Winter 1993

A massively parallel SIMD processor for neural network and machine vision applications

Michael A. Glover
University of New Hampshire, Durham

A MASSIVELY PARALLEL SIMD PROCESSOR FOR NEURAL NETWORK AND MACHINE VISION APPLICATIONS

BY

Michael A. Glover

BSEE University of Texas at Austin, 1980

MSE University of Texas at Austin, 1984

DISSERTATION

Submitted to the University of New Hampshire

in Partial Fulfillment of

the Requirements for the Degree of

Doctor of Philosophy

in

Engineering

December, 1993

This dissertation has been examined and approved.

[illegible]

Dissertation Director, W. Thomas Miller III
Professor of Electrical Engineering

[illegible]

Michael J. Carter, Associate Professor of Electrical Engineering

[illegible]

Filson H. Glanz, Professor of Electrical Engineering

[illegible]

William Wren Stine, Associate Professor of Psychology

[illegible]

Lee L. Zia, Associate Professor of Mathematics

11/12/93

Date

DEDICATION

To Hannah, Caitlin, and Carol

ACKNOWLEDGEMENTS

I would like to thank Dr. T. Miller, Dr. M. Carter, Dr. F. Glanz, Dr. L. Zia, and Dr. W. Stine for their help and support. I would like to thank D. C. Current for his sponsorship of this project. I would like to thank Paul Panish, Wayne Sanborn, and Dave Buck for the work they did on the prototype. I would finally like to thank my family, Hannah, Caitlin, and Carol, for their patience and support.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

LIST OF FIGURES xii

LIST OF TABLES xiii

ABSTRACT xiv

Chapter 1

INTRODUCTION 1

Chapter 2

REVIEW OF PAST WORK 5

2.1 Computational Tasks in Neural Networks 5
2.2 Signalling and Computation 8
2.3 Sources of Parallelism in Neural Networks 10
2.4 Measures of Neural Network Performance 12
2.4.1 External Measures 12
2.4.2 Internal Measures 13
2.5 Example Neural Networks 14
2.5.1 Nearest Neighbor 14
2.5.2 Content Addressable Memory 15
2.5.3 Local Basis Functions 15
2.5.4 Kohonen Network 16
2.5.5 ART 16
2.5.6 CMAC 17
2.5.7 Multilayered Perceptrons 17

2.6 Neural Network Hardware 18
2.6.1 Traditional Computers 18
2.6.2 DSP Microprocessors 19
2.6.3 MIMD Parallel Computers 19
2.6.4 Vector Supercomputers 20
2.6.5 Massively Parallel SIMD Supercomputers 20
2.6.6 Siemens SYNAPSE-1 Systolic Array Using MA16 Chip 21
2.6.7 CMAC Board 21
2.6.8 CNAPS 1064 Chip 22
2.6.9 Intel 80170 ETANN Chip 22
2.6.10 Intel Ni1000 Chip 23
2.6.11 Hitachi 8 Megabit Chip 23
2.6.12 Analog Radial Basis Function Chip 24
2.6.13 ATT Bell Labs ANNA Chip 24
2.6.14 ATT Bell Labs NET32K Chip 25
2.6.15 Neocognitron CCD Chip 25
2.6.16 Optical Network 26
2.7 Comparison of Thesis to Previous Techniques 26

Chapter 3

HARDWARE 27

3.1 Controller 29
3.1.1 Host Interface Port 29
3.1.2 Controller CPU 29
3.1.3 Internal Data Memory 30
3.1.4 Internal Program Memory 30
3.1.5 External Program Memory 30
3.1.6 Timer 31
3.1.7 Controller Bus 31
3.2 PE Array 33
3.2.1 PE Page 33
3.2.1.1. DRAM Memory Array 34
3.2.1.2. A, B, and M Registers 35

3.2.1.3. A and B Tap Pointers....35
3.2.2 PE Instruction Register....37
3.2.3 PE Array Instruction Set....37
3.3 Switch....42
3.3.1 Switch Connection to Pages....43
3.3.2 Switch Slice....44
3.3.3 Switch Setup....45
3.3.4 Switch Operation....46
3.3.5 Shifting Data in a Page....47
3.4 Input-Output Port....47

Chapter 4

LOGICAL MACHINE....49

4.1 Logical Machine....50
4.1.1 Logical PE....51
4.1.2 Logical Switch....54
4.2 Data Organization....54
4.2.1 An Integer is Formed from Bits....54
4.2.2 A Vector is Formed from Integers....57
4.2.3 Vectors may be Longer than the Machine is Wide....58
4.2.4 Vectors and Virtual PEs....61
4.3 PE Programming....61
4.3.1 PE Instruction Set....62
4.3.2 The Instruction Set is Functionally Complete....63
4.3.3 PE Programming Techniques....63
4.3.4 A Program to Add Two Bits....67
4.3.5 Bit Serial Arithmetic....68
4.3.6 Align and the Switch....71
4.4 Vector Instruction Set....73
4.4.1 Read a Word from a Vector....75
4.4.2 Write a Word to a Vector....75
4.4.3 Write a Constant to a Vector....76
4.4.4 Copy a Vector....76

4.4.5 Arithmetic Shift of a Vector 77
4.4.6 Bitwise NOT of a Vector 78
4.4.7 Bitwise OR of a Vector and a Vector 78
4.4.8 Bitwise AND of a Vector and a Vector 79
4.4.9 Bitwise XOR of a Vector and a Vector 80
4.4.10 Bitwise OR of a Vector with a Scalar 80
4.4.11 Bitwise AND of a Vector with a Scalar 81
4.4.12 Bitwise XOR of a Vector with a Scalar 82
4.4.13 Logical NOT of a Vector 82
4.4.14 Negate a Vector 83
4.4.15 Absolute Value of a Vector 84
4.4.16 Add a Vector to a Vector 84
4.4.17 Subtract a Vector from a Vector 85
4.4.18 Add a Scalar to a Vector 86
4.4.19 Subtract a Scalar from a Vector 87
4.4.20 Subtract a Vector from a Scalar 87
4.4.21 Multiply a Vector by a Vector 88
4.4.22 Multiply a Vector by a Scalar 90
4.4.23 Subroutine to Divide 91
4.4.24 Divide a Vector by a Vector 92
4.4.25 Divide a Vector by a Scalar 92
4.4.26 Divide a Scalar by a Vector 93
4.4.27 Modulo a Vector by a Vector 94
4.4.28 Modulo a Vector by a Scalar 94
4.4.29 Modulo a Scalar by a Vector 95
4.4.30 Compare if a Vector is EQUAL to a Vector 95
4.4.31 Compare if a Vector is NOT EQUAL to a Vector 96
4.4.32 Compare if a Vector is GREATER OR EQUAL to a Vector 97
4.4.33 Compare if a Vector is GREATER than a Vector 98
4.4.34 Compare if a Vector is EQUAL to a Scalar 98
4.4.35 Compare if a Vector is NOT EQUAL to a Scalar 99
4.4.36 Compare if a Vector is GREATER than a Scalar 100
4.4.37 Compare if a Vector is GREATER OR EQUAL to a Scalar 101
4.4.38 Compare if a Vector is LESS than a Scalar 102

4.4.39 Compare if a Vector is LESS OR EQUAL to a Scalar 103
4.4.40 Select Elements from Two Vectors 104
4.4.41 Align a Vector 105
4.4.42 Subroutine to Test All Processors 106
4.4.43 Global OR 107
4.4.44 Global AND 108
4.4.45 Global MAX 109
4.4.46 Global MIN 110
4.4.47 Global SUM 111
4.4.48 First Nonzero Element in a Vector 112
4.4.49 Create Vector with Element Indices 113

Chapter 5

C++ PROGRAMMING ENVIRONMENT 115

5.1 C++ Class Background 115
5.2 MM_VECTOR Class 116
5.3 PE Memory Heap 117
5.4 MM_VECTOR Implementation 118
5.5 Heap Garbage Collection 121
5.6 Overloaded Operators and Functions 121
5.6.1 constructor 122
5.6.2 function helper constructor 122
5.6.3 copy constructor 122
5.6.4 destructor 122
5.6.5 operator[] 123
5.6.6 set the value of an element 123
5.6.7 operator= 123
5.6.8 operator+ 123
5.6.9 operator- 124
5.6.10 operator- negate 124
5.6.11 operator| 124
5.6.12 operator& 125
5.6.13 operator^ 125

5.6.14 operator<< 125
5.6.15 operator>> 126
5.6.16 operator* 126
5.6.17 operator/ 126
5.6.18 operator% 126
5.6.19 operator== 127
5.6.20 operator!= 127
5.6.21 operator<= 127
5.6.22 operator< 128
5.6.23 operator>= 128
5.6.24 operator> 128
5.6.25 align function 128
5.6.26 select function 129
5.6.27 truncate function 129
5.6.28 operator~ 129
5.6.29 operator! 130
5.6.30 abs function 130
5.6.31 index function 130
5.6.32 sum function 130
5.6.33 or function 131
5.6.34 and function 131
5.6.35 minimum function 131
5.6.36 maximum function 131
5.6.37 first function 131
5.7 C++ Programming Considerations 131

Chapter 6

RESULTS 138

6.1 Vector Instruction Set Performance 138
6.2 Performance and the Host 144
6.3 Comparison to Other SIMD Machines 144
6.4 Miller-Scalera Neural Network Benchmarks 149
6.4.1 Nearest Neighbor Search 150

6.4.2 Radial Basis Function Neural Network 151
6.4.3 Kohonen Neural Network 151
6.4.4 Multilayer Perceptron 152
6.4.5 Discussion of Miller-Scalera Benchmarks 153
6.5 Discussion of Performance Limitations 153
6.5.1 SIMD Instruction Set Inefficiencies 154
6.5.2 Switch Inefficiencies 154
6.5.3 Global Operation Inefficiencies 155
6.5.4 Controller Inefficiencies 155
6.5.5 Host Inefficiencies 156
6.5.6 System Memory 156
6.5.7 Data Input-Output Bottleneck 157

Chapter 7

CONCLUSION 158

7.1 The MM32k SIMD Processor Array 158
7.2 Future Work 159
7.2.1 MM32k Improvements 160
7.2.2 Neural Network Implementations 161

REFERENCES 162

LIST OF FIGURES

Figure 3.1 MM32k Block Diagram 28

Figure 3.2 Controller 32

Figure 3.3 PE Array 34

Figure 3.4 PE Page 36

Figure 3.5 Crossbar Switch 43

Figure 3.6 Crossbar Switch Slice 45

Figure 4.1 Logical Machine 50

Figure 4.2 Processing Element 53

Figure 4.3 Data Format in PE Memory 56

Figure 4.4 Vector Format Across PEs 58

Figure 4.5 Vector Longer than 32768 60

Figure 4.6 PE Program to Compute ~(x XOR y) 65

Figure 4.7 PE Program to Implement a One Bit Adder 67

Figure 4.8 PE Program to Copy a Word 69

Figure 4.9 PE Program to Add Two Three Bit Numbers 71

Figure 4.10 Align Operation 73

Figure 5.1 MM_VECTOR Heap 120

LIST OF TABLES

Table 6.1 MM32k Vector Performance 139

Table 6.2 Performance Across Hosts 144

Table 6.3 Other SIMD Computers 148

Table 6.4 Nearest Neighbor Performance 150

Table 6.5 Radial Basis Function Performance 151

Table 6.6 Kohonen Performance 152

Table 6.7 Multilayer Perceptron Performance 153

ABSTRACT

A MASSIVELY PARALLEL SIMD PROCESSOR FOR NEURAL NETWORK AND MACHINE VISION APPLICATIONS

by

Michael A. Glover

University of New Hampshire, December, 1993

This thesis describes the MM32k, a massively parallel SIMD computer which is easy to program, high in performance, low in cost and effective for implementing highly parallel neural network architectures. The MM32k has 32768 bit serial processing elements, each of which has 512 bits of memory, and all of which are interconnected by a switching network. The entire system resides on a single PC-AT compatible card. It is programmed from the host computer using a C++ language class library which supports variable precision vector arithmetic. The MM32k also supports direct video input and output for machine vision applications.

Chapter 1
INTRODUCTION

Biological neural networks achieve complex behavior from large numbers of simple neurons. Artificial neural networks take their inspiration from biological networks in that they also can achieve complex behavior from large numbers of simple operations. However, for some artificial neural network architectures, the number of simple operations is so large that the network cannot be implemented in an acceptable amount of time or for an acceptable cost. This thesis describes a massively parallel computer called the MM32k which can implement highly parallel artificial neural networks much faster and more cost effectively than is possible on existing computers.

Local basis function networks are a class of neural networks which are easy to train and have good generalization properties. Included in this class are Kohonen networks, ART networks, nearest neighbor algorithms, content addressable memory, and radial basis function networks. Networks of this class have a large number of neurons, inside each which is stored an exemplar from the training set. To perform recall, all neurons are searched against the input vector and the one with the best match or a weighted sum of those matching is returned as the network response. As the capacity and accuracy of the network goes up, so does the number of neurons and therefore the amount of time required to perform the search, making the network impractical for many time sensitive applications.

A design goal of the MM32k was to speed up the implementation of local basis function type networks by performing the search and sum operations in parallel. The MM32k processor is a single instruction multiple data (SIMD) parallel processor with 32768 processing elements. Each processing element has 512 bits of memory and a one bit wide arithmetic and logic unit (ALU). All are linked together by an interconnection network. The basic advantage of the MM32k is that one processor can be assigned to each neuron in the network and all neurons can operate in parallel. Together, the processing elements can perform billions of operations per second on suitable neural network algorithms.

In the past, special purpose neural network hardware has been designed to implement neural networks and has achieved very high performance on the particular network for which it was designed. Frequently though, the flexibility of the hardware was compromised in favor of performance and the machine was unable to perform different but related algorithms. A second design goal of the MM32k was to obtain as much performance as possible while retaining as much flexibility as possible.

If the flexibility of the MM32k is to be fully exploited, then there must be effective tools for programming it. A third design goal was to implement an effective programming language. A good programming language should expose the underlying architecture of the machine to the programmer without requiring undue worry about the hardware details. A good language should structure the programmer's thinking by facilitating the natural expression of the problem in a way that can be efficiently executed by the machine. The MM32k solves this problem using a C++ class library which abstracts the MM32k as a machine for doing computation on long vectors of integers.

The MM32k is hosted by a standard PC-AT computer and is interfaced via the standard bus. The C++ class library is built on top of commercial C++ programming tools for the PC, rather than being a MM32k specific cross compiler. The C++ program runs on the host and controls the MM32k via the ISA bus. This means that ordinary C++ language programming can be mixed with MM32k programming and that all the familiar host computer programming tools including compilers, linkers, editors, optimizers, and debuggers can be taken advantage of to program the MM32k.

The general organization of this thesis is as follows:

Chapter 2 discusses computational issues in implementing neural networks including generic computational tasks, signalling, sources of parallelism, and performance measures. It also describes neural networks which are appropriate for implementation on the MM32k. It describes existing hardware on which neural networks have been implemented and shows how the MM32k is different.

Chapter 3 describes the MM32k hardware. It describes the component parts of the MM32k computer, which include the controller, the processing element array, and the switch, and describes the functions they perform. It covers the basic SIMD processing element instruction set which is visible to the MM32k controller.

Chapter 4 describes the MM32k firmware. It describes how the MM32k hardware is viewed to form a logical machine with 32768 processing elements and how that machine is programmed to perform vector arithmetic. It describes the vector instruction set which is visible to the host computer.

Chapter 5 describes the MM32k C++ class library. It describes how the MM32k can be abstracted by a class called MM_VECTOR which represents a vector of integers and how it can be programmed by manipulating variables of that class from a C++ program running on the host. It describes the overloaded C++ arithmetic operators available to the MM_VECTOR class.

Chapter 6 discusses the performance of the MM32k. The performance of the MM32k executing individual C++ class library operators is measured. The MM32k is compared in a general way to other SIMD parallel computers and its performance on some neural network algorithms is compared to traditional serial computers. Performance limitations and improvements are also discussed.

Chapter 7 is a summary of the thesis. It reviews the MM32k and affirms that the machine is effective at implementing neural networks. It discusses other applications at which the MM32k could be effective and identifies future work.

Chapter 2

REVIEW OF PAST WORK

The purpose of this chapter is to review existing hardware implementations of neural networks. To do this, it first reviews the computational tasks common to most neural networks. It then reviews signalling and computation techniques used in implementing neural networks. It then lists sources of parallelism which an implementation might exploit. It lists measures of neural network performance to provide a framework for the evaluation of an implementation. It then lists example neural networks at which the MM32k would be effective at implementing. It then describes existing neural network hardware. Finally, it shows how the processor described in this thesis differs from existing implementations. When this chapter refers to a neural network, it is referring to an artificial neural network, unless otherwise noted.

