- Not only does the data controller presented here contain a interface to memory from the array, it includes cache management and tag checking in the interface standard, which allows the construction of fast multi-level memory systems at a reasonable cost.
- The data controllers are cascadable to an arbitrary size. In fact, if a multichip solution is used with the address generation and tag checking hardware on one chip and the data registers and sundry arithmetic on another, only the data chip need be replicated for expansion, as only one address generator per interface is required.
- The bus interface will allow common or independent memory subsystems. However, unlike the SCAP system, a common memory system must allow some form of bank switching or multi-access memory, as the interface depends that each controller can access memory independently.
- As the address generator is a Marwood Difference Engine that has been extended to four dimensions, the same zero cost operations available on SCAP are also available, as well as zero cost partitioning and numerical algorithms.
- There is no direct support for complex matrices, although these can be supported in software. Sub-matrices and non-square matrices are supported via the difference engine.

- Arbitrary sized matrices are directly supported. The four dimensional difference engine will allow the data controller to automatically partition any applied matrix to fit on the fixed sized processing array.
- The data controller can be used to either load or store results.
- Allocating a small amount of instruction memory in the Harvard architecture style will allow the data controllers to execute their own instruction streams independently from a host system, allowing complicated algorithms to be performed in their entirety by the matrix processing system.
- Each data controller can fetch or store operands at a rate of 100M floating point words per second, memory subsystem permitting.

A block diagram of the complete system is shown in Figure 9.1.

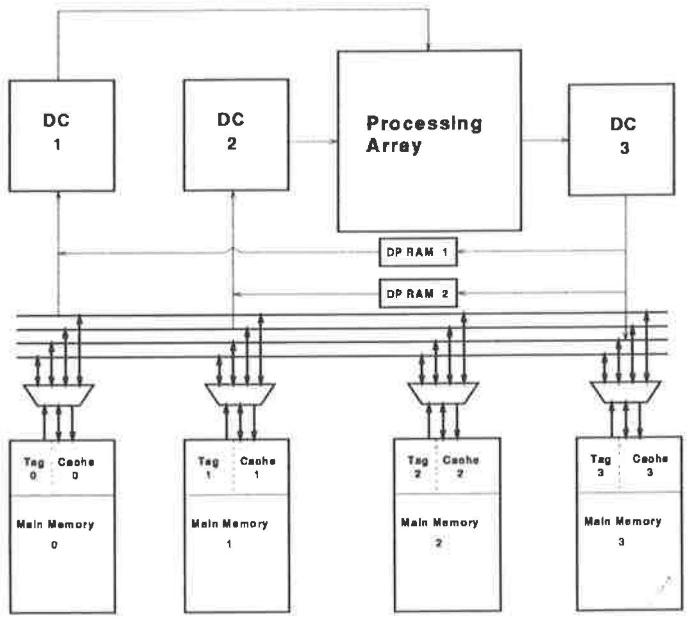


Figure 9.1: **Complete System Block Diagram**

The system is capable of speeds only currently available using expensive supercomputer technology, but instead requiring only CMOS technology and a much smaller physical area. A Multi-Chip Module containing an array capable of three to five sustained GigaFLOPs would be hand-holdable, and a system built around the array could fit on a single printed circuit board. Because of the small size of each processing array, multiprocessors constructed with the order of one hundred processing arrays are viable, which puts the realm of TeraFLOP computing within grasp. The possibility of connecting 100 CRAY-2 supercomputers together is minimal, even in terms of the physical area that would be consumed, whereas one hundred systolic processing arrays connected together is not infeasible.

Although the range of applications that can be solved on the processing array is limited, suitable applications typically form the basis for computations that consume a great deal of available computing power with little user interaction. For example, large air flow and fluid flow simulations that are well suited to our design may take days to run on current vector supercomputers. The impact of improvements in this field will be resoundly felt.

Appendix A

SCAP Data Sheets

Pages 174 to 178
Removed
for reasons of commercial confidentiality

Appendix B

Proof of Convergence for Iterative Matrix Inversion

B.1 Initialising the Pivot Inverse Iteration

The *residual matrix* **R** is defined as the difference between the *actual* inverse and the initial *approximate* inverse. If it is noted that the product of the *actual* inverse (call this $\hat{Y}$) and the matrix for which the inverse is sought (ie **A**) is the identity matrix **I**, then the residual matrix is the difference between the identity **I** and the product of $\mathbf{Y}_0$ and **A**. This is shown in Equation B.1.

$$\mathbf{R} \equiv \mathbf{I} - \mathbf{Y}_0.\mathbf{A} \tag{B.1}$$

This leads to the (infinite) series

$$\begin{aligned} \mathbf{A}^{-1} &= (\mathbf{I} - \mathbf{R})^{-1}\mathbf{Y}_0 \\ &= (\mathbf{I} + \mathbf{R} + \mathbf{R}^2 + \mathbf{R}^3 + \cdots).\mathbf{Y}_0 \end{aligned} \tag{B.2}$$

From Equation B.2, for convergence, the norm of **R** must satisfy

$$||\mathbf{R}|| < 1 \tag{B.3}$$

where the norm is defined to be the largest amplification of length that the matrix is able to induce on a vector [73], ie

$$||\mathbf{R}|| \equiv \max_{\mathbf{v} \neq 0} \frac{|\mathbf{R}.\mathbf{v}|}{|\mathbf{v}|} \tag{B.4}$$

Press *et al.* [73] provides a good analysis on the work of Pan & Reif [71] for choosing a good initial guess. They point out that a suitable choice is a sufficiently small constant, ϵ, multiplied by the transpose of the matrix for which the inverse is desired, ie

$$\mathbf{Y}_0 = \epsilon\mathbf{A}^T \tag{B.5}$$

If Equation B.5 is to satisfy the requirement from Equation B.3, then it can be shown [73] that **R** is of the form

$$\mathbf{R} = diag(1 - \epsilon\lambda_1, 1 - \epsilon\lambda_2, \ldots, 1 - \epsilon\lambda_N) \tag{B.6}$$

where λ_i are the eigenvalues of $\mathbf{A}^T.\mathbf{A}$.

Pan & Reif [71] point out that the vector norm requirement in Equation B.4 need not be the Euclidean, or l_2, norm, but could be either the l_∞ norm or the l_1 norm. These lead to the two possible choices for ϵ, given below:

$$\epsilon \leq \sum_{j,k} a_{jk}^2 \qquad \text{or} \qquad \epsilon \leq \frac{1}{\max_i \sum j|a_{ij}| \times \max_j \sum_i |a_{ij}|} \tag{B.7}$$

which are much easier to calculate than the eigenvalues.

A further simplification can be made by choosing a different matrix norm from Equation B.4. Three other common norms include the "Frobenius" norm

$$||\mathbf{R}||_F = \sqrt{\sum_i \sum_j r_{ij}^2} \tag{B.8}$$

, the "column-sum" norm

$$||\mathbf{R}||_1 = \max_j \sum_i |r_{ij}| \tag{B.9}$$

, and the "row-sum" norm

$$||\mathbf{R}||_\infty = \max_i \sum_j |r_{ij}| \tag{B.10}$$

any of which may be satisfied to guarantee convergence.

Recalling that Gaussian elimination[1] is stable only for a diagonally dominant matrix ($|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$), then the row-sum norm of the residue matrix can be made less than one if the initial estimate of the inverse ($\mathbf{Y}_0$) is set to

$$\mathbf{B}_0 = diag(\frac{1}{a_{11}}, \frac{1}{a_{22}}, \ldots, \frac{1}{a_{nn}}) \tag{B.11}$$

To see this, consider the row-sum norm of the residue matrix.

$$\begin{aligned}
||\mathbf{R}||_\infty &= \mathbf{I} - \mathbf{Y}_0\mathbf{A} \\
&= \left\| \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} - \begin{pmatrix} \frac{1}{a_{11}} & 0 & \cdots & 0 \\ 0 & \frac{1}{a_{22}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{a_{nn}} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \right\| \\
&= \left\| \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} - \begin{pmatrix} 1 & \frac{a_{12}}{a_{11}} & \cdots & \frac{a_{1n}}{a_{11}} \\ \frac{a_{21}}{a_{22}} & 1 & \cdots & \frac{a_{2n}}{a_{22}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{a_{n1}}{a_{nn}} & \frac{a_{n2}}{a_{nn}} & \cdots & 1 \end{pmatrix} \right\| \qquad \text{(B.12)} \\
&= \left\| \begin{pmatrix} 0 & \frac{a_{12}}{a_{11}} & \cdots & \frac{a_{1n}}{a_{11}} \\ \frac{a_{21}}{a_{22}} & 0 & \cdots & \frac{a_{2n}}{a_{22}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{a_{n1}}{a_{nn}} & \frac{a_{n2}}{a_{nn}} & \cdots & 0 \end{pmatrix} \right\| \qquad \text{(B.13)}
\end{aligned}$$

If the row-sum norm from Equation B.10 is used, then the norm of the residue matrix **R** becomes

$$||\mathbf{R}|| = \max_i \sum_{j=1}^{n} |r_{ij}|$$

[1] which includes Gauss Jordan elimination and LU decomposition

$$
\begin{aligned}
&= \max_i \sum_{j=1, j\neq i}^{n} \left|\frac{a_{ij}}{a_{ii}}\right| \\
&\leq \max_i |1| \qquad\qquad |a_{ii}| \geq \sum_{j=1, j\neq i}^{n} |a_{ij}| \\
&\leq ||\mathbf{1}|| \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(B.14)}
\end{aligned}
$$

Therefore, an initial estimate of

$$
b_{ij} = \begin{cases} \frac{1}{a_{ii}} & \text{if } i \equiv j \\ 0 & \text{otherwise} \end{cases} \qquad \text{(B.15)}
$$

will converge to the inverse matrix as $n \to \infty$. [2]

[2] Actually, we hope much sooner than $n \to \infty$. The convergence should take approximately $\log_2 prec$ iterations, where *prec* is the precision in bits of the scalars in the matrix, due to the quadratic nature of the algorithm.

Appendix C

Broadcasting Data on a Hypercube

One of the major bottlenecks with multiprocessor systems is the accessing of data residing in main memory. If the memory is physically distributed throughout the system, data needs to be sent among the nodes by message passing strategies. Two nearest neighbours can communicate directly, but two remote processors can only exchange data by routing a message through intermediate nodes. Although this appears to by detrimental to the performance of the system by interrupting the intermediate node(s), advantage can be made of this mechanism if a broadcast is required, as the intermediate node can not only pass the message on, but keep a copy for itself. The broadcast mechanism is very useful in matrix algorithms such as matrix-matrix multiplication and Gaussian elimination, where complete rows or columns are required simultaneously at all nodes.