2.1 Computational Tasks in Neural Networks

This section lists computational tasks which are common to neural networks. It does not distinguish between tasks performed during recall and during learning and is also not intended to a cover all neural networks, because new networks and variations on old networks are being created all the time. Rather, it is intended to give a broad overview of the types of computations which a general purpose neural network engine should be able to perform. The computational requirements of some are put into a uniform framework by Kung [1989].

Neural networks must have some form of memory in which to store weights. Each neuron has its own set of weights and collectively they form the internal representation of the function which the network has learned. The more weights, the more the network can potentially learn. Weights have been stored in digital static random access memory (SRAM) cells [Hammerstrom, 1990], digital dynamic random access memory (DRAM) cells [Watanabe, 1993], analog charge coupled device (CCD) cells [Chuang, 1990], electrically erasable programmable read only memory (EEPROM) cells [Intel, 1991], photographic film [Lu, 1991], and chemicals in biological synapses [Llinas, 1988].

Neural networks must have some way in which the weights can be set. If the weights can be adjusted while the network is running, then the network can potentially be trained on line, instead of off line. Some networks require a probabilistic weight update [Melton, 1992].

Data must be moved around between the network input, the neurons, and the network output to implement the connections in a neural network. Either individual physical paths can be constructed for each connection, or fewer paths can be shared over time. Depending upon the topology, the resources devoted to connections can dominate the system. Connections have been implemented electrically with wires, optically with holograms and lenses [Krishnamoorthy, 1992], and biologically with axons and dendrites [Llinas, 1988].

The basic arithmetic operation of the neurons in many networks is to compute a dot product between an input vector and a weight vector. A dot product requires many multiply and add operations. Addition operations are usually relatively fast and inexpensive to implement, but multiplication can be slow, more expensive, or more difficult to implement, depending upon the technology used. In some networks, this dot product will be normalized, which requires a division. The dot product has been computed using digital arithmetic, using pulse modulation [Hamilton, 1992], by using analog transistor circuits [Intel 1991], and by using a photographic film and light [Lu, 1991].

The basic operation of the neurons in some networks is the computation of the distance between an input vector and a weight vector. This distance can be a euclidean distance, a city block distance, or some approximation. The distance differences may be weighted. Depending upon how it is implemented, the metric can require some combination of addition, subtraction, absolute value, comparison, multiplication, and square root. Addition and subtraction are relatively easy to implement, but absolute value and comparison are more difficult and multiplication and square root are more difficult and expensive still. The distance metric has been computed using digital computer arithmetic and using analog transistor circuits [Anderson, 1992].

The dot product or distance metric of each neuron is fed into a nonlinear activation function, which determines the output of the neuron. The simplest form of an activation function is a unit step function, where the output is one if the input is greater than a threshold value and zero otherwise. A variation on this is to allow the threshold to be gradual so that the output is one or zero for an input which is much greater than or less than the threshold value, respectively, and makes a gradual transition in between. The derivative of the output would then exist everywhere. This function has been computed by digital computer arithmetic, by lookup tables [Hammerstrom, 1990], and by analog transistor circuits [Intel, 1991].

In some networks, the sum of all neuron outputs must be formed and in others, the neuron with the maximum output must be selected and the outputs of the other neurons suppressed. These operations can be difficult because they are global and require some form of communication between all neurons in a particular layer. This operation can take time which is linear, logarithmic, or fixed relative to the number of neurons in a layer, depending upon implementation. Global operations have been computed using digital and analog [Lazzaro, 1988], electrical circuitry.

In some networks, a boolean predicate will be evaluated at each neuron and the neuron will subsequently be active or inactive depending upon the outcome. The predicate may be an arbitrary function of the weights and the inputs and may use arbitrary arithmetic operators and functions.

2.2 Signalling and Computation

All networks must move and compute with the data using some signalling scheme. The best scheme for a particular network depends upon whether the information is continuous or discrete, upon how it is stored, upon how it is sent, and upon how it is used in computation. Many systems are hybrid and combine different schemes to take advantage of the strengths of both.

Neural networks have been implemented using digital floating point arithmetic. An advantage is that the implementor has a large amount of dynamic range and accuracy. Networks using it can be implemented simply using common computers and digital signal processing (DSP) microprocessors. Digital signals are relatively immune to noise. One disadvantage is that addition and particularly multiplication use a lot of silicon resources and another is that it uses a lot of memory for data which frequently has only five significant bits of precision.

Neural networks have been implemented using digital fixed point arithmetic. An advantage is that it requires less silicon to implement than digital floating point and is usually faster. The disadvantages are that the implementor must be concerned with dynamic range and that it can waste memory.

Digital floating point and digital fixed point have been implemented in a bit serial fashion [Hwang, 1984]. In bit serial implementations, the data paths of the memory, ALU, and interconnections are all one bit wide and computations are performed one bit at a time. The advantage is that only as many bits need be stored and processed as are needed by the problem. The disadvantage is that the parallelism of bits across a word is lost.

The amplitude of a signal can be encoded as the duty cycle or frequency of a digital pulse stream [Hamilton, 1992]. The advantages are that data paths can be one bit wide and can be implemented using digital devices. Multiplication can be done with an AND gate. The signals can be converted to an analog voltage using an integrator. The disadvantage is that a signal cannot be directly stored in a memory, limiting the flexibility of a system. Another disadvantage is that the accuracy of the scheme is limited and it is susceptible to noise. The same input presented twice may yield two different outputs.

The advantage of analog electrical circuits is that a given amount of silicon has a higher information processing bandwidth than with digital circuits [Intel, 1991]. Multiplication and addition can be done cheaply. However, accuracy is limited to around six bits and some nonlinear functions are difficult to achieve. Analog electrical circuits can be susceptible to noise and so the behavior of the network may not be deterministic, which complicates testing and debugging. It is sometimes difficult to implement some functions in the presence of semiconductor fabrication process variation. An analog signal can be stored in a capacitor, but will decay if not refreshed periodically, limiting the general usefulness of this technique. Analog weights are not easily updated and the network must usually be trained off line.

All of the digital and analog techniques listed above may in theory be implemented optically. Optical networks can potentially move a large amount of data very quickly in parallel. They can also potentially take advantage of free space interconnect using lenses and holograms [Krishnamoorthy, 1992]. Addition can be performed by integrating the photons which fall on a sensor. Multiplication can be performed by directing a beam of light through an attenuator such as a piece of film or a liquid crystal light modulator [Lu, 1991]. Correlation, which is equivalent to a dot product, can be performed using only lenses and an attenuator. The disadvantage is that the technology for manipulating optical signals is very poorly developed compared to that for electrical signals. Many optical neural networks use optics as a large fast correlation engine.

Biological signalling is the signalling technology of living organisms. Human beings and animals can easily perform many useful tasks which computers cannot. Biology is worthy of mention here because it is clearly successful and, to a greater or lesser degree, is the inspiration for artificial neural network computation. In a human brain, there are 10 billion neurons, each of which is connected to 10000 others for a total of 100 trillion connections. Each neuron can fire up to 1000 times a second giving a potential total of 100 quadrillion connections per second. When a neuron fires, a pulse travels down an axon which spreads into a series of terminal fibers. The tip of each terminal fiber terminates at a synapse on another neuron. The neuron might learn by increasing the transmission efficiency of the synapse, which is analogous to changing the value of a weight. A pulse arriving at a synapse can induce the neuron to fire or inhibit it from firing. This model of biological neurons is greatly simplified and is included only to indicate that on some level, biological neural networks are similar to artificial neural networks. The functioning of individual biological neurons is relatively well understood when compared with the functioning of higher level operations. The human brain is described by Sholl [1956], and human and nonhuman nervous systems in general by Llinas [1988].

2.3 Sources of Parallelism in Neural Networks

There is parallelism in common places across different types of neural networks. This is because most types of networks are made up of a large number of simple but identical neurons. One way to go faster is to build hardware which will operate in parallel on the same network. Another way is to take advantage of the vector nature of most neural computation and build more efficient hardware, for example by pipelining. In any network, there is a limit in the available parallelism, and once the limit is reached, any further speed increase must come through improvements in the hardware technology.

There is parallelism across the nodes in a layer of a network since they all compute at the same time. All nodes are independent and so their computations may proceed independently. A separate piece of hardware may be assigned to each node.

There is parallelism across the inputs of a node. Whether the node computes a dot product or a distance metric, all of the inputs may be processed in parallel to form partial sums. There is more parallelism in that the complete sum may be formed using a tree of adders, although it takes log(n) steps, where n is the number of inputs. If the sum is performed using analog circuits, the fan-in of the tree is usually small enough that the sum can be performed in a single step.

If the network is implemented digitally using word parallel arithmetic, there is parallelism across the bits of a word. This form of parallelism is traditionally exploited by most digital computers in operation today. A separate piece of hardware may be assigned to each bit. Word parallel arithmetic must actually be performed in a bit serial fashion since a carry bit must usually propagate across all bits or at least through a fast carry tree. However, since the propagation times are usually shorter than a single cycle, this serial aspect of word parallel arithmetic may be ignored.

Some types of networks require the global sum, OR, AND, or maximum of neuron outputs to be computed. This is listed separately from parallelism across neurons because the time to perform global operations is logarithmic in the number of neurons. The base of the logarithm depends upon the implementation technology and is frequently two in digital implementations, but may be as large as the number of neurons in analog technology or by using a digital wire OR, yielding single step computation.

If more than one result from the network is desired, then parallelism can be exploited by assigning hardware for each result to be computed or by pipelining computations through a single set of hardware. Pipelining shortens the critical paths in the circuit and allows the system clock to run faster, although it requires a little extra hardware and introduces latency.

2.4 Measures of Neural Network Performance

This section attempts to list different ways in which neural network performance can be measured. The first group, recalls per second, recall latency, quality of recall, recall capacity, training speed, and number of training presentations, are externally oriented and indicate the useful performance of a network implementation in the context of its application. The remaining measures, connections per second, connection updates per second, connections per second per weight, internal storage capacity, and internal precision, are internally oriented and indicate the amount of internal computation going on in a network implementation.

2.4.1 External Measures

The number of recalls per second indicates how many outputs per second the implementation can perform and therefore indicates how much useful work is performed. The number and precision of input and output vectors should be considered, but this measure does not attempt to look inside the network.

There may be latency in some implementations due to pipelining or data set parallelism. In this case, the time to process an input may be longer than the interval at which inputs are presented to the network. In real time control problems where the network is used to close a control loop, this latency can slow down the effective response rate of the controller.

The quality of recall indicates the usefulness of the network output in the context in which it is being used. If emulating a function, it may be the mean squared error over the useful range of the function. If performing a classification, it may be the error rate of the classifier. This measure is included here because it is often possible to trade off other network performance measures to improve the quality of the network result.

The recall capacity is measured in bits and indicates how much information the network can store. A network must adapt its internal weights to learn and mimic a function which may be simple or complicated. Since the network has finite internal storage, there is a limit to the complexity of the function which can be represented.

The training speed indicates how quickly the network can be trained. It is usually measured as total time to achieve a certain level of performance. Some network hardware can only perform recall, but cannot perform a training algorithm and must trained off line. Some network types have required months to train and others can be trained trivially.

Some networks require multiple presentations of the same training set of data. On each presention, the network improves its performance. If a network is designed to be trained by online observation, then the rate at which data is collected may limit the training rate.

2.4.2 Internal Measures

<u>Connections per second</u> (CPS) indicates how may connections are evaluated per second when the network is being used for recall. In some networks, each connection corresponds to an element of a dot product and involves a multiplication of the neuron input with a weight. In other networks, each connection corresponds to a dimension of a distance metric where the weights correspond to the center of the neuron receptive field. Neurons often have a nonlinear squashing function applied. This time is usually folded into the connections per second measure.

<u>Connection updates per second</u> (CUPS) indicates how many connections can be updated per second while the network is being trained. In some types of networks, all

weights are incrementally modified on each training cycle and training can become long while in other types, training time is small.

Connections per second per weight (CPSPW) indicates how may times a particular connection is evaluated per second. The number of CPSPW tends to be constant for a particular implementation technology, so this measure allows different technologies to be compared.

Internal storage capacity is the amount of memory internal to the network and is measured in bits. The memory in some networks is analog, but this can be represented as the number of significant bits above noise. This measure puts an upper bound on the recall capacity and allows different implementation technologies to be compared.

All network inputs, outputs, internal connections, and weights have an inherent precision which is usually expressed in bits. This measure is listed here because whenever the speed of a computation is listed, it is also appropriate to indicate how much information was involved. For instance, weights can be 32 bit floating point values or one bit boolean values.

2.5 Example Neural Networks

This thesis describes a SIMD parallel computer with 32768 processing elements. With this architecture come a unique set of advantages and disadvantages. This section will focus on those neural networks which can be implemented efficiently on the MM32k.

2.5.1 Nearest Neighbor

A nearest neighbor network stores the training examples in its memory. When asked to recall a value, is searches the items stored in its memory and returns the output value of the item which is closest to the input vector. The distance metric can be euclidean distance which requires multiplication, but city block and other distance metrics can be used which are faster to compute. An advantage of nearest neighbor is that it is easy to train, because the network is trained when the training set is placed into the neurons. A disadvantage is that all items in the memory must be searched in order to return a value. One way to reduce the number of items in the memory is to only store exemplars, or representative examples. A variation on nearest neighbor is K nearest neighbor. In K nearest neighbor, the average or consensus of the K nearest items is returned. In this type of network, there is parallelism across neurons, across the elements of the distance metric, across the bits of the elements of the distance metric, and across the neurons as the maximum is selected. Nearest neighbor algorithms were not originally developed as neural network algorithms, but are similar in form and function to local basis function networks and are considered as such. Nearest neighbor algorithms are described in more detail in Duda [1973].

2.5.2 Content Addressable Memory

Content addressable memory is very similar to nearest neighbor. The weights of each neuron contain the elements of each training vector, only the elements are not restricted to be numerical values, but may be a string of characters. To recall a value, an arbitrary test is made on the elements of all training examples and those meeting the test are selected. Some or all of the elements of the selected neurons may be returned as outputs. If no neurons respond the recall effort has failed. The set of elements which participate in the test and the set which are outputs is arbitrary and may overlap. In this type of network , there is parallelism across neurons, across independent parts of the arbitrary test, across the bits of elements tested, and across neurons as the results are returned. Content addressable memory is described in more detail in Foster [1976]. Content addressable memory is also known as associative memory and is described as such by Potter [1992].

2.5.3 Local Basis Functions

The neurons of a local basis function network store a center vector and an output amplitude as weights. The neuron output is the amplitude value weighted by a distance function which is unity when the input vector is near the neuron center and drops off to zero as the input vector moves away. Each neuron is said to have a local response. The output of the network is the sum of the outputs of the neurons. Intuitively, the centers of the neurons correspond to the centers of the training set. A modification to the basic algorithm is that a width weight vector can be added to vary the rate at which the distance function drops off as the input vector moves away. This allows a single neuron to represent more than one point in the training set. Another modification is that the distance function can be normalized and the network output is a weighted average of the neuron outputs. In this type of network, there is parallelism across neurons, across the elements of the distance metric, across the bits of the elements of the distance metric, and across the neurons as the global sum is computed. Local basis function networks are described by Moody [1988], Broomhead [1988], and Moody [1989].

2.5.4 Kohonen Network

The neurons of a Kohonen network store a center vector and an output value or label. To perform recall, the neuron closest to the input vector is identified and the output which it contains is returned. The input vector is normalized to unit length before being presented to the network. During training, all neurons are on a two dimensional grid. For each training input, the neuron with a weight vector closest to the input vector is selected. Then all neurons near the selected neuron on the grid are modified to be more similar to the input. In this type of network, there is parallelism across neurons, across the elements of the distance metric, across the bits of the distance metric elements, across the neurons as the maximum is selected, and across the neurons near the selected neuron. The Kohonen network is described by Kohonen [1984].

2.5.5 ART

The neurons of an adaptive resonance theory (ART) network store a weight vector.

To perform recall, each neuron computes the dot product between the input vector and the weight vector. If the match is not close enough, the selected neuron is disabled and the process is repeated. In this type of network, there is parallelism across neurons, across the elements of the dot product, across the bits of the dot product elements, and across the neurons as the maximum is selected. The ART network is described by Carpenter [1988].

2.5.6 CMAC

A cerebellar model arithmetic computer (CMAC) network implements virtual neurons which are regularly spaced on a grid. To perform recall, the input vector is split into a family of input vectors which are near by the original input vector in the input space. Each of these vectors is mapped through a hash table and selects a neuron which can be implemented in a table in ordinary computer memory. The network returns the average of the physical neuron outputs. An advantage of CMAC is that it is a local basis function type network which can be implemented efficiently on a traditional serial computer. In this type of network, there is parallelism across the family of input vector, across the hash function generation, and across the selected elements of the physical memory which form the average which is returned. The CMAC algorithm is described by Albus [1975] and Miller [1990].