The theoretical limit for the performance of a broadcast on a hypercube with 2^m nodes can be obtained by realising that it takes at least $m(\beta + \tau_c)$ time units for the first word to reach the last node (the furthest along the critical path), where β is the latency per word and τ_c is the time to send one word over a nearest-neighbour link. This word can arrive simultaneously with $(m-1)$ others (one per link). After the first word arrives, the last node can receive at most m words per cycle for the remaining $\frac{(L-1)}{m}$ cycles, where L is the number of words that are to be broadcast. Therefore, if the number of packets of length (L/m) is greater than the critical path, the theoretical minimum time for a broadcast is

$$\begin{aligned}
\tau_{optimal} &= m(\beta + \tau_c) + \frac{(L-m)}{m}.\tau_c \\
&= m\beta + \tau_c(m + \frac{(L-m)}{m}) \\
&= m\beta + (\frac{L}{m} + m - 1)\tau_c \qquad \text{(C.1)}
\end{aligned}$$

C.1 Simple Broadcast

If a broadcast function on a hypercube were to simply broadcast from each node to its nearest neighbours, which in turn would broadcast to each of their nearest neighbours, a large amount of redundant data would be transmitted and the control would be difficult. In a system where the communication bandwidth is a critical factor, such a scheme would result in a serious reduction in sustained performance. A much better way to broadcast data in an $m-$cube is to generate a spanning tree with m links, as shown in Figure C.1 for a broadcast from node 0. The tree is embedded in the topology, so the transfer of data from node 0 flows along the heavier lines in Figure C.1.

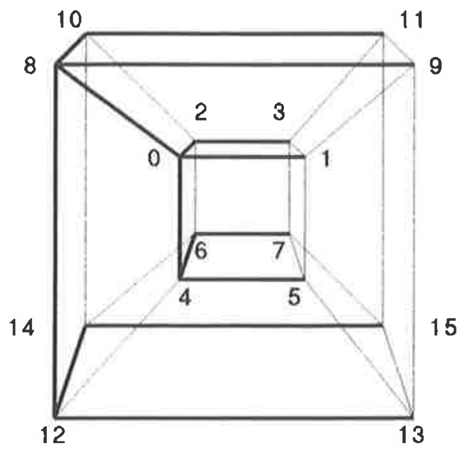


Figure C.1: **Spanning Tree for a 4 Dimensional Hypercube**

The spanning tree uses the same Gray-code that is used in the labelling of the $m-$cube, by receiving on the link that corresponds to the first bit becoming a '1' from a sending node that has the same bit equal to '0'. As the critical path is 'm' links long, the broadcast takes m broadcast time units, with communication over the longest length started first. For a broadcast initiated at node 0, then

1. send data from node 0 to node 8 (first 0-1 at bit position 3)
2. send data from node 0 to node 4, and from node 8 to node 12 (first 0-1 at bit position 2)
3. send data from node 0 to node 2, from node 4 to node 6, from node 8 to node 10, and from node 12 to node 14 (first 0-1 at bit position 1)
4. send data from node 0 to node 1, from node 2 to node 3, from node 4 to node 5, from node 6 to node 7, from node 8 to node 9, from node 10 to node 11, from node 12 to node 13, and from node 14 to node 15 (first 0-1 at bit position 0)

Of course, if the broadcast were to originate from any other node, a bit-wise XOR between the absolute node label and the label of the originating node will produce the correct broadcast table. It is obvious from the above procedure and the corresponding spanning tree that, if the time to transfer 'L' words between two nodes is $\beta + L\tau_c$, where β is the start-up overhead and τ_c is the time to pass one word between nodes, the total broadcast time is given by

$$\tau_{SB} = m(\beta + L\tau_c)$$

C.2 Pipelined Broadcast

A method for improving the throughput of data is to pipeline the data flow [76]. The theory of pipelining is straight forward, with packets of data progressing in a pipelined manner through a single spanning tree. Hence, if a node at distance 'j' from the root node has the first packet of data, then packet 2 will be a distance $j-1$ from the root node, while the j^{th} packet is at the neighbouring nodes of the root node. If the broadcast of length 'L' is broken into 'μ' packets for pipelining, each of size 'L/μ', then the total time for a broadcast is simply the time for the first packet to reach the last node plus the time for the remaining packets to propagate through. Using the simple broadcast times, the time for the first packet of length 'L/μ' to reach the last node is $m(\beta + \tau_c L/\mu)$. The remaining packets reach the last node every $(\beta + \tau_c L/\mu)$ units, and there are $(\mu - 1)$ such packets, so the remaining packets take $(\mu-1)(\beta+\tau_c L/\mu)$ units. Therefore, the total time is $T(\mu) = (m + \mu - 1)(\beta + \tau_c L/\mu)$.

The only variable that can be manipulated is the packet-size μ, as m, τ_c and β are machine dependent and L is problem dependent. Therefore, to minimize the time T_m, separate the components with and without μ and then minimize with respect to μ.

$$T_m(\mu) = ((m-1)\beta + \tau_c L) + \left(\mu\beta + \frac{(m-1)\tau_c L}{\mu}\right) \tag{C.2}$$

$$\dot{T}_m(\mu) = \left(\beta - \frac{(m-1)\tau_c L}{\mu^2}\right) \tag{C.3}$$

Equation C.3 is zero when

$$\mu = \sqrt{\frac{(m-1)\tau_c L}{\beta}}$$

As Equation C.2 is hyperbolic, then an examination of μ end-points ($\mu = 0$ & ∞) shows that

$$\mu_{min} = \sqrt{\frac{(m-1)\tau_c L}{\beta}}$$

is a minimum. Therefore, the optimum (minimum) time for a pipelined broadcast is

$$T_m(\mu) = \left(\sqrt{L\tau_c} + \sqrt{\frac{(m-1)\tau_c L}{\beta}}\right)^2 \tag{C.4}$$

The procedure above assumes that the node to node links can all be activated simultaneously, and that they are unidirectional. If the links are bidirectional, then only half the possible bandwidth will be used, so the modification of Section C.4 should be used.

C.3 Parallel or Rotated Broadcast

The spanning tree in Figure C.1 only uses $2^m - 1$ of the available $m.2^{(m-1)}$ links, which is a considerable waste of resources. To utilise the resources in a more efficient manner, 'm' separate spanning trees can be created. If these are rotated into the $m-$dimensions and the data is split into 'm' pieces, 'm' concurrent broadcasts can proceed, making use of all the links of the hypercube. Figure C.2 shows four spanning trees for a 4-cube.

If 'wormhole' routing is to be used (packets of data 'burrow' through the network in a wormhole fashion), the act of splitting the data into m pieces is a natural requirement, and can be used to our benefit. For a broadcast of 'L' words, the time to broadcast using the rotating broadcast scheme is the same as the time to broadcast L/m words using the first (simple) scheme, as the initial data transfer has been split into m packets of the same length. Therefore, the total time to transfer 'L' words is

$$m(\beta + \tau_c L/m) = m\beta + L\tau_c$$

. Note that the packet-size independent term $L\tau_c$ is not dependent on the dimension of the hypercube, due to the use of all links all the time. The only implementation-size dependent term is the start-up overhead.

C.4 Pipelined and Rotated Broadcast

To pipeline a rotated broadcast strategy[1] is not a simple matter of combining the two approaches, as the spanning trees (eg of Figure C.2) are not edge dependent. This is due to the fact that a pipelined broadcast utilises all links in a spanning tree at an intermediate time (not a start or finish time), which will cause contention over links if another spanning tree is used simultaneously. A new tree called the m-Edges disjoint Spanning Binomial Tree (m-ESBT) was used by Johnson and Ho [42] to fully utilize the links in a hypercube. They constructed an m-ESBT as follows [42, 76]:

1. start with a standard Spanning Binomial Tree (SBT), rooted at node 0.
2. rotate the binary node number left to obtain $(m-1)$ new SBT's[2].
3. to make these SBT's edge-disjoint, negate the $(k-1)^{th}$ bit of each node on the k^{th} spanning tree for $1 \leq k \leq m-1$, and also negate the most significant bit of the original SBT.
4. node 0 is now the root for all the 'm' resulting spanning trees, so the trees can be merged into a single tree of height $m+1$. This tree is the required m-ESBT, and is shown for a 3−cube in Figure C.3.

Using the expression for the time to broadcast for a pipelined broadcast, the new time to broadcast data using an m-ESBT can be determined by simply replacing the length L with the value $\frac{L}{m}$ (the new length sent via each SBT), and the critical path by the new tree height $m+1$. Then the new expression for the time to broadcast is

$$T_{PR}(\mu_{opt}) = \left(\sqrt{\frac{L\tau_c}{m}} + \sqrt{m\beta}\right)^2 \tag{C.5}$$

[1]Or rotate a pipelined one

[2]If a node in the cube is represented by the binary number $b = (b_{m-1}, b_{m-2}, \ldots, b_1, b_0)$, the left rotation operator, $R_l(b)$, is defined to be $R_l(b) = (b_{m-2}, b_{m-3}, \ldots, b_0, b_{m-1})$, while '$k$' rotations is represented by the operator $R_l^k(b) = R_l(R_l^{k-1}(b))$, the '$k^{th}$' SBT; ie the operator $R_l()$ is recursive.