2.5.7 Multilayered Perceptrons

The neurons of a multilayered perceptron network store a weight vector and a threshold value. During recall, the neurons compute the dot product of the weight vector with the input vector and apply a threshold function, whose transition is given by the threshold weight. The output of the threshold function is between zero and one and is passed to the next layer in the network. This network is presented here for completeness, but is not implemented efficiently on the MM32k because of the large number of multiplications and inter-neuron communication operations. In this type of network, there is parallelism across neurons, across the elements of the dot products, across the bits of the elements of the dot product, and across the layers of the network if the implementation is pipelined. Multilayered perceptron networks are described in McClelland [1986].

2.6 Neural Network Hardware

The purpose of this section is to describe the computing hardware on which neural networks have been implemented in the past. This hardware sits on a continuum with flexible but slow systems at one end and fast but inflexible systems at the other. The most flexible systems are the general purpose computers and microprocessors. Less flexible, but more powerful are vector processors such as DSP microprocessors and vector supercomputers. More powerful still are the SIMD and MIMD parallel computers. The most powerful are the specialized neural network architectures which have hardwired algorithms and sometimes nondigital processing techniques. These systems give the best cost effectiveness but are frequently ineffective on neural network problems which are only incrementally different. A design goal of the MM32k is to achieve high performance without giving up flexibility.

2.6.1 Traditional Computers

Traditional computers can be used to implement neural networks. They are usually built around one of the popular general purpose microprocessors. Examples include the Sparc series, the Motorola 680x0 series, the Intel 80x86 series, the MIPS R4000 series, and the Digital Equipment Corporation Alpha series. The advantages are that they are cheap, have good programming tools, and are easy to work with. These computers utilize bit level parallelism and a moderate amount of pipelining on the instruction stream. They are capable of CPS measures in the neighborhood of one to ten million. Most are good at integer arithmetic and some are better than others at floating point arithmetic. They usually have a lot of weight memory and good weight memory density because it is implemented with

DRAMs. They are presented here because they are a popular choice.

2.6.2 DSP Microprocessors

Neural networks have been implemented on board level systems available which utilize digital signal processing (DSP) type microprocessors. The DSP microprocessors are distinguished from traditional computers in several ways. First, these machines have a multiply-accumulate unit and autoincrementing address generation hardware which allows them to evaluate a dot products at one partial sum per instruction. Second, they usually have a Harvard type architecture, with separate data and program memories, so that instruction fetch may be overlapped with data fetch. Third, they are relatively difficult to program at an assembly language level and are not efficient executing a higher level language. The DSP microprocessor chips frequently contain both data memory and program memory internally and are low in cost. Some types have 16 bit data paths and perform fixed point calculations and others have 32 bit data paths and perform floating point calculations. They can achieve 10 to 40 million multiply-accumulate operations per second. The Texas Instruments TMS320x0 series, described by Texas Instruments [1992], is representative of DSP microprocessors.

2.6.3 MIMD Parallel Computers

Neural networks have been implemented on multiple instruction multiple data (MIMD) parallel computers. A MIMD parallel computer is a set of ordinary computers which can communicate through an interconnection network. The neural network computation and weight storage is spread across the nodes of the MIMD computer. The topology of the interconnection network does not have to match the topology of the neural network, although it should efficiently support it. The Ring Array Processor (RAP), described by Morgan [1990], is representative. It is a group of TMS320C30 32 bit floating point microprocessors, each with 16 megabytes of DRAM memory, 256 kilobytes of fast SRAM memory, and each capable of 32 megaflops while performing multiply-accumulate operations. The microprocessors are interconnected in a ring topology, which is expandable. Up to 40 microprocessors have been operated in a single system for a measured performance of 574 million connections per second. A system is a collection of VME bus cards, each with four DSP microprocessors, and all connected to and hosted by a Unix workstation.

2.6.4 Vector Supercomputers

Neural networks have been implemented on vector supercomputers. These systems are optimized for 32 bit or 64 bit floating point vector arithmetic. The architectural advantage is that the arithmetic and logic unit can be deeply pipelined and is sometimes replicated. They are usually implemented using exotic logic technology running at an extremely high clock rate. They usually have large amounts of main memory and require vectors to be 50 to 100 elements long before the vector instruction overhead is a small part of computation. They can achieve 100 million to 10 billion floating point operations per second of performance. They are very expensive. They are programmable in high level languages by using vector libraries. The series of computers manufactured by Cray Research is representative. The first Cray machine, the Cray 1, is described by Russell [1978], although higher performance models have superceded it.

2.6.5 Massively Parallel SIMD Supercomputers

Neural networks have been implemented on massively parallel single instruction multiple data (SIMD) supercomputers. These systems have 1024 to 65536 simple processors, each with memory and an arithmetic and logic unit. All processors are interconnected through a switching network and all processors execute the same instructions at the same time using different data. Economics or available problem set parallelism, not technology usually limits the number of processors which can be built in a single machine. Processors usually have data paths which are one bit wide and do bit serial arithmetic. These computers are typically built around a custom chip which implements 16 processors with the processor memory being implemented using standard memory chips. Like vector supercomputers, these machines are usually very large and expensive, although they are more cost effective in appropriate applications. For a SIMD parallel computer to be effective, the network must have enough parallelism to keep all of the processors busy and the required connections must be efficiently implementable by the switching network. Example SIMD parallel computers are the Distributed Array Processor (DAP), described by Active Memory Technology [1989], the Massively Parallel Processor (MPP), described by Potter [1985], the Connection Machine 1 (CM-1), described by Hillis [1985], and the Maspar series, described by Nickoll [1990].

2.6.6 Siemens SYNAPSE-1 Systolic Array Using MA16 Chip

The SYNAPSE-1 system is an all digital systolic array of processors. A systolic array is similar to a SIMD computer in that it is an interconnected array of processors which receive the same instructions at the same time. It is different in that the data is not located at each processor, but is pumped across the array from an external memory. This system contains 32 processors organized into eight MA16 chips and interconnected in a two dimensional mesh. The system can compute 5.1 billion 16 by 16 bit connections peak using a 40 megahertz clock although it cannot fetch weights from the weight memory at this rate. It can perform both recall and training and is supported by a C++ class library which abstracts the elementary operations of the chip. The core area of each chip is 98 square millimeters, contains 488 thousand transistors, and is implemented in 0.8 micrometer CMOS. This system is described by Ramacher [1991], and Ramacher [1993].

2.6.7 CMAC Board

This board implements the CMAC algorithm with a combination of a fixed point DSP microprocessor, static RAM memory, and programmable gate arrays. The number of

16 bit inputs can be from one to 512 and the number of 16 bit outputs can be from one to eight. Up to eight independent virtual CMACs can be simultaneously stored on a single card. The CMAC algorithm contains a large number of virtual connections and physically implements only the small fraction of those which are nonzero. Networks with 32 inputs and eight outputs can be implemented in less than one millisecond making the board ideal for robotic control problems. The board can perform both recall and training and is described by Miller [1990].

2.6.8 CNAPS 1064 Chip

The CNAPS chip has 64 processors and is intended to be part of a SIMD parallel processor optimized for neural networks. Each processor has a fixed point ALU which operates at a 25 megahertz rate for a total of 1.6 billion 16 bit by 8 bit multiply-accumulate operations per second. Each processor has 4096 bytes of SRAM memory for a total of 256 kilobytes per chip, which can be configured as 1, 8, or 16 bit weights. The peak memory bandwidth is 3.2 gigabytes per second. A total of 80 processors are fabricated on each chip and each local memory of each processor has redundant rows. Bad processors and bad rows are switched out to increase the yield of the manufacturing process and reduce the cost of the chip. Usually four chips are mounted together with a controller to form a system, although more chips can be added. The chip can perform both recall and training and is supported by a compiler for the C* parallel programming language. The chip is all digital, is implemented in 0.8 micrometer CMOS, is 26.2 by 27.5 millimeters, and is described by Hammerstrom [1990].

2.6.9 Intel 80170 ETANN Chip

The Intel 80170 Electrically Trainable Analog Neural Network (ETANN) chip is designed to perform dot products. It contains 64 neurons, each with 128 synapses and a sigmoid function. The weights are stored in nonvolatile analog EEPROM cells with six bits of precision. All inputs, outputs, and computations are analog. The chip has internal feedback circuitry and can be configured to implement multilayer and hopfield networks. Multilayer networks can also be formed by cascading multiple chips. The chip has only 64 physical inputs, which must be multiplexed in time to connect to the 128 synapses, and 64 outputs. It can achieve over two billion connections per second during recall and has provision to support learning at 100 thousand connection updates per second. The chip is described by Intel [1991].

2.6.10 Intel Ni1000 Chip

The Intel Ni1000 chip is designed to compute radial basis function type neural networks. It implements 1024 neurons, each of which computes a city block distance metric between a 256 dimensional five bit input vector and 256 five bit weights. Neurons within a specified radius of the input vector are selected and a six bit classification label is returned. Estimation of class probability distribution functions is also supported by using a weight on the output of each neuron. The chip contains a microcontroller and can be programmed to implement other algorithms. The chip can perform a classification every 50 microsecond in pipelined mode for a total of five billion connections per second. All weights are stored in nonvolatile digital EEPROM memory so they are not lost when the chip is powered down. This chip is all digital and is described by Scofield [1991] and Intel [1993]. It is currently under development.

2.6.11 Hitachi 8 Megabit Chip

This proposed chip is built around an 8 megabit DRAM, is all digital, and will perform dot products. The memory has 16384 data lines and cycles in 1500 nanoseconds for a gross memory bandwidth of 1.3 gigabytes per second. This data is multiplexed into 256 eight bit by eight bit multiply-accumulators which can perform 1.37 billion connections per second. The memory array can hold up to one million eight bit weights. The chip is to be implemented in 0.5 micrometer CMOS, will be 15.4*18.6 millimeter, will operate at 1.5 volts, and will dissipate 75 milliwatts when operating. The weights are trained off line. A one sixteenth slice of this chip has been fabricated and tested. The chip is described by Watanabe [1993].

2.6.12 Analog Radial Basis Function Chip

This chip implements a radial basis function type network with eight inputs 159 neurons, and four outputs. For each neuron, distance differences are computed on a per axis basis and a gaussian function is applied to each difference. The outputs of the gaussians are then summed and compared to a threshold. If the comparison difference is greater than zero, the comparison difference squared is returned as the weight associated with the neuron, otherwise zero is returned. A weighted average of neurons outputs is computed and returned as the output of the network. The distance metric is not truly radial in a multidimensional sense. In two dimensional space, the basis function of a neuron covers the union of two stripes instead of the insides of a circle. This chip is unique in that it does this with analog instead of digital circuitry. The weights are implemented with charges on capacitors and are trained off line. This chip is described by Anderson [1992].

2.6.13 ATT Bell Labs ANNA Chip

This analog-digital hybrid chip implements 4096 physical synapses and is designed to implement dot products. The synaptic weights have six bits of precision and are stored as analog charges which are periodically refreshed. The input vector has three bit elements. The multiplication is done by a multiplying digital to analog converter. To form a dot product, the results are summed in analog and converted back to digital by a three bit analog to digital converter. The chip is flexible and can be configured to have from 16 neurons with 256 inputs to 256 neurons with 16 inputs. The chip can store over 130 thousand connections and compute 5 billion connections per second. The weights are trained off line. The chip measures 4.5 by 7 millimeters, is fabricated in 0.9 micrometer CMOS, and contains 170 thousand transistors. This chip is described by Boser [1991].

2.6.14 ATT Bell Labs NET32K Chip

This analog-digital hybrid chip implements 32768 one bit connections in 256 neurons and is designed to implement dot products. Each neuron has 128 one bit digital weights which are multiplied by 128 one bit digital inputs from an input vector. The one bit products are summed and compared using an analog circuit. All neurons receive the same input vector. The chip is not restricted to dealing with binary inputs and outputs. The sum at each neuron can be scaled by 1, 1/2, 1/4, or 1/8 and added to the scaled sums of the neighboring neurons to produce a single larger composite neuron. In this way, the input vectors can have up to four bits of precision, although some weights must be set to zero and are wasted. The composite output can be detected with different thresholds by using the threshold detectors of each of the individual neuron threshold detectors to produce an output vector with up to 3 bits of precision. The chip can cycle in 100 nanoseconds giving a performance of 320 billion one bit connections per second. It has a high degree of flexibility from a simple analog design, but has no provision for training the weights on line. The chip measures 4.5 by 7 millimeters, is fabricated in 0.9 micrometer CMOS, contains 412 thousand transistors and is described by Graf [1991].

2.6.15 Neocognitron CCD Chip

This chip implements the neocognitron neural network using analog CCD processing. It applies a seven by seven spatial filter to the pixels of an image to search for features. The filter is a 49 point dot product. The chip contains a 775 stage CCD tapped delay line for holding and shifting six lines of 128 pixel values each plus seven pixels of the following line. There are 49 multiplying digital to analog converters and an analog summer which produce an output pixel each time the delay line is shifted. Each multiplying digital to analog converter has storage for 20 weights of eight bits so that 20 different features can be computed without reloading weights. The device occupies 29 square millimeter and performs over one billion operations per second when clocked at 10 megahertz while using less than one watt of power. It is described by Chuang [1990].

2.6.16 Optical Network

Neural networks can be built from optical components. The most common use is to perform a dot product by multiplying in the spatial frequency domain. This dot product is equivalent to a correlation. A lens performs a Fourier transform on an image which is then projected onto a light attenuator where a point by point multiplication is performed. The attenuator is implemented by a piece of film or a liquid crystal light modulator and holds the fourier transform of the pattern which is to be correlated. A second lens then performs another fourier transform on the signal to return it to its original domain and form the correlation of the original image with the pattern image. The correlation image is then searched for a peak to obtain the best matching pattern. This technique is used by Wunsch [1991] to implement the ART1 network.

2.7 Comparison of Thesis to Previous Techniques

The processor described in this thesis is unique in several ways. First and most significantly, it is unique in that it is far more cost effective than other SIMD designs on suitable problems. This design implements 2048 processing elements on a single chip where typical SIMD designs implement 16 processing elements on five chips. The ALU and instruction set of the processing elements are also unique. The software support is unique in that it dynamically varies the precision of computation on a bit basis at run time, where most processors bind the precision on a byte basis at compile time.

Chapter 3

HARDWARE

The purpose of this chapter is to describe the hardware of the MM32k. The chapter after it describes a logical machine which can be implemented with the hardware and the chapter after that describes how that logical machine can be programmed using a C++ class library. This chapter will describe the capabilities of the hardware in enough detail to show that it can support the model in the next chapter.

The hardware is divided into four parts, the processing element (PE) array, the switch, the input-output port, and the controller. The PE array contains the memory and ALUs of the SIMD machine. The switch allows the PEs to exchange data. The input-output port allows data to be transferred in to and out of the machine at high speed. The purpose of the controller is to receive and execute instructions from the host. A block diagram of the system is given in the next figure.

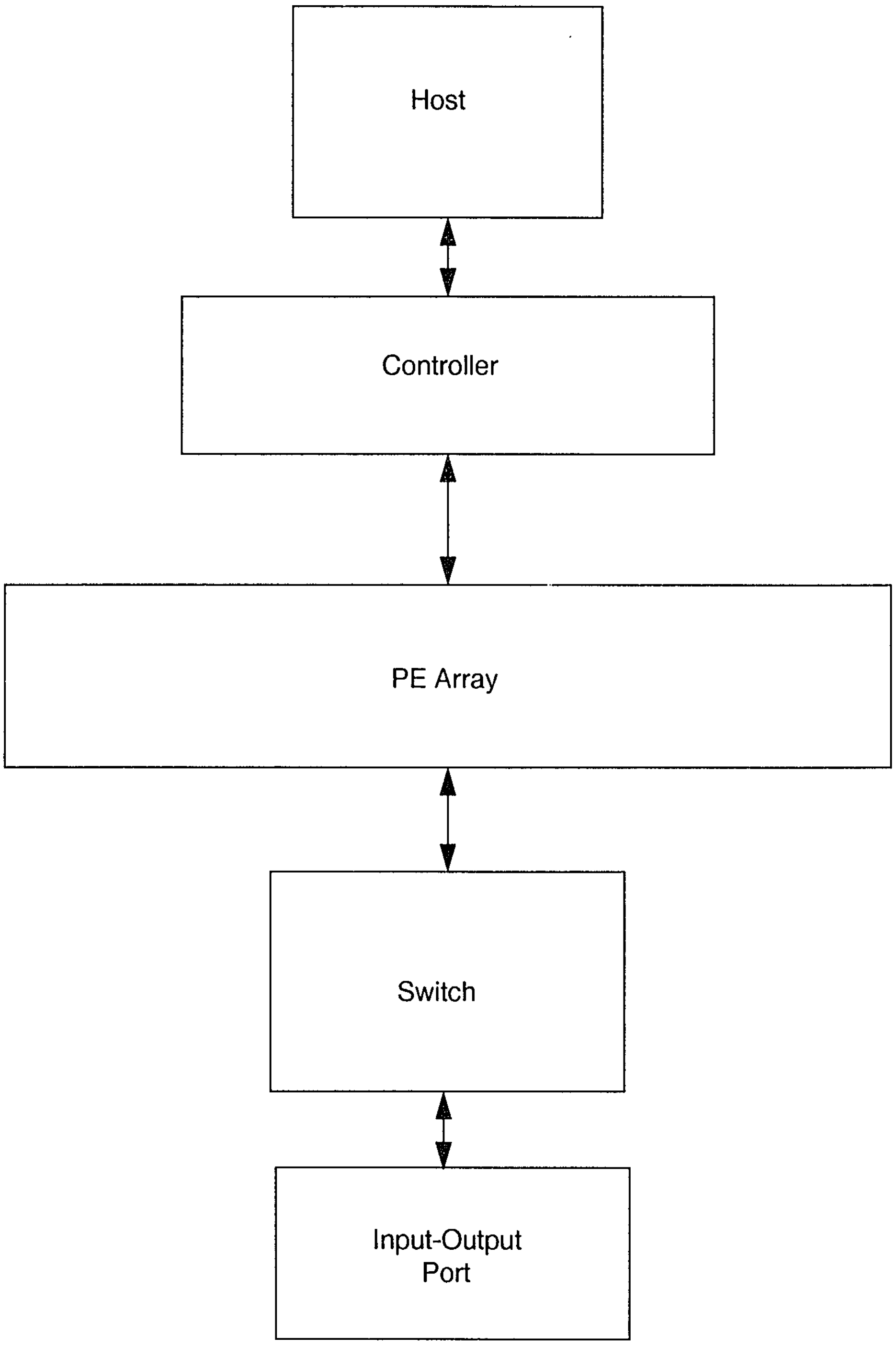


Figure 3.1 MM32k Block Diagram

3.1 Controller

The purpose of the controller is to receive and direct the execution of SIMD operations requested by from the host computer. It is shown in the next figure. The controller is an Analog Devices ADSP2111, which is a single chip 16 bit DSP microprocessor with internal data memory, internal program memory, an internal timer, an interrupt system, and an interface to the host computer. Internally, the controller has a Harvard type architecture, but externally has a single bus, to which external program memory, the PE array, and the switch are connected.

3.1.1 Host Interface Port

The host interface port (HIP) allows the controller to communicate with the host. The HIP, which is integrated onto the DSP chip, consists of six 16 bit bidirectional data registers and two status and control registers. The controller can configure the HIP so that an interrupt is generated when the host computer writes data to or reads data from the registers. The controller is reset and booted via the HIP. A complete copy of the controller microcode is downloaded from the host and placed into program memory during this process.

3.1.2 Controller CPU

The controller central processing unit (CPU) has an ALU, a shifter, a multiply accumulate (MAC) unit, two address generators, program control mechanisms, and duplicate register sets. It usually executes an instruction in a single 80 nanosecond cycle, providing a 12.5 megahertz instruction rate. The ALU allows it to perform 16 bit arithmetic and logic operations. The shifter allows it to logically and arithmetically shift and mask 16 bit data. The MAC block allows it to perform single cycle multiply and accumulate operations. The two address generators allow it to specify two autoincremented addresses to reference data and program memory and can also implement circular buffers with zero overhead. The program control mechanism supports subroutine calls, interrupts, and zero overhead looping.

For example, a loop with a body of one instruction will execute at a rate of one cycle per iteration. This makes it easy to code the loops associated with bit serial arithmetic. The data registers associated with the ALU, the shifter, and the MAC are duplicated, so context switching may be performed quickly during an interrupt.

3.1.3 Internal Data Memory

The internal data memory is 1024 words of 16 bits. It contains a 100 word queue of instructions from the host, implemented as a circular buffer. Instructions are placed in the queue in response to interrupts to the DSP microprocessor from the HIP, caused by the host writing data into the HIP registers. To reduce the overhead, instructions are always enqueued in groups of 6 words. Data memory also contains working variables associated with the execution of instructions. Since it is internal to the DSP microprocessor, data memory can be accessed without causing a cycle on the controller bus.

3.1.4 Internal Program Memory

The internal program memory is 2048 words of 24 bits and holds native instructions for the DSP microprocessor. It is loaded by the host system when the MM32k is booted via the HIP. Data may also be kept in program memory, and fetched in parallel with data from the data memory without adding cycles to a DSP instruction. Since it is internal to the DSP microprocessor, program memory may be accessed without causing a cycle on the controller bus. This is important because instructions to the PE array are issued over the controller bus.

3.1.5 External Program Memory

The external program memory is 6144 words of 24 bits and was added because the 2048 words of internal program memory are not enough to support the complete MM32k instruction set. It is attached to the DSP via the controller bus and therefore DSP instructions executed out of it require a controller bus cycle. This is important because it can prevent the DSP from issuing PE array instructions over the controller bus, slowing down the MM32k.

3.1.6 Timer

The timer is internal to the DSP and can be programmed to interrupt the DSP at regular intervals. It can also be read by a DSP program, giving a real time clock. The timer interrupts are used to trigger the refresh of the DRAM arrays which make up the PE array.

3.1.7 Controller Bus

The controller bus is the bus of the DSP microprocessor and allows it to communicate with the PE array, switch, input-output port, and external program memory. The data path is 24 bits wide, although the bottom 8 bits are only used for external program memory transfers. The address path is 14 bits wide. The PE array memory is mapped onto the controller bus so it may be read and written in response to instructions from the host. The controller issues instructions to the PE array by writing a 16 bit word into a 512 word segment of its external data memory address space. The offset in the segment specifies a nine bit address in PE memory, which has 512 bits, and the 16 bit data specifies the operation to be performed. The switch configuration is set by writing to registers which are located in the switch chip and mapped onto the controller bus.

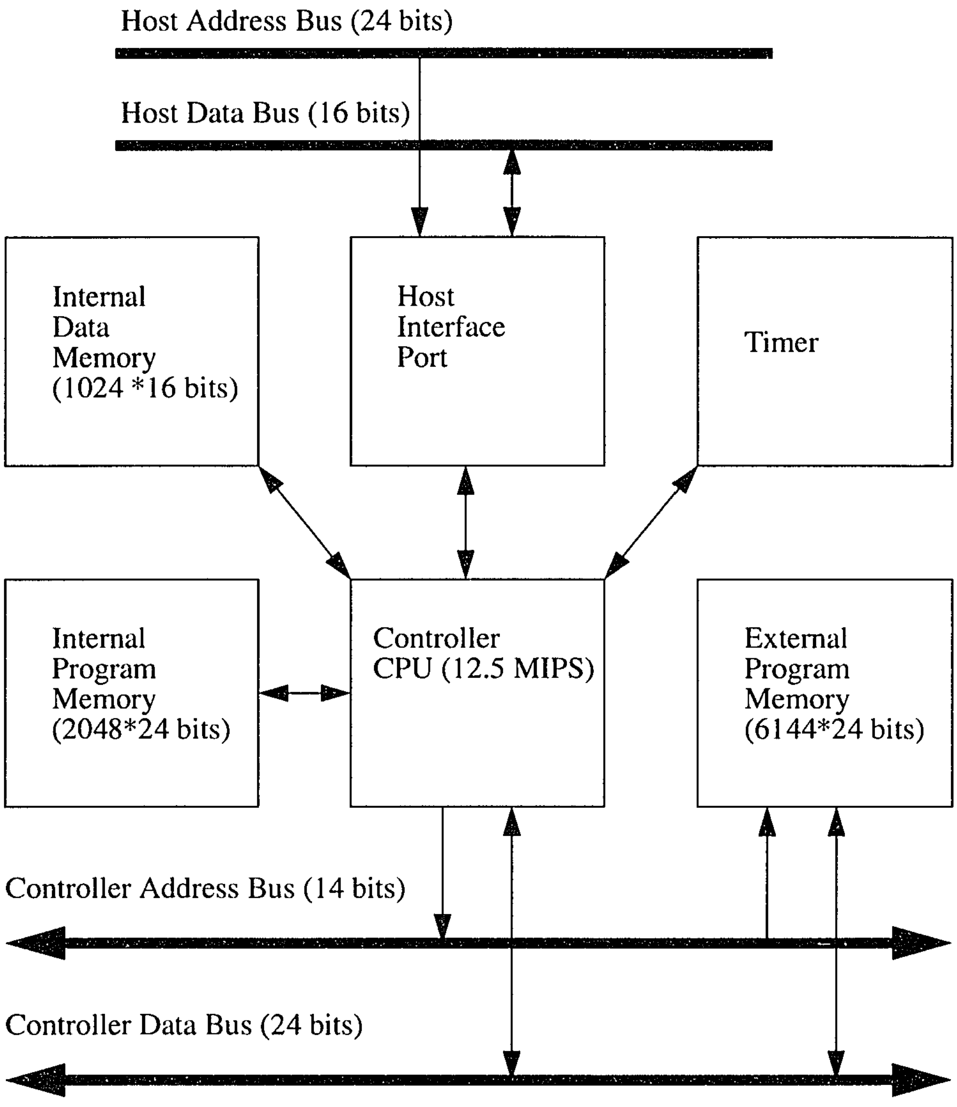

Host Address Bus (24 bits)
Host Data Bus (16 bits)
Internal Data Memory (1024 *16 bits)
Host Interface Port
Timer
Internal Program Memory (2048*24 bits)
Controller CPU (12.5 MIPS)
External Program Memory (6144*24 bits)
Controller Address Bus (14 bits)
Controller Data Bus (24 bits)

Figure 3.2 Controller

3.2 PE Array

This sections describes the PE array, which is the computational engine of the MM32k. It is composed of 32768 processing elements, or PEs. The PE array contains a total of 2 megabytes of memory, organized as 512 bits attached to each PE. The 2 megabytes of PE memory is mapped onto the controller bus. The PE array is connected to the switch via two 64 bit wide data paths, one to and the other from the switch. The PE array instruction register is also on the controller bus. The PE array is shown in the next figure.

3.2.1 PE Page

The PEs are organized into 64 pages, numbered 0 to 63, each implementing 512 individual PEs. Each page contains 262144 bits of DRAM memory, implemented in a 512 by 512 array. Each page also contains three 512 bit registers, called the A register, the B register, and the M register. There is also an A tap pointer register and a B tap pointer register associated with each page, which direct bits from the A register to the switch, and form the switch into the B register. There are four pages implemented per integrated circuit and a total of 16 integrated circuits in the PE array.

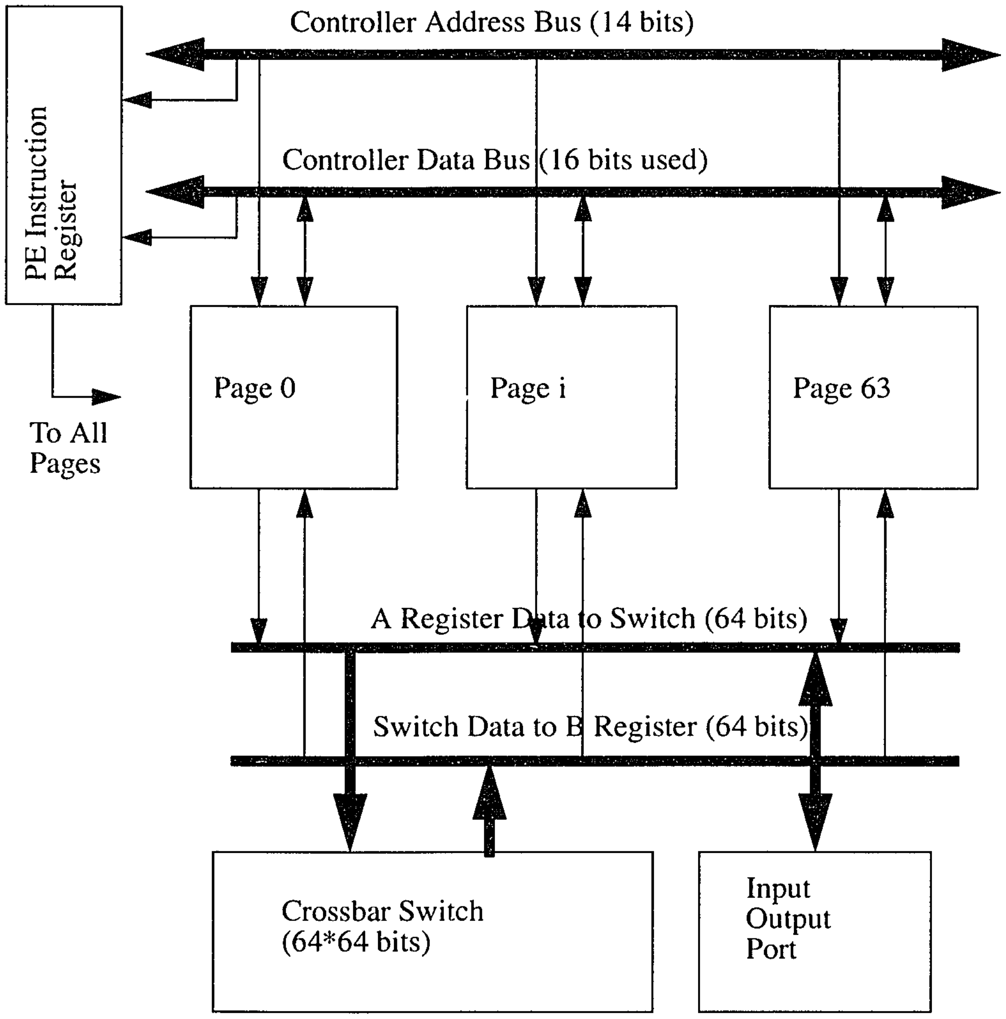

PE Instruction Register
Controller Address Bus (14 bits)
Controller Data Bus (16 bits used)
To All Pages
Page 0
Page i
Page 63
A Register Data to Switch (64 bits)
Switch Data to B Register (64 bits)
Crossbar Switch (64*64 bits)
Input Output Port

Figure 3.3 PE Array

3.2.1.1 DRAM Memory Array

The DRAM memory array for each page contains 262144 bits and is organized as a 512 by 512 array, where the columns correspond to individual PEs within the page. It has two access ports, one which is one bit wide and another which is 512 bits wide. The one bit port is mapped onto the controller bus and so is accessible via memory read and write cycles. The bit of page i is mapped onto bit (i mod 16) of the controller bus. There is a 9 bit row address register which holds the row address and a 9 bit column address register which holds the column address of the currently addressed bit. These registers are both loaded from the least significant address lines of the controller bus. The memory array must be refreshed periodically and can internally generate its own refresh address. The 512 bit wide port can transfer a 512 bit row of data into and out of the A, B, and M registers. In this case, the row address register supplies the row address.

3.2.1.2 A, B, and M Registers

Each page has an A register, a B register, and an M register, each of which is 512 bits long. 512 bits of data can be transferred into or out of these registers from a row of the DRAM memory array in a single cycle. These transfers can also be performed conditionally on a per bit basis. These transfers and conditional transfers are the basis for computation in the PE array. When three bits, one from each of the A, B, and M registers is logically paired with a column of memory in the DRAM memory array, a PE is formed.

3.2.1.3 A and B Tap Pointers

Each page has an A tap pointer register and a B tap pointer register. These registers are 9 bits long and contain pointers into the A and B registers. When the switch is in operation, data is transferred out of the A register bit referenced by the A tap pointer register and into the input of the switch. Simultaneously, data is transferred from the output of the switch into the bit referenced by the B tap pointer register. Both tap pointers are autoincremented as bits are transferred. The transfer takes place at a 35 megahertz rate. Both registers are preset from the least significant bits of the controller address bus.