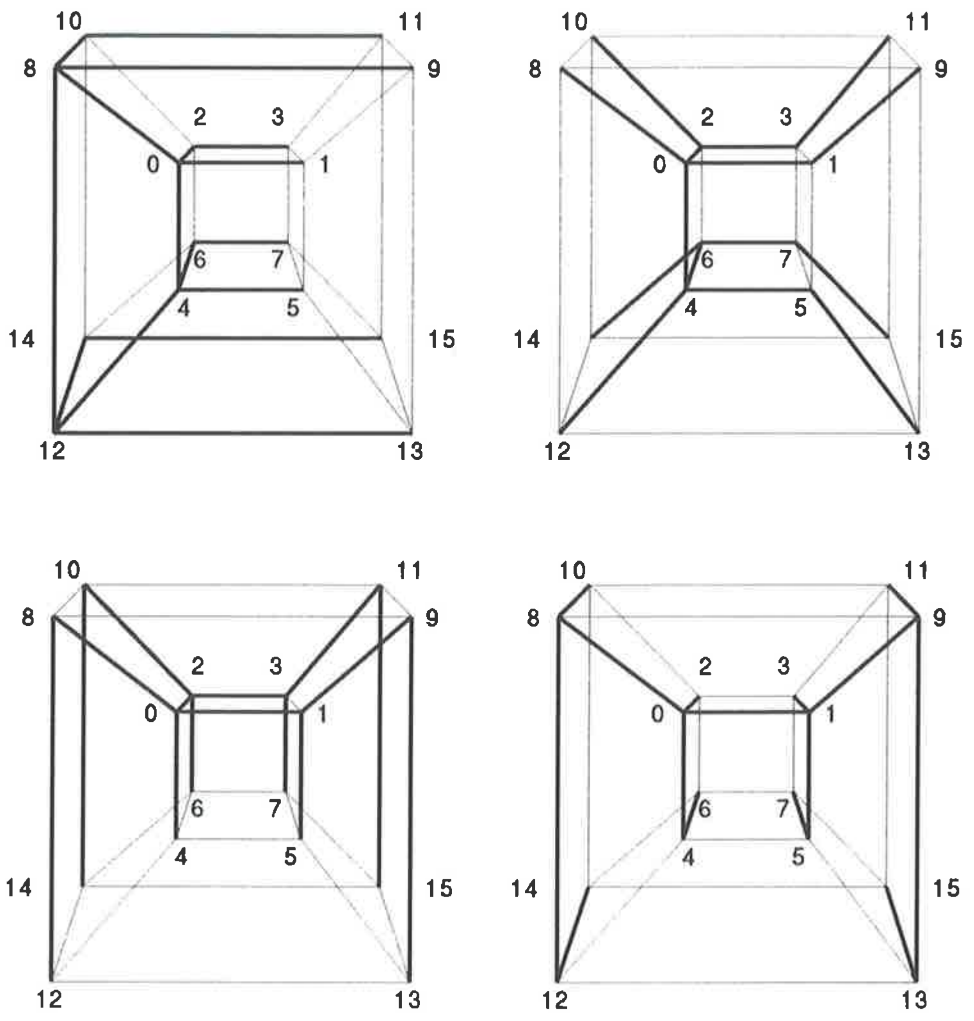


Figure C.2: **4 Rotated Spanning Trees for 4-Dimensional Hypercube**

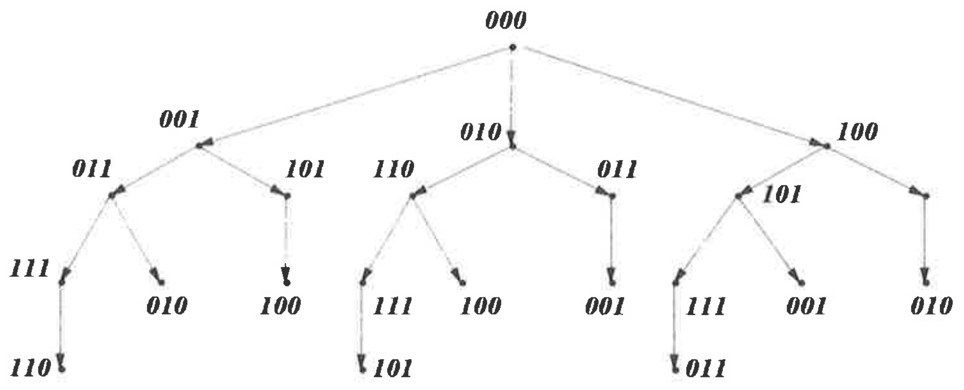


Figure C.3: **3-Dimensional Edge-disjoint Spanning Binary Tree**

Bibliography

[1] Anderson E., Bai Z., Bischof C., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., Ostriuchov S. & Sorensen D.: "LAPACK User's Guide" SIAM Press, Philadelphia, 1992.

[2] Ang P.H, Ruetz P.A. & Auld D.:"Video Compression Makes Big Gains." *I.E.E.E. Spectrum*, Vol 28, No 10, pp 16-19, October 1991.

[3] Avizienis A., "Signed-Digit Number Representations for Fast Parallel Arithmetic." *IRE Transactions on Electronic Computers*, pp 389-400, September, 1961.

[4] Balakrishnan W. & Burgess N.: "Very-high-speed VLSI 2s-complement multiplier using signed binary digits." *IEE Proceedings-E*, Vol 139, No 1, pp 29-34, January 1992.

[5] Beaumont-Smith A., Marwood W. & Eshraghian K.: "The Gallium Arsenide Implementation of a Systolic Floating Point Element." *Proceeding of the 12th IREE Australian Microelectronics Conference*, pp 255-260, October 1993.

[6] Berry M., Gallivan K., Harrod W., Jalby W., Lo S., Meier U., Philippe B. & Sameh A.H.:"Parallel Algorithms on the Cedar System.", in *Proceedings of the International Conference on Parallel Processing and Applications*, pp 25-39, 23-25 September, 1987. Published by Elsevier Science Publishing Co.