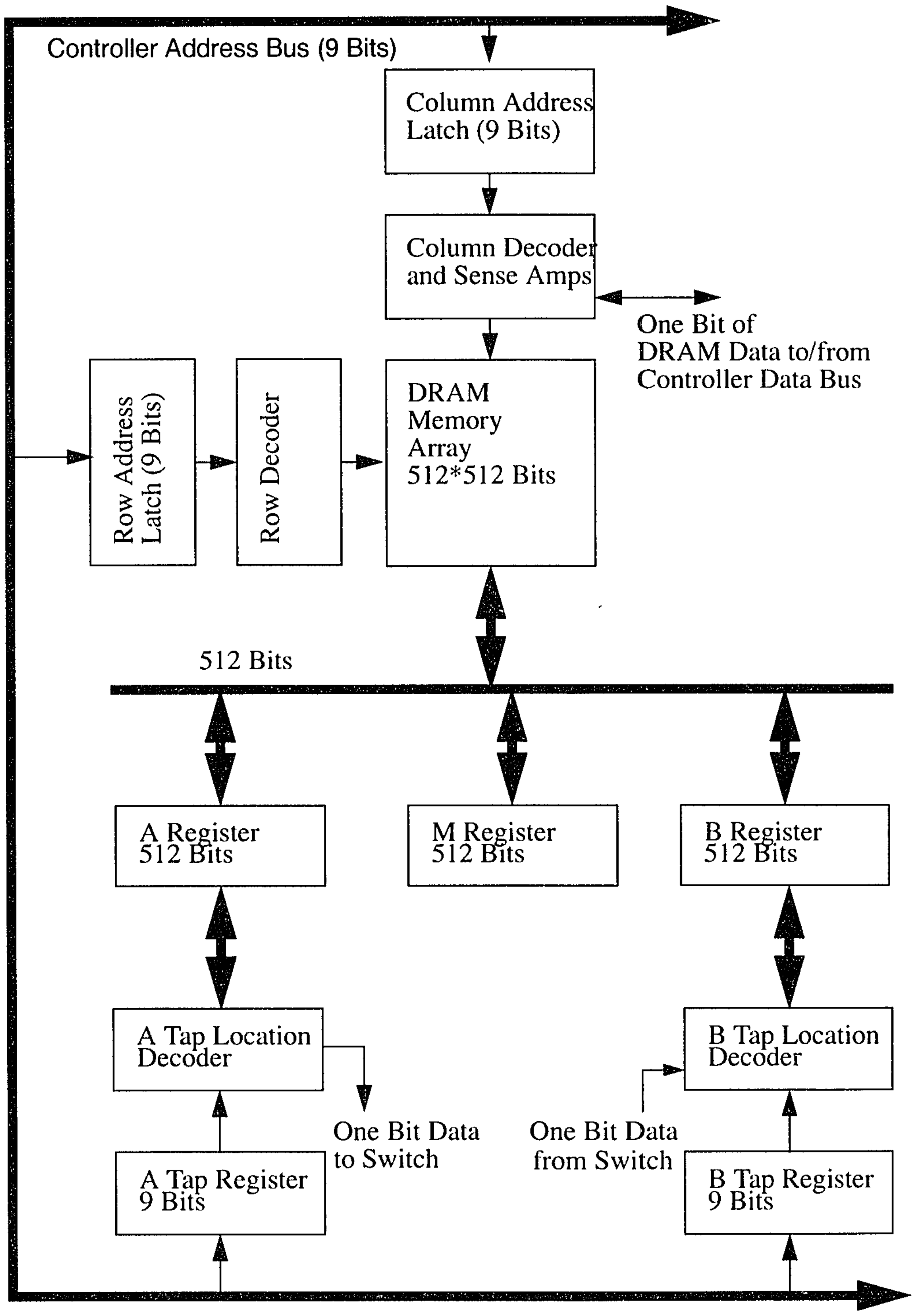


Figure 3.4 PE Page

3.2.2 PE Instruction Register

The purpose of the PE instruction register is to allow the controller to issue instructions to the PEs. It does this by writing PE instruction opcodes into the PE instruction register. The register is mapped into a 512 word long segment of the controller external bus, so the PE array gets an address from the least significant 9 bits of the controller address bus. The PE array usually takes two controller cycles to complete a PE instruction and the controller must not issue another instruction until the earlier one is complete. However, the controller may use the extra cycle to perform other calculations such as deciding which PE instruction to issue next. The controller instruction set has an instruction format that allows a 16 bit data constant, which is a PE instruction opcode, to be written in a single cycle, while autoincrementing the PE instruction address kept in one of the controller CPU address generators.

3.2.3 PE Array Instruction Set

The purpose of this section is to describe the PE array instruction set. Most of the instructions view the hardware as being an array of PEs, but a few view it as being an array of pages. In either case, an instruction issued to a PE is issued to all PEs. An instruction issued to a page is issued to all pages. Most instructions are composed of an opcode and an address. When the instruction is PE oriented, the address usually is a DRAM memory array row address and is 9 bits long. When the instruction is page oriented, the address is usually a column address and is also 9 bits long. All PEs and pages receive the same address.

MEM_TO_A <row_address>

In each page, this instruction reads a row from the DRAM memory array into the A register. Over all pages, 32768 = 64*512 bits are transferred.

MEM_TO_B <row_address>

In each page, this instruction reads a 512 bit row form the DRAM memory array into the B register. Over all pages, 32768 = 64*512 bits are transferred.

A_TO_MEM <row_address>

In each page, this instruction writes the A register into a row of the DRAM memory array. Over all pages, 32768 = 64*512 bits are transferred.

B_TO_MEM <row_address>

In each page, this instruction writes the B register into a row of the DRAM memory array. Over all pages, 32768 = 64*512 bits are transferred.

MEM_TO_M <row_address>

In each page, this instruction reads a 512 bit row of the DRAM memory array into the M register. Over all pages, 32768 = 64*512 bits are transferred.

INV_MEM_TO_M <row_address>

In each page, this instruction reads a 512 bit row of the DRAM memory array, complements the row, and copies the result into the M register. The bits in the DRAM memory array are left unchanged. Over all pages, 32768 = 64*512 bits are read and complemented.

M_TO_MEM <row_address>

In each page, this instruction writes the M register into a row of the DRAM memory array. Over all pages, 32768 = 64*512 bits are transferred.

INV_M_TO_MEM <row_address>

In each page, this instruction complements and writes the M register to a row of the DRAM memory array. The bits in the M register are left unchanged. Over all pages, 32768 = 64*512 bits are complemented and written.

A_TO_M

In each page, this instruction copies the A register to the M register. Over all pages, 32768 = 64*512 bits are copied.

B_TO_M

In each page, this instruction copies the B register to the M register. Over all pages, 32768 = 64*512 bits are copied.

M_TO_A

In each page, this instruction copies the M register to the A register. Over all pages, 32768 = 64*512 bits are copied.

M_TO_B

In each page, this instruction copies the M register to the B register. Over all pages, 32768 = 64*512 bits are copied.

CLR_M

In each page, this instruction sets the M register to zero. Over all pages, 32768 = 64*512 bits are set to zero.

IF_M_MEM_TO_A <row_address>

In each page, this instruction reads a row of the DRAM memory array, and if the corresponding M register bit is set, copies it into the A register. Over all pages, 32768 = 64*512 bits are set to zero.

IF_M_MEM_TO_B <row_address>

In each page, this instruction reads a row of the DRAM memory array, and if the corresponding M register bit is set, copies it into the B register. Over all pages, 32768 = 64*512 bits are set to zero.

IF_M_A_TO_MEM <row_address>

In each page, this instruction writes the A register to the DRAM memory array if the corresponding M register bit is set. Over all pages, 32768 = 64*512 bits are set to zero.

IF_M_B_TO_MEM <row_address>

In each page, this instruction writes the B register to the DRAM memory array if the corresponding M register bit is set. Over all pages, 32768 = 64*512 bits are set to zero.

MAP_ROW_TO_BUS <row_address> <page_group>

This instruction maps a row of DRAM memory array in 16 pages onto a 512 word segment of the controller bus. Page i is mapped to bit (i mod 16) of the controller data bus. Controller read and write bus cycles can read and write bits in the mapped pages, where the offset in the 512 word segment specifies the column number within each page. The row address specifies which row in the PE memory is mapped. The page group is an integer between 0 and 3 which specifies which group of pages, 0 to 15, 16 to 31, 32 to 47, or 48 to

63, are mapped. In total, 512 16 bit words from 16 pages are mapped onto the bus. Any other instruction unmaps the pages from the bus.

REFRESH_ROW

This instruction performs a refresh cycle on the DRAM memory array of all pages. The refresh address is maintained internally to each page. All 512 rows of the DRAM memory array must be refreshed at least once every 8 milliseconds.

SET_TAP_A <column_address>

This instruction sets the A register tap pointer register on all pages. This register points to the bit of the A register where the next bit of data sent to the switch will come from.

SET_TAP_B <column_address>

This instruction sets the B register tap pointer register on all pages. This register points to the bit of the B register where the next bit of data from the switch will be placed.

ENABLE_SWITCH_TRANSFER

On all pages, this instruction causes data to be transferred out of the A register, through the switch, and into the B register. Data is removed from the bit in the A register referenced by the A register tap pointer and placed into the bit in the B register referenced by the B register tap pointer register. Both tap pointer registers are incremented for each bit transferred. A bit from an A register on a particular page will in general be transferred into the B register on a different page. The transfer continues until explicitly stopped by the DISABLE_SWITCH_TRANSFER instruction. The transfer takes place at a 35 megahertz rate and may transfer up to 512 bits per page. The peak switch throughput is 280 = 35*64/ 8 megabytes per second.

DISABLE_SWITCH_TRANSFER

This instruction stops the transfer of data through the switch, which was started by the ENABLE_SWITCH_TRANSFER instruction.

3.3 Switch

The purpose of the switch is to allow the pages to exchange data and to move data around within a page. This data exchange operation is used to move data from one PE to another and to compute cumulative results, such as a global sum, for the entire PE array. The switch is a full crossbar with 64 inputs and 64 outputs and can form a connection from any page to any other page. Up to 64 different one way connections can be formed simultaneously. In operation, the B registers are filled with data from the A registers to which they are connected. The switch can transfer data at a 35 megahertz rate, which corresponds to an overall data transfer rate of 280 megabytes per second.

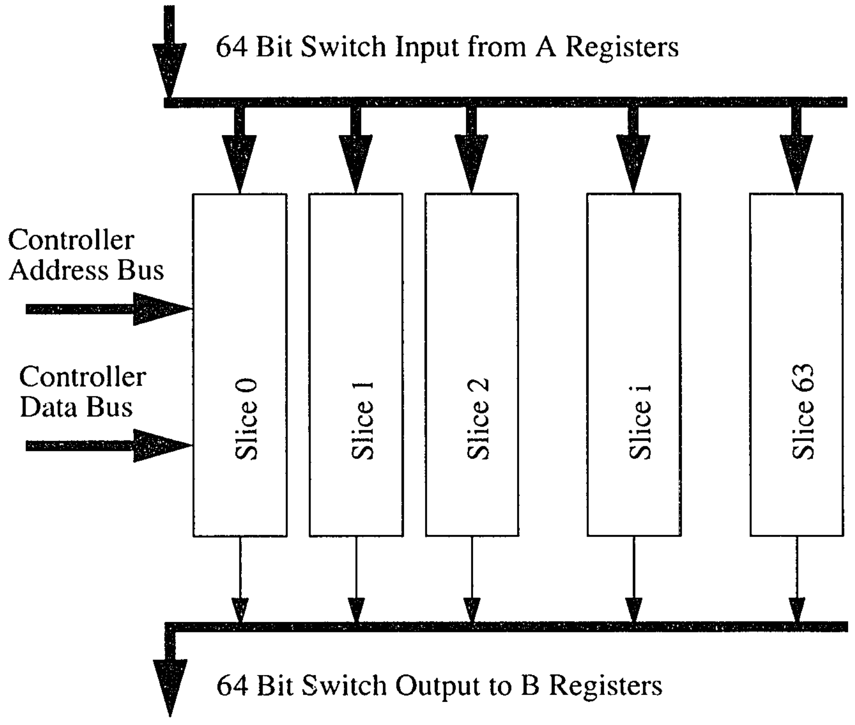


Figure 3.5 Crossbar Switch

3.3.1 Switch Connection to Pages

The switch has 64 inputs and 64 outputs. The inputs are connected to the A register tap decoder, which is configured as an output and provides the bit of A register data referenced by the A register tap pointer. The outputs of the switch are connected to the B register tap decoder, which is configured as an input and stores the bit of data provided by the switch into the B register location referenced by the B register tap pointer. To improve performance, the data path through the switch is pipelined. The path from the pages to the switch has a pipeline register, the switch has an internal pipeline register, and the path from the switch to the pages has a pipeline register.

3.3.2 Switch Slice

The switch is composed of 64 identical and independent slices. Each slice has 64 inputs, corresponding to 64 switch inputs and one output. Each slice has a 6 bit address register, which identifies the input which is to be connected to the output. The slice output has a one bit pipeline register. The address of each slice can be set independently. Each slice also has a 6 bit staging register for the address register. The purpose of the staging register is to allow a complete set of slice addresses to be loaded into the switch while the switch is transferring data, and then change to the new set in a single cycle.

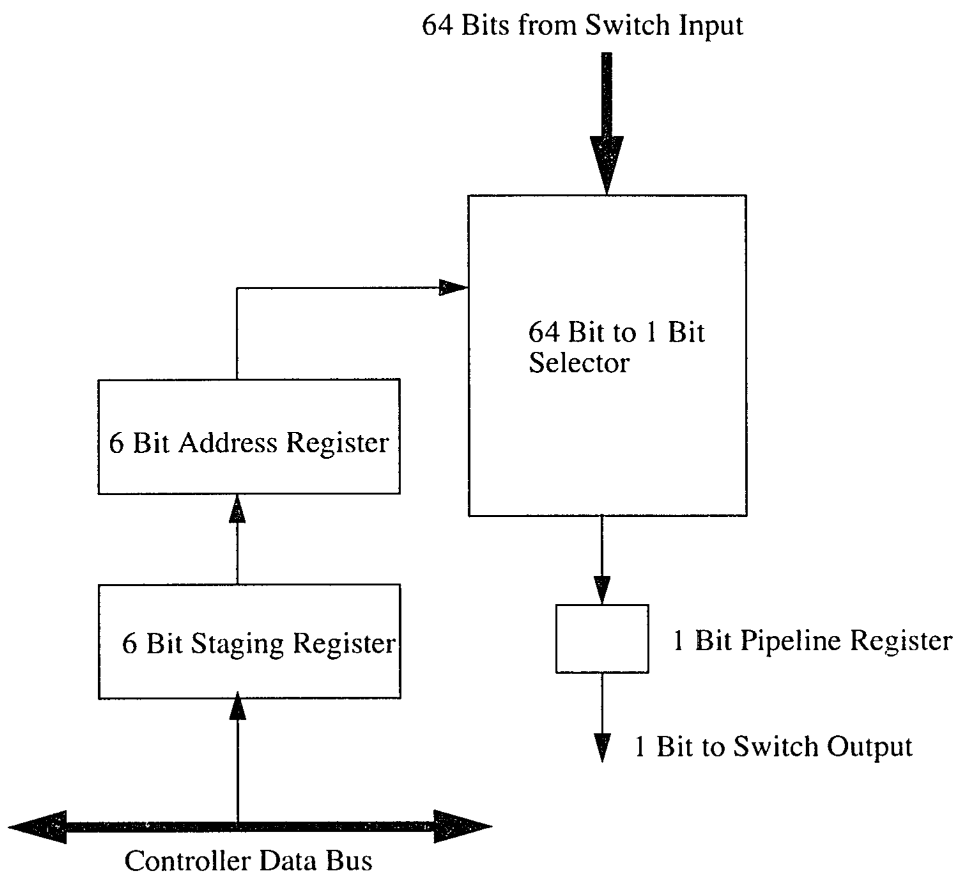


Figure 3.6 Crossbar Switch Slice

3.3.3 Switch Setup

The 64 address registers cannot be loaded directly, but only from the staging registers. The staging registers for each of the 64 slices are mapped into the controller bus in a 64 word block. Since the input to output mapping of the switch is usually regular and incremental, this makes it easy for the controller to set the switch up in a single cycle loop. The setup time for the switch is about 6 = 64/12.5 microseconds and the time to transfer all 512 bits from the A register to the B register is 15 = 512/35 microseconds. Ideally, the setup time should be much shorter than the transfer time, but it isn't. This isn't as bad as it might seem because several transfers are usually made for each switch setup. The switch setup time is approximately half as short as a single transfer, so in some circumstances the switch can be reconfigured for the next transfer while the current transfer is in progress by using the staging registers.

3.3.4 Switch Operation

The switch allows blocks of data to be transferred from the A registers to the B registers in all pages. This is done under control of the controller in a number of steps. First, the switch must be set up to connect the pages which are the source of data to their corresponding destination pages. Next, the A and B tap pointers must be set up to mark the source of data in the A registers and the destination of data in the B registers. All A register tap pointers get the same address and all B register tap pointers get the same address. Due to the pipeline registers and clocking circuitry, the B register tap pointers are actually set up with a value two less than the first destination address, but this detail will be ignored elsewhere in this document. Then, the data is transferred into the A register, usually from the DRAM memory array. Next, the ENABLE_SWITCH_TRANSFER PE array instruction is issued. Then the controller waits while the block of data is transferred, 64 bits per cycle, at a 35 megahertz rate. As the bits are transferred, the A and B register tap pointers are autoincremented. The controller performs an idle timing loop to wait until all bits have been transferred, taking pipeline delay into account. It then issues the DISABLE_SWITCH_TRANSFER PE array instruction and the transfer is stopped. At this point, a block of data which was in the A register of one page is now in the B register of another page or realigned within the B register of the same page.