[7] Blackley W.S., Lim C.S. & Eshraghian K.: "Architecture for Very High Speed Gallium Arsenide Processing Elements for Matrix Based Computations." *IREE 2nd International Electronics Convention and Exhibition 1989.*

[8] Bode, Dal Chin (eds.):"Parallel Computer Architectures". Lecture Notes in Computer Science no. 732. Published by Springer Verlag, 1993.

[9] Bradley D. & Larson J.:"Fine-grain Measurements of Loop Performance on the Cray Y-MP." CSRD Report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1991.

[10] Briggs W.S. & Matula D.W.:"A 17 × 69 Bit Multiply and Add Unit with Redundant Binary Feedback and Single Cycle Latency." *Proceedings of the I.E.E.E. Eleventh Symposium on Computer Arithmetic*, pp 163-170, June 29-July 2, 1993, Windsor, Ontario.

[11] Burgess N.: Lecture Notes, Department of Electrical & Electronic Engineering, The University of Adelaide, 1994.

[12] Burrus C.S.:"Index Mappings for Multidimensional Formulation of the DFT and Convolution." *IEEE Transactions on ASSP*, Vol 25, pp 239-242, June 1977.

[13] Bursky D.:"Synchronous DRAMs Clock at 100 MHz." *Electronic Design*, pp 45-49, February 18, 1993

[14] Campbell J.K, Synnott S.P. & Bierman G.J.: "Voyager Orbit Determination at Jupiter." in *Kalman Filtering: Theory and Application*, IEEE Press 1985, reprinted from *IEEE Trans. Automat. Contr.*, vol AC-28, pp 256-268, Mar. 1983.

[15] Carey G.F. (editor):"Parallel Supercomputing: Methods, Algorithms and Applications." John Wiley & Sons, 1989.

[16] Carnevali P., Radicati G., Robert Y. & Sguazzero P.:"Block algorithms for Gauss elimination and Householder reduction on the IBM 3090 Vector Multiprocessor." in *Proceedings of the international*

Conference on Parallel Processing and Applications pp 297-302, 23-25 September, 1987. Published by Elsevier Science Publishing Co.

[17] Catanzaro B.:"SPARC MBus Overview." Sun Microsystems, Inc.

[18] Curtis I.A., Clarke R.J., Clarke A.P. & Marwood W. " Data Formatter." *PCT Patent Application*, November 1992.

[19] Daubechies I.:"Orthonormal Bases of Compactly Supported Wavelets." *Communications on Pure and Applied Mathematics* Vol 41, pp 909-996, 1988.

[20] "DECChip^{TM} 21064-AA RISC Microprocessor Preliminary Data Sheet." Digital Equipment Corporation, Maynard, Massachusetts, 1992.

[21] "Introduction to Designing a System with the DECChip^{TM} 21064 Microprocessor." Revision 1.0. Digital Equipment Corporation, Maynard, Massachusetts, 1992.

[22] Dolecek Q.E.:"Parallel Processing Systems for VHSIC." *VHSIC Applications Workshop*, 1984.

[23] Dongarra J.J., Bunch J.R., Moler C.B. & Stewart G.W.:"LINPACK Users' Guide." SIAM Press, 1979.

[24] Dongarra J.J. (ed.):"Experimental Parallel Computing Architectures." Elsevier Science Publishers, New York, USA, 1987.

[25] Edelman A.:"Large Dense Numerical Linear Algebra in 1993 - The Parallel Computing Influence." *Internal Report*, Department of Mathematics, University of California, Berkeley, California, 1992.

[26] Fisher A.L., Kung H.T., Monier L.M., Walker H. & Dohi Y.:"Design of the PSC: A Programmable Systolic Chip". *Proceeding of the Third Caltech Conference on Very Large Scale Integration*, Pasadena, California, USA, pp 287-302, 21-23 March 1983.

[27] Foulser D. & Scheiber R.:"The Saxpy Matrix-1: A General-Purpose Systolic Computer". *Computer*, Vol 20, No 7, pp35-43, July 1987.

[28] Fox G. Hey A.J.G. & Otto S.:"Matrix Algorithms on the Hypercube I: Matrix Multiplication". *Parallel Computing*, Vol 17, No 4, 1987.

[29] Fox G., Johnson M., Lyzenga G., Otto S., Salmon J. & Walker D.: "Solving Problems on Concurrent Processors, Volume 1." Prentice Hall, 1988.

[30] Gallivan K., Jalby W., Meier U. & Sameh A. H.: "Impact of Hierachical Memory Systems on Linear Algebra Algorithm Design." *International Journal of Supercomputer Applications*, Vol 2, No 1, pp12-48, 1988.

[31] Gaston F.M.F. & Irwin G.W.:"Systolic Kalman filtering: an overview." *IEE Proceedings*, Vol 137, No 4D, pp 235-244, July 1990.

[32] Goldberg D., "Computer Arithmetic" in "Computer Architecture - A Quantitative Approach.", Appendix A. Morgan-Kaufmann Publishers, Inc., San Mateo, Cal., 1990.

[33] Good I.J.:"The Relationship Between Two Fast Fourier Transforms." *IEEE Transactions on Computers*, Vol C-20, No 3, pp 310-317, March 1971.