Due to the pipeline delay and inaccuracies in the controller timing loop, the switch is usually allowed to run a few cycles longer than is required to transfer the bits, so the B register will contain more data bits than were intended to be transferred. These extra data bits are removed by loading the M registers with a mask and conditionally transferring only the valid data bits to a destination memory location.

3.3.5 Shifting Data in a Page

The page hardware cannot directly move data from one location to another in the A, B, or M registers. This corresponds to moving data from one PE to another within the same page. However, this can be accomplished with an ordinary transfer by setting the switch mapping to identity and setting the A and B register tap addresses to the intra page displacement. This method is slow, because moving data to a neighboring PE should take one cycle, but instead takes 512 cycles. A consequence of this is that data can be moved to any PE in an amount of time independent of the distance the PE is from the source, but that the time to do this is slow.

3.4 Input-Output Port

The purpose of the input-output port is to provide a high speed data path in to and out of SIMD memory in the PE array. The port is 64 bits wide and can input or output data at a 280 megabyte per second rate. Physically, the port is a connector attached to the lines between the A register outputs of the pages and the switch input.

To input data via this port, the switch is set up so that the mapping function is identity and the B register tap pointers are set to contain the location in the B register where the first data will be placed. The data is placed onto the port and clocked into the switch by an external data clock. As they emerge from the switch, the 64 data bits are placed into the B registers and the B register tap pointers are incremented. When filled, the B register is usually written into the DRAM memory array. To output data via this port, the A register tap pointer is set to contain the A register address from where the first bit will come. The data are clocked out of the A registers 64 bits at a time by an externally generated output clock. Each time 64 bits are read out of the A registers, the A register tap pointers are incremented. This process continues until all the of the data bits have been transferred. The input-output device must count the number of transfer cycles and stop when the transfer is complete.

The port can support continuous transfers, both in and out, by using double buffering. In the case of input, the B register is divided into two sections, bits 0 through 255 and bits 256 through 511. When the first section is filled, it is written into the page memory array using the IF_M_B_TO_MEM instruction, while the second half of the B register is filled. When the second half is filled, it is written into the page memory array while the first half is being refilled. The B register tap pointer wraps in a circular fashion when it moves beyond bit 511 in the B register. This process may continue indefinitely or until the DRAM memory array is filled. In the case of output, the A register is loaded with the first 512 bits of data and the transfer is started. When the first half of the A register is empty, it is refilled while data from the second half is being transferred. When the second half is empty, it is filled using the IF_M_MEM_TO_A instruction while data form the first half is being transferred. In both the input and output cases, the M register must be filled with bits marking which half of the register to read or write.

The port can also support narrower input-output port widths. This is done by having the switch realign the data and having M register bits mark which pages are to receive data.

No PE array computation, other than that directly associated with implementing the input-output transfer, can be performed while the transfer is in progress.

The width of the port could have been doubled to 128 bits by using both the A register and B registers simultaneously. This would have doubled the peak throughput to 560 megabytes per second, but was not done because of the size of the required connector.

Chapter 4
LOGICAL MACHINE

The purpose of this chapter is to describe a logical model for the MM32k, which can be implemented by the hardware described in the last chapter. The SIMD computer described by this logical model can be programmed directly by a user, but also serves as the basis for the C++ programming environment which will be described in the chapter after this one. The model is defined by a logical piece of hardware and the vector oriented instruction set by which it is programmed. The logical piece of hardware is an abstraction of the actual hardware and the vector instruction set is implemented by the microcode written for the controller.

This chapter also describes the vector instruction set implemented on the logical machine. The logical machine is a vector coprocessor for the host computer. The host can place a vector of integers into the PE memories and perform vector arithmetic on the vectors using the full computational power of the MM32k. The number of bits of precision per element of a particular vector is variable. The number of elements in a vector may also be larger than 32768.

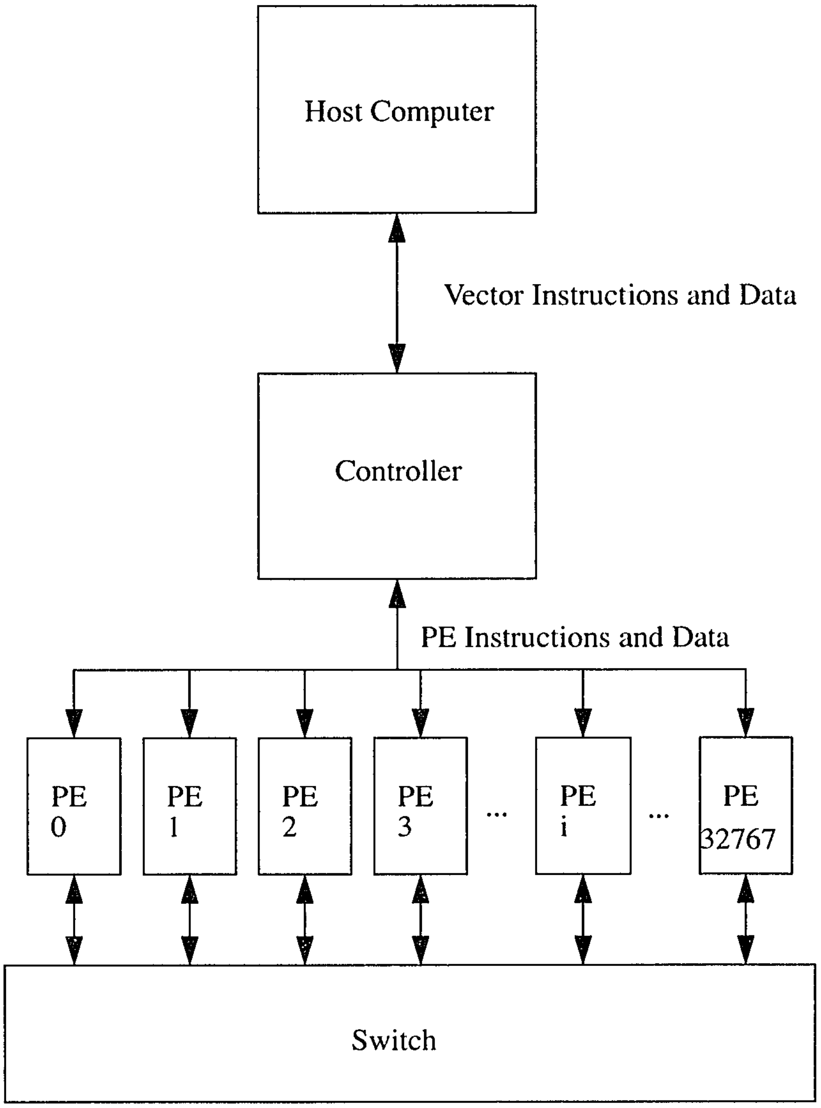


Figure 4.1 Logical Machine

4.1 Logical Machine

The logical machine is shown in the figure above and represents a SIMD architecture. The 64 pages are viewed as an array of 32768 PEs. Each PE contains 3 one bit registers, 512 bits of memory and a connection to the switch. The switch is no longer viewed as a full crossbar, but as a barrel shifter which can shift 32768 bits of data an arbitrary distance across the PE array. The processors are numbered from 0 to 32767.

In the logical machine, the host sends the MM32k vector instructions, which the controller places in an instruction queue. The instructions are then fetched and executed by the controller as fast as it can process them. The controller issues PE instructions to the PEs to implement the vector instructions. The execution of a single vector instruction causes a large number of PE instructions to be issued. A vector add of two eight bit numbers would require approximately 80 PE instructions to be issued by the controller. A vector multiply of two eight bit image arrays containing 262144 pixels each would cause approximately 6600 PE instructions to be issued by the controller.

4.1.1 Logical PE

A logical PE is shown in the next figure. It contains a one bit A register, a one bit B register, a one bit M register, a one bit ALU, 512 bits of memory, and a two way connection to the switch. There are 512 PEs implemented per page. Each bit of the 512 bit long A, B, and M registers on a page corresponds to a particular PE. Each column of the 512*512 bit DRAM memory array on a page corresponds to the 512 bit memory of a particular PE. The switch connection is shared for all 512 PEs on a page and is implemented by a combination of the A and B tap pointers and the A and B tap decoders. For all PEs on a page, there is only one physical connection to the crossbar switch. A particular PE i is implemented on page i/512 by bit i%512 of the A register, bit i%512 of the B register, and bit i%512 of the M register. The memory is implemented by column i%512 of the DRAM memory array. The 512 memory locations in the PE are numbered 0 to 511 and correspond to row addresses in the DRAM memory array. The one bit ALU for all PEs is implemented by the PE instruction set which transfers bits between the DRAM memory array and the A, B, and

M registers. When instructions are executed, the controller supplies the 9 bit PE memory address.

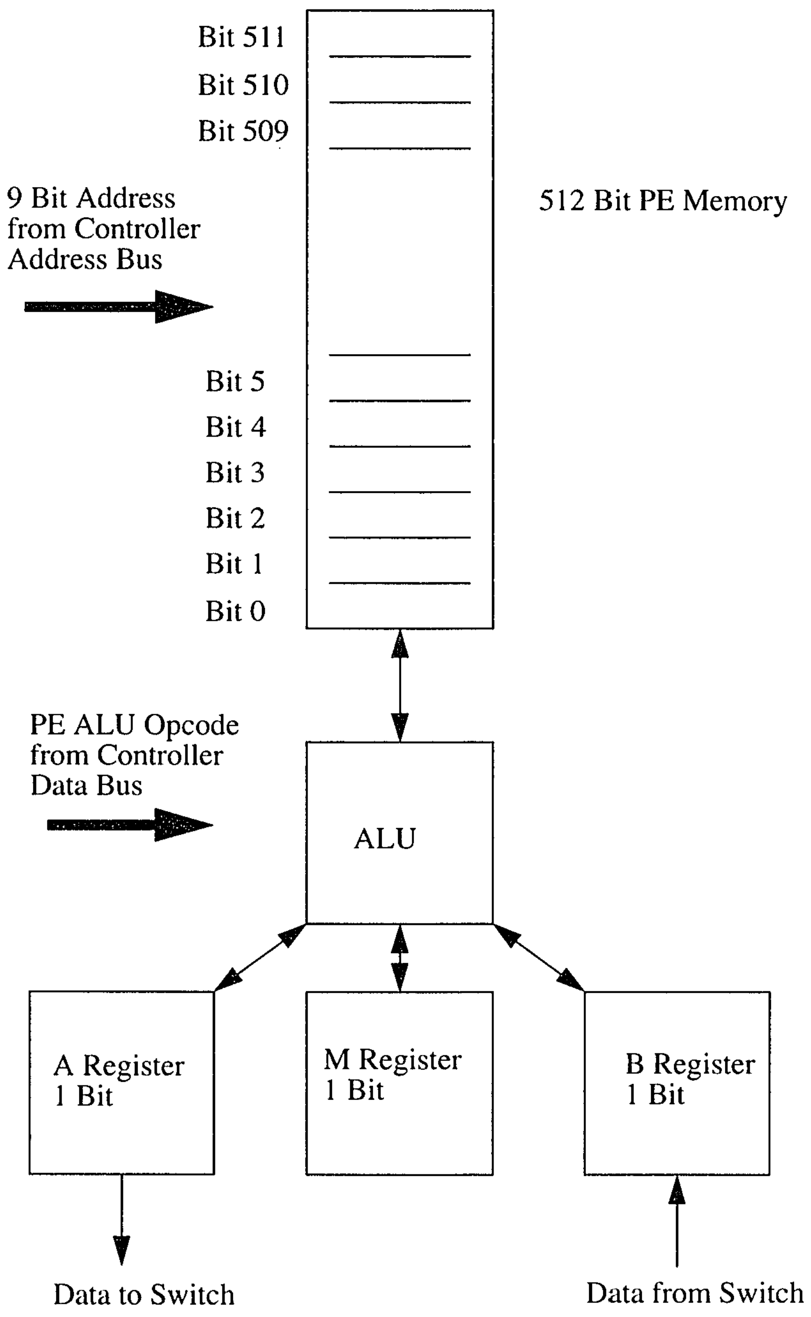


Figure 4.2 Processing Element

4.1.2 Logical Switch

The logical switch can take 32768 bits of data, one bit from each PE, and move those bits to different PEs in a single logical switch cycle. The mapping function performed is a rotation. A bit in PE i is moved to PE (i+D)%32768, where D is the shift distance. Negative rotations can be performed by doing a positive rotation by D = (32768- NEGD). The time to perform a shift is independent of the value of D. Data is moved at a rate of 280 megabytes per second. The logical switch is implemented by a combination of the crossbar switch, the A and B register tap pointers, and the A and B registers.

4.2 Data Organization

In this section, the organization of data in the memories of the PEs is described. The logical machine represents vectors of integers by distributing them across the memories of the PEs. First, the layout of bits in the memory of a single PE and how they form an integer is described. Then, the layout of words to form a vector is described. Finally, the layout of vectors longer than 32768 elements is described.

4.2.1 An Integer is Formed from Bits

This section describes how an integer is formed in PE memory. Integers are represented as a contiguous group of bits, in 2's complement form, in the memory of a PE. As an example, see the next figure where two integers are stored. The integer at address 150 is 5 bits long and has a value of -13. The integer at address 200 is 6 bits long and has a value of 22. The number of bits in an integer can vary between one and 512. If the length of an integer is one bit, then 512 one bit integers could be stored in the memory of a single PE. If the length of an integer is 512 bits, then only one integer could be stored in the memory of a single PE. Although one bit integers are frequently used as flags, a more practical integer length is 8 bits, corresponding to a typical pixel during machine vision processing. In this case 64 8 bit integers could fit into the memory of a single PE.

An advantage of variable bit length integers is that they need only occupy as much storage as is needed to avoid overflow. For instance, if an 8 bit integer were added to a 5 bit integer, then the result could be stored as a 9 bit integer without danger of overflow. Another advantage is that only 9 cycles would be needed to compute the result.

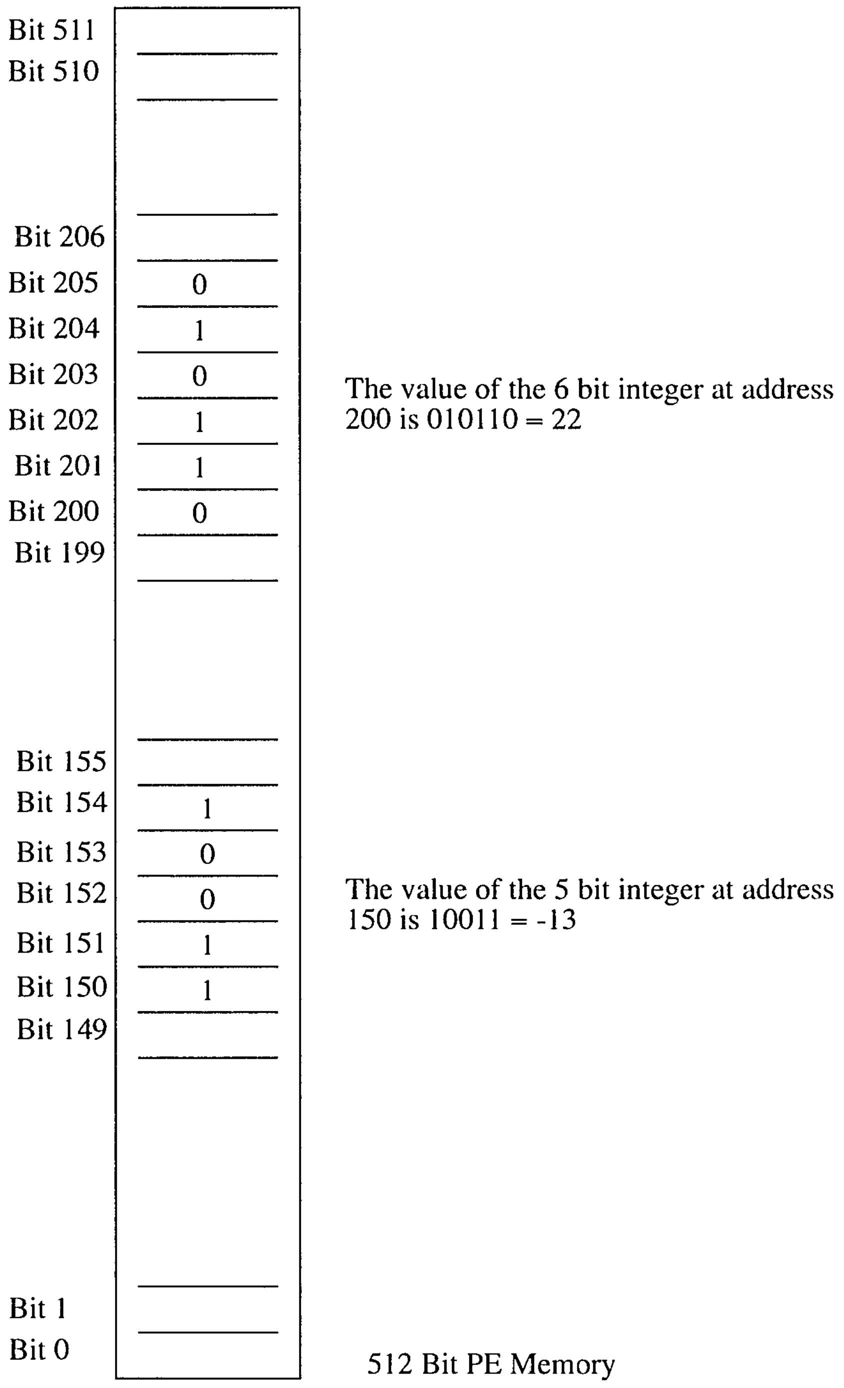


Figure 4.3 Data Format in PE Memory

4.2.2 A Vector is Formed from Integers

This section describes how a vector is formed from integers. A vector is an array of integer elements, which are stored across the PE memories. The first element is numbered 0 and is assigned to the memory of PE 0, the second is numbered 1 and is assigned to PE1, and so on until the vector is complete. All of the integers of the vector have the same number of bits of storage. All of the integers of a vector are assigned the same PE memory address, provided that the vector is less that 32768 elements long. In the figure below, there is a 5 bit vector named x which is located at address 51 in PE memory. The vector is 32768 elements long and fills the PEs.

The notation is x[i].j, where i is the vector index and j is the bit number

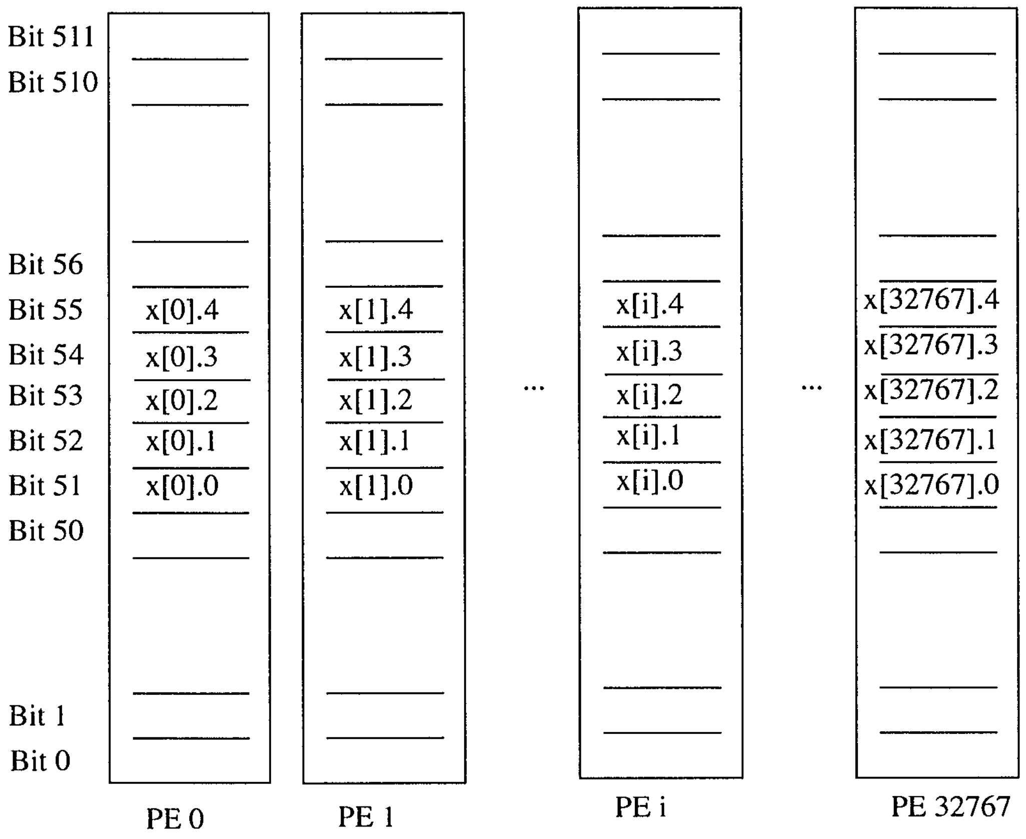


Figure 4.4 Vector Format Across PEs

4.2.3 Vectors may be Longer than the Machine is Wide

A vector may be longer than 32768, which is the number of PEs in the machine. All vectors must be a multiple of 32768 elements long. If not, then their length must be rounded up to the next higher multiple of 32768 and the storage associated with the unused PE memory locations is wasted. A vector longer than 32768 is represented in PE memory by wrapping the elements around the end of the PE array and into the next higher PE memory locations. This process is repeated until all elements of the vector are assigned a location in

PE memory. A particular element x[i] of a vector is assigned to the memory of PE i%32768. A bit in that element x[i].j is assigned to address b+(i/32768)*n+j, where b is the base address of the vector x in PE memory and n the number of bits per element of x. This is illustrated in the figure below, where a vector x, which is 98304 = 3*32768 elements long with 5 bit words, is placed in PE memory at address 51.

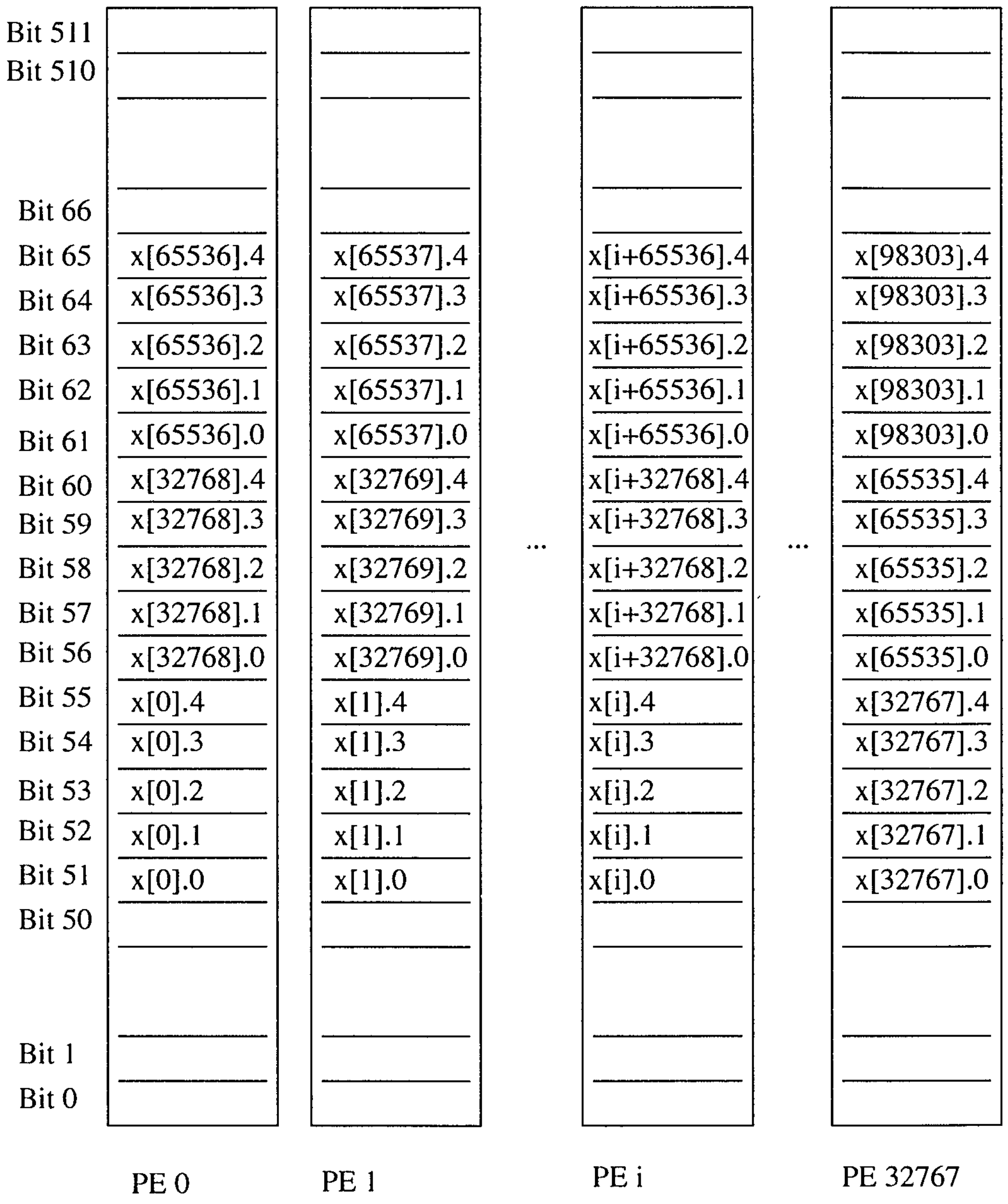


Figure 4.5 Vector Longer than 32768

4.2.4 Vectors and Virtual PEs

Virtual PEs are a different way of dealing with situations where the problem size is bigger than the machine. When using virtual PEs, the memory of each PE is divided into two or more equal size pieces, yielding two or more virtual processors, each with half or less of the memory and computational power of the original PE. The PEs are divided into as many pieces as are necessary to handle the problem. The total memory and computational throughput of the machine has not changed, just the logical organization. Some SIMD machines provide this capability to save the application programmer from having to do it in the application code.

On the MM32k, long vectors provide the same functionality as virtual PEs. The machine may store and compute with vectors of different lengths in the same program by creating and manipulating them with the desired size. This is conceptually different than logically reconfiguring the machine to the size necessary to contain a group of vectors used together in a calculation.

Vectors of different lengths may be used together in the same arithmetic or logic operation. In this case, either the longer vector can be assumed to be truncated and the result is the same length as the shorter vector or the higher numbered elements of the shorter vector are assumed to be undefined and the higher order elements of the result may need to be handled specially.

4.3 PE Programming

This section describes how a PE is programmed to do arithmetic and logic operations. The PE can manipulate data only one bit at a time, so the object is to compose a sequence of one bit operations which will perform a word operation. The PE instruction set is very simple and is not well suited to either arithmetic or logic. This section will show how bitwise NOT and OR functions can be performed. If the PE can perform NOT and OR, then it is functionally complete and can perform any logic function. A more elaborate operation, ADD, is also described. Finally, a methodology is described for building programs using the PE instruction set.

4.3.1 PE Instruction Set

The PE instruction set is the same as that described in the previous chapter for a page and is listed below in a different format. Only the page instructions which are logical PE instructions are listed. The expression **mem(<address>)** refers to the bit of PE memory at location **<address>**.

A = mem(<address>)	**copy memory to A register**
B = mem(<address>)	**copy memory to B register**
mem(<address>) = A	**copy A register to memory**
mem(<address>) = B	**copy B register to memory**
M = mem(<address>)	**copy memory to M register**
M = ~mem(<address>)	**copy complement of memory to M register**
mem(<address>) = M	**copy M register to memory**
mem(<address>) = ~M	**copy complement of M register to memory**
M = A	**copy A register to M register**
M = B	**copy B register to M register**
A = M	**copy M register to A**
B = M	**copy M register to B**
M = 0	**set M register to zero**
if M then A = mem(<address>)	**if M register is set, copy memory to A register**
if M then B = mem(<address>)	**if M register is set, copy memory to B register**
if M then mem(<address>) = A	**if M register is set, copy A register to memory**

if M then mem(<address>) = B **if M register is set, copy B register to memory**

As a summary, the instruction set can move bits between registers, it can generate the constant 0, it can complement a bit, it can move bits between registers and memory, and it can conditionally move bits between registers and memory.

4.3.2 The Instruction Set is Functionally Complete

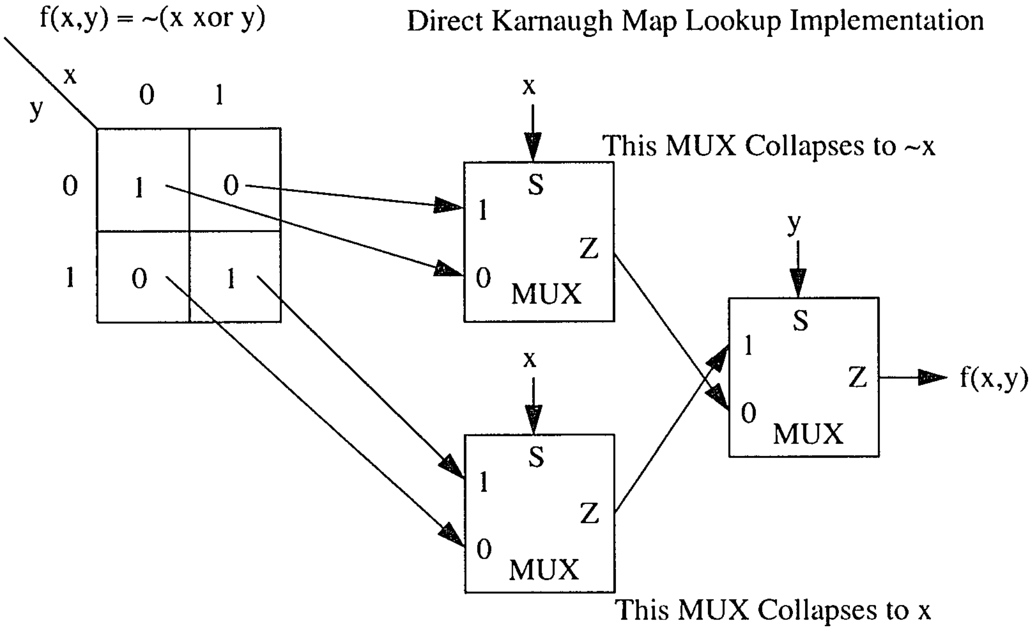
The instruction set listed above is functionally complete in that it can perform any boolean arithmetic function, and can therefore perform any arithmetic or logic operation which a traditional computer can perform. To demonstrate this, it is necessary to show that it can perform the NOT and OR operations. If it can, then any other operation can be composed of combinations of these two. The machine can perform the NOT operation directly, so a short program will demonstrate that it can perform the OR operation. The following three step program will compute the bitwise OR of two bits located in PE memory at locations 10 and 11 and leave the result in the A register. The original value of the M register is destroyed.

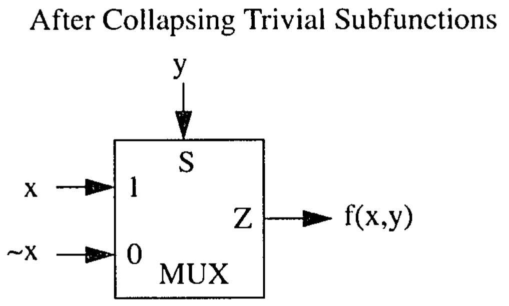
A = mem(10)
M = mem(11)
if M then A = mem(11)

Since the instruction set can perform OR and NOT, it is functionally complete, but how efficient is the instruction set? How many instructions does it require to perform useful work?

4.3.3 PE Programming Techniques

Assembling a logic function from a series of OR and NOT operations will not usually produce the shortest program. This section describes techniques for building shorter programs. The basic building block provided by the PE instruction set is the conditional copy, which is equivalent to a two input selector logic block. Therefore, the first technique is to try to formulate the logic function as a series of two input multiplexers. This can be done in a direct fashion by creating a tree of multiplexers to do a table lookup on the karnaugh map of the function. The inputs to the multiplexers are the constants in the karnaugh map and the control variables are the input variables of the function. This tree can then be collapsed by replacing multiplexers which implement trivial functions with the variable or constant directly. All of the first layer of multiplexers in the tree will collapse into either the constant 0, the constant 1, the input variable, or the complement of the input variable. A simple example of this is given in the figure below, where the complement of the exclusive or function is computed.





Program to Compute f(x,y)

```
M = ~mem(x)
A = M
M = mem(y)
if M then A = mem(x)
```

Figure 4.6 PE Program to Compute ~(x XOR y)

If the function has a lot of input variables, it may be useful to try to partition the function into subfunctions which can be combined to create the complete function. The subfunction may be used in more than one place as the complete function is built. This may also be useful if multiple output functions are to be computed from a single set of input variables.

Another technique is that of partitioning karnaugh maps. A simple case of this is the direct karnaugh map table lookup just described, where the karnaugh map is recursively partitioned by the input variables into subfunctions, until the subfunctions are trivial. However, the karnaugh map may be partitioned by subfunctions instead of by input variables.

In the figure below, a block diagram of a one bit adder is developed by computing an intermediate term ~(x xor y) and by using it to help generate the sum and carry output terms. The sum is implemented by another ~XOR block. The carry is implemented by partitioning the carry karnaugh map and reducing the terms going to the carry output multiplexer to y and c.