[34] Handler W., Haupt D., Jeltsch R., Juling W. & Lange O.:"Compar"*Lecture Notes in Computer Science*, No. 237., Springer-Verlag.

[35] Hennessy J.L. & Patterson D.A., "Computer Architecture - A Quantitative Approach." Morgan-Kaufmann Publishers, Inc., San Mateo, Cal., 1990.

[36] Hipes P. & Kuppermann A.:"Gauss-Jordan Matrix Inversion With Pivoting on the Hypercube." Unpublished Caltech report C^{3}P-347, 1986.

[37] Hockney R.W. & Jesshope C.R.: "Parallel Computers: Architecture, Programming and Algorithms". Hilger, UK, 1981.

[38] Hord, R. Michael:"Parallel Supercomputing in SIMD Architectures". CRC Press, Florida, USA, 1990.

[39] Houstis E.N., Papatheodorou T.S. & Polychronopoulos C.D.:"Supercomputing"*Lecture Notes in Computer Science*, No. 297., Springer-Verlag, 1988.

[40] IDT:"Special Memories and Modules." Published 1992.

[41] Johnson K.T., Hurson A.R. & Shirazi B.:"General Purpose Systolic Arrays.", *IEEE Computer*, Vol 26, No 11, November 1993.

[42] Johnsson S.L. & Ho C.T.:"Optimum Broadcasting and Personalized Communication in Hypercubes." *IEEE Transactions on Computers*, Vol 38, No 9, pp 1249-1268, 1989.

[43] Kahaner D.K.:"Japan: a competitive assessment." *IEEE Spectrum*, pp42-47, Sept. 1992.

[44] Kahaner D.K.:"Supercomputing - the View from Japan." *IEEE Micro*, pp67-70,Feb 1993.

[45] Kalman R.E,:"A New Approach to Linear Filtering and Prediction Problems." *Transactions ASME, Journal of Basic Engineering*, Vol 82D, pp34-45, 1960.

[46] Katona E.:"A lattice model for cellular (systolic) algorithms.", *Parallel Computing*, Vol 3, pp 251-258, 1986.

[47] Kitagawa K. M.:"An MBus Tutorial." Revision 1.0a, Sun Microsystems, SPARC Technical Marketing Division, January 25, 1991.

[48] Kreyszig E.:"Advanced Engineering Mathematics, 6th Edition." John Wiley & Sons, New York, 1988.

[49] Kung S.Y.: "VLSI Array Processors" *Prentice Hall Information and System Sciences Series*, Englewood Cliffs, New Jersey, 1988.

[50] Lenoski D., Laudon J., Gharachorloo K., Weber W.-D., Gupta A., Hennessy J., Horowitz M. & Lam M.: "The Standford Dash Processor." *IEEE Computer* Vol 25, No 3, March 1992.

[51] Mace M.E.:"Memory Storage Patterns in Parallel Processing." Klewer Academic Publishers, 1987.

[52] Mallat S.G.: "Theory for Multiresolution Signal Decomposition using the Wavelet Representation." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol 11, pp 674-693, 1989.

[53] Mallat S.G. & Zhong S.:"Compact Image Coding from Edges with Wavelets". *Proceedings of the 1991 International Conference on Acoustics, Speech and Signal Processing - ICASSP91.* Vol 4, pp 2745-2748, 1991.

[54] Marwood W., "A Generalised Systolic Ring Serial Floating Point Multiplier." *Electronic Letters*, Vol. 26 No. 11, pp 753-754, 24th May 1990.

[55] Marwood W., "The Implementation of the Discrete Fourier Transform on a Systolic Configurable Array Processor." *Internal Report GAAS-89-1*, Department of Electrical and Electronic Engineering, The University of Adelaide, 1989.

[56] Marwood W., "A Number Theory Mapping Generator for Addressing Matrix Structures." *Patent Cooperation Treaty Patent*, June 1990.

[57] Marwood, W.:Discussions and Correspondence at the University of Adelaide, November 1993.

[58] Marwood W.: "An Integrated Multiprocessor for Matrix Algorithms" *Ph D. Thesis*, Department of Electrical and Electronic Engineering, The University of Adelaide, June, 1994.

[59] Marwood W. & Beaumont-Smith A., "The Implementation of a Generalised Systolic Serial Floating Point Multiplier." *APCCAS'92, IEEE, IREE and IEAust Asia-Pacific Conference on Circuits and Systems*, 8-11 Dec. 1992.

[60] Marwood W. & Clarke A.P.:"A Generic Time-Domain Beamformer Architecture." *The Australian Computer Journal*, Vol 20, No 3, August 1988.

[61] Marwood W. & Clarke A.P.:"On Computing Fourier Transforms Using a Matrix Product Machine." *Proc. 7th Australian Microelectronics Conference*, Sydney, May 16-18, 1988.

[62] Marwood W. & Clarke A.P.: "Overhead Penalties in Dynamically Reconfigurable Arrays of Processing Elements." *Journal of ELectrical and Electronics Engineering, Australia*, June 1987.

[63] Marwood W. & Clarke A.P., "A Matrix Product Machine and the Fourier Transform." *IEE proceedings - G, Circuits, Devices and Systems*, Vol 137, No 4, pp 295-301, August 1992.

[64] Marwood W. & Lim C.C.: "A GaAs Systolic Processor for Implementing a Kalman Filter." *Proc. 9th Australian Microelectronics Conference*, pp 109-114, July 2-4, 1990.

[65] Marwood, W., Shaw T., Liebelt M. and Eshraghian K.:"A Data Controller for a Systolic Outer Product Engine." *Proceeding of the 12th IREE Australian Microelectronics Conference*, pp 221-226, October 1993.

[66] Mead C. & Conway L.:"Introduction to VLSI Systems." Addison-Wesley, Reading Mass., 1980.

[67] Moore W., McCabe A. & Urquhart R.:"Systolic Arrays." Adam Hilger, 1987.

[68] Motorola: "Memory Device Data DL113, Rev7." Published 1991, previous edition 1990.

[69] Nwachukwu E.O.:"Address Generation in an Array Processor." *IEEE Transactions on Computers*, Vol C-34, No 2, pp 170-173, 1985.

[70] Ortega J.M.:"Introduction to Parallel and Vector Solution of Linear Systems." Plenum Press, New York, 1989.

[71] Pan V. & Reif J.:"Efficient Parallel Solution of Linear Systems." in *Proceedings of the Seventennth Annual ACM Symposium on Theory of Computing*, Association for Computing Machinery, New York, pp 143-152, 1985.

[72] Petkov N.:"Systolic Arrays for Matrix I/O Format Conversion." *Electronic Letters*, Vol 24, No 13, 23 June 1988.

[73] Press W.H., Teukolsky S.A., Vetterling W.T. & Flannery B.P.:"Numerical Recipes in C." Second Edition. Cambridge University Press, 1992.

[74] Quinn M.J.:"Designing Efficient Algorithms for Parallel Computers." McGraw-Hill, 1987.

[75] Reinhold O. & Marwood W.:"Silicon Hybrids - A Technique for 'Zero Defect' Wafer-Scale Processors.", *Journal of Electrical and Electronics Engineering, Australia*, IEAust. & IREE Aust., Vol 11, No. 3, September 1991.

[76] Robert Y.:"The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm." Manchester University Press/Halstead Press, 1990.

[77] Robbins K.A. & Robbins S.:"The CRAY X-MP/Model 24." *Lecture Notes in Computer Science*, No. 374, Springer-Verlag.

[78] Saad Y. & Schultz M.H.:"Topological Properties of Hypercubes." *IEEE Transactions on Computers*, Vol 37, No 7, pp 867-872, 1988.

[79] Sarkies K.: Department of Electrical and Electronic Engineering, The University of Adelaide, with Dr K. Sarkies, specialist in high speed communications systems. Personal Communication, 1994.

[80] Scheck P.B.:"Supercomputer Architecture." Klewer Academic Publishers, 1987.

[81] Schonauer, Willi: "Scientific Computing on Vector Computers." Elsevier Science Publishing Co., New York, USA, 1987.

[82] Schwider J., Streibl N. & Zürl K.:"Optoelectronic Interconnections", in *Parallel Computer Architectures*, Lecture Notes in Computer Science 732. Published by Springer-Verlag, 1993.

[83] Shaw T.J. & Marwood W., "Gauss-Jordan Elimination on a Systolic Outer Product Engine." *Internal Report, HPCA-DC-93/2*, Department of Electrical and Electronic Engineering, The University of Adelaide, 1993.

[84] Shaw T.J. & Marwood W., "QR decomposition on a Systolic Outer Product Engine." *Internal Report, HPCA-DC-93/3*, The Department of Electrical and Electronic Engineering, The University of Adelaide, 1993.

[85] Shaw T.J. & Marwood W., "A High Bandwidth, Configurable Memory Interface to a Systolic Array." *Internal Report, HPCA-DC-93/4*, The Department of Electrical and Electronic Engineering, The University of Adelaide, 1993.

[86] Shaw T.J. & Marwood W., "Towards a TeraFLOP: A Parallel Architecture to Support 1000 GFlops - Sustained." *Internal Report, HPCA-DC-93/5*, The Department of Electrical and Electronic Engineering, The University of Adelaide, 1993.

[87] Snyder L.:"Introduction to the Configurable, Highly Parallel Computer." *IEEE Computer*, pp 47-56, January 1982.

[88] Sorenson H.W. ed.:"Kalman Filtering: Theory and Application." IEEE Press, 1985.

[89] Stone H.S.:"High Performance Computer Architecture, Second edition." Addison Wesley, 1990.

[90] "SPARC MBus Interface Specification." Revision 1.1, Sun Microsystems, March 29, 1990.

[91] Takagi N., Yasuura H. & Yajima S., "High-Speed VLSI Multiply Algorithm with a Redundant Binary Addition Tree." *IEEE Transactions in Computers*, Vol C-34, No 9, September 1985.

[92] Waser S. & Flynn M.J.:"Introduction to Arithmetic for Digital System Designers." Holt, Rinehart and Winston CBS College, 1982.

[93] Zorpette G.:"The Power of Parallelism." *IEEE Spectrum*, pp 28-33, September 1992.

[94] Zorpette G.:"Large Computers." *IEEE Spectrum*, pp 34-37, January 1993.

[95] Zyner G.B.: "Design of arithmetic systems in VLSI.", PhD. thesis., The Department of Electrical and Electronic Engineering, The University of Adelaide, 1988.