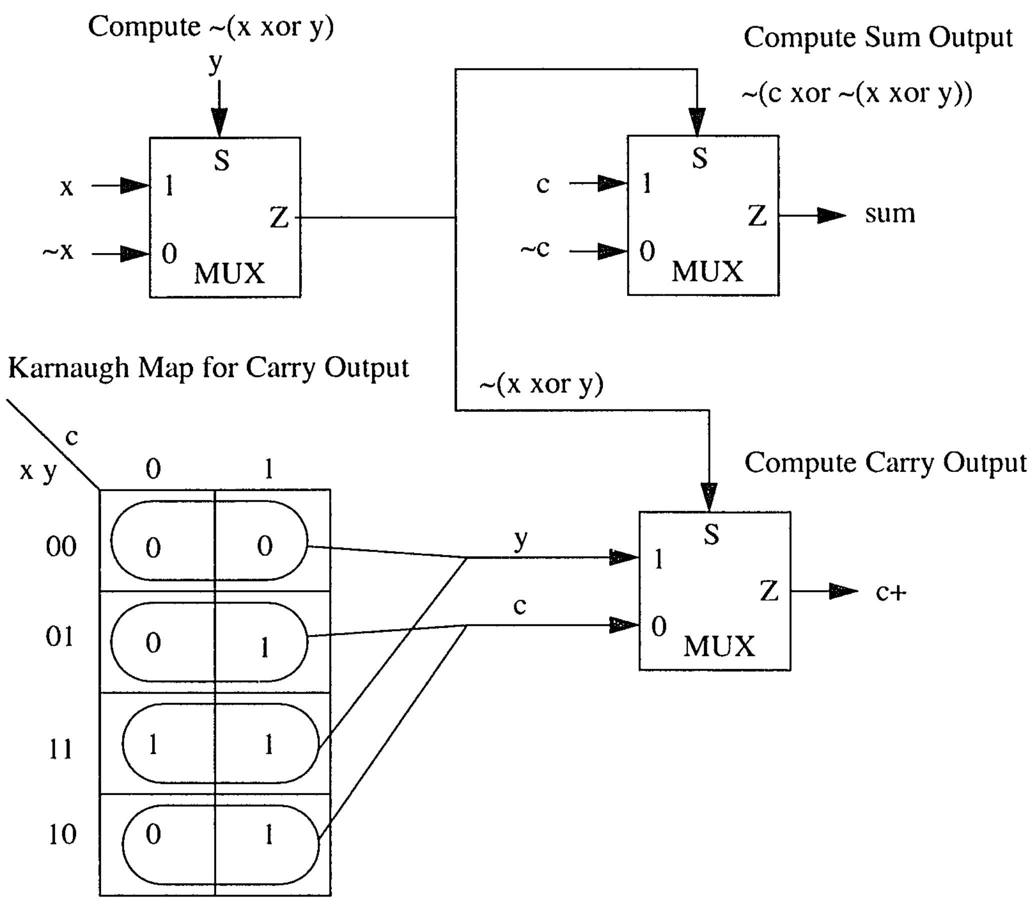


Figure 4.7 PE Program to Implement a One Bit Adder

4.3.4 A Program to Add Two Bits

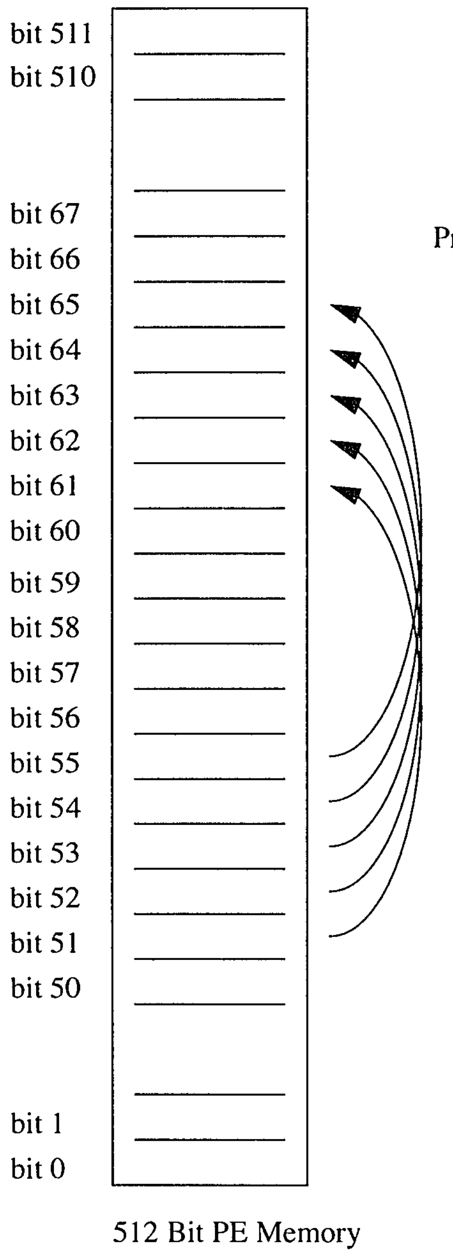
This section will describe a program which can implement the one bit addition just described in the block diagram. It takes two bits, located at locations 10 and 20, adds them together, and leaves the result at location 30. The carry bit is kept in the B register before the program is called and is updated and left in the B register when the program completes. The contents of the A and M registers are destroyed.

```
M = ~mem(10)
A = M
M = mem(20)
if M then A = mem(10)
M = B
mem(30) = ~M
M = A
if M then mem(30) = B
if M then B = mem(10)
```

This program takes nine instructions to perform an operation which involves reading two bits from memory and writing one bit to memory. Ideally, this program should take three instructions, but it actually takes nine. Therefore, when performing addition, the instruction set is 33% efficient with respect to memory bandwidth.

4.3.5 Bit Serial Arithmetic

Word operations are performed by a series of one bit operations. The technique of performing word arithmetic one bit at a time is called bit serial arithmetic. For example in the figure below, a 5 bit word at location 51 is copied to location 61 by a program which is 10 instructions long. The arcs in the figure indicate the dataflow.



Program to Copy 5 Bit Word

```
M = mem(51)
mem(61) = M
M = mem(52)
mem(62) = M
M = mem(53)
mem(63) = M
M = mem(54)
mem(64) = M
M = mem(65)
mem(65) = M
```

Figure 4.8 PE Program to Copy a Word

Another example is given in the figure below. In this case, the 3 bit number 3 at location 51 is added to the 3 bit number -1 at location 56 and the 4 bit result 2 is placed at location 63. In this example, four adder blocks are used to indicate the one bit addition program which was given earlier. This program would take 37 = 4*9+1 cycles to execute. The addition of each bit requires 9 cycles and the extra cycle is to set the carry bit in the B register to zero initially. The addition is assumed to be of 2's complement numbers, so the sign bits of the two operands are duplicated to produce the last bit of the result.

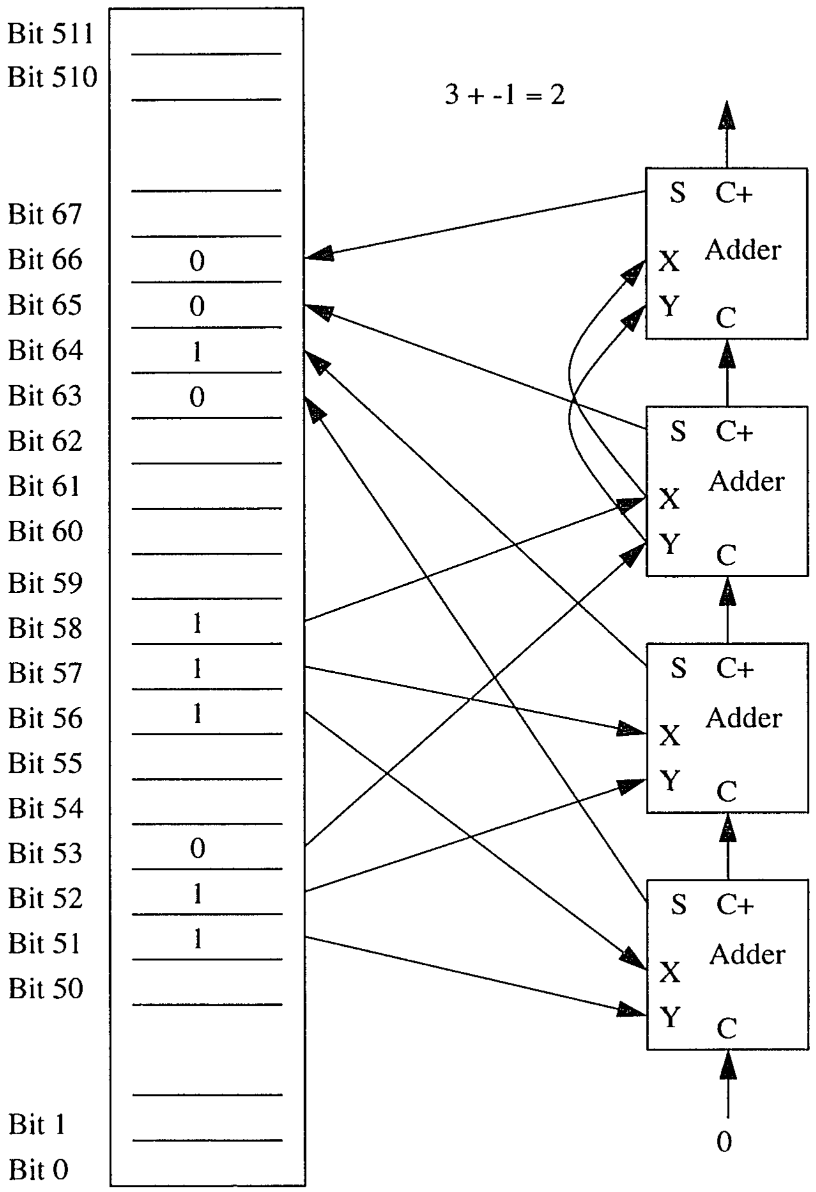


Figure 4.9 PE Program to Add Two Three Bit Numbers

4.3.6 Align and the Switch

The switch can be programmed to form a logical barrel shifter 32768 bits wide, one for each logical PE. The logical operation of the switch is to shift data from PE i to PE (i+D)%32768. This is done by setting up the switch and tap registers and moving the data in blocks at 35 megahertz. The physical switch is set up as a 64 bit wide barrel shifter and determines how may 512 PE pages data is moved. If the distance D by which data is to be moved is a multiple of 512, then a single switch setting can be used to move all the data. Usually this is not the case. For a particular PE i, the amount of rotation in the switch should be (i+D)/512. As i varies, this number assumes two values, so two switch settings are used to move two different groups of PE data. The data in PEs with index number i such that 0<= i%512 < 512-D%512 form one block and the data in PEs with index number i such that 512-D%512 <= i%512 <512 form the other block. This is illustrated in the next figure, where data is to be moved from a source vector to a destination vector by D=400 PEs. The data labeled A, C, M, andY is moved within a single page and the data which is labeled B, D, L, N, X, and Z is moved to the next page. Data in PEs 0 through 399 all have a page offset of 0 and data in PEs 400 through 511 all have a page offset of 1. To move the first segments, the switch is set for a rotation of 0 = D%512, the A tap pointer is set to 0 and the B tap pointer is set to 112 = 512-D%512. One bit from the source vector is loaded into the A register and then the switch is enabled for 400 cycles. The switch is disabled and the B register is copied into the result vector. All bits of the source vector are copied with a switch offset of 0 by this process. To move the second segments, the switch is set to have a rotation of 1 = D%512+1, the A tap pointer is set to 400 and the B tap pointer is set to 0. One bit of the source vector is loaded into the A register and the switch is enabled for 112 cycles. the switch is then disabled and the B register is conditionally copied into the result vector. The conditional mask marks those bits which correspond to segments B, D, L, N, X, and Z and was computed from the PE id numbers, which are stored as constants in each PE. All bits of the source vector are copied with a switch offset of 1 by this process. At this point, the source vector has been shifted by distance D. The time required to perform this operation is nbits*(512 PEs/35 megahertz) = nbits * 14.6 microseconds, where nbits is the number of bits in each vector element. This operation uses the switch efficiently, in that data is moved through the switch only once to reach its final destination. The time used by this algorithm in setting up the switch is roughly the same as the time to shift a one bit vector. Therefore, the switch overhead component becomes small as the number of bits to be shifted grows.

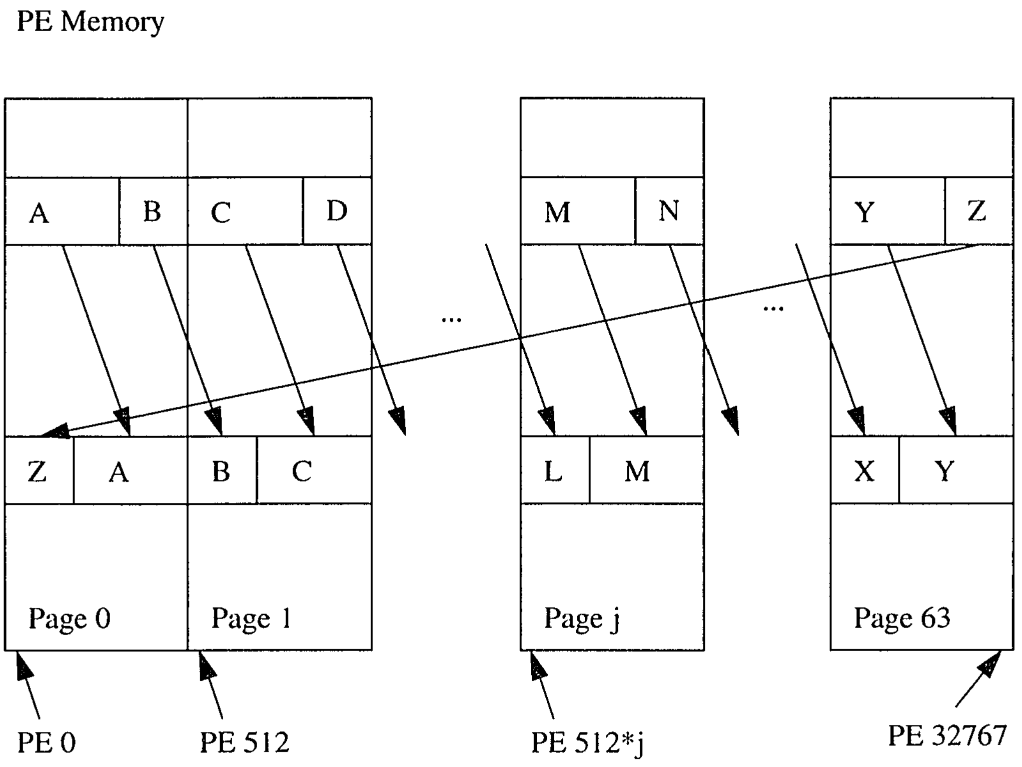


Figure 4.10 Align Operation

4.4 Vector Instruction Set

This section describes the vector instruction set. These instructions are sent by the host computer to the controller and placed in an instruction queue. They are fetched out of the queue and executed by the controller as fast as possible. All of these instructions assume that vectors are kept in PE memory in the format described earlier in this section. All of these instructions handle vectors with arbitrary element length and bit length.

Three pieces of information must be specified to describe a vector, its location in PE memory, the number of bits in each word, and the number of words per PE. The number of words per PE indicates how many times a vector longer than 32768 has wrapped around the machine. If the number of words is one, then the vector is 32768 elements long.

In general, binary operators, such as +, will use a three address format, where two source addresses are given and one destination. Most unary operators, such as ~, use a two address format, where separate source and destination addresses are given. In general, only PE memory at the destination address is altered and the operands to an operator are not changed. In general, the destination and operand addresses may not be the same and may not overlap. Unless otherwise specified, arithmetic operations are performed on all elements of a vector and binary operators are applied to corresponding elements of two vectors.

The controller makes no effort to keep track of which vectors are at what addresses, of how many bits a vector may have, or how long a vector may be. All of this information is maintained by the host and is encoded in each instruction as it is sent to the controller.

Three constants are written by the host to PE memory when the MM32k is booted. The one bit constant 0 is written at location 0 in PE memory, the one bit constant 1 is written at location 1 in PE memory, and the 15 bit PE index number, between 0 and 32767, is written into locations 2 through 16 of PE memory. These constants are used by some of the instructions in the vector instruction set. There are also two bits of PE memory assigned to be scratch locations.

All of these vector instructions may be issued to the controller by calling a C language macro. The calling sequences of these macros are given as the first line of each of the following descriptions. The types of the arguments are denoted using C language function prototyping syntax. PE memory addresses, bit counts, and word counts are 16 bit integer values. PE indexes and PE data values are 32 bit integers. Return values are 32 bit integers and are passed by reference.

4.4.1 Read a Word from a Vector

mm_simd_read(short address, short nbits, long index, long *value)

This instruction reads a single element of a vector and returns a 32 bit value. It operates by mapping PE memory onto the controller bus and reading PE memory a bit at a time. It requires four controller cycles per bit read.

```
result word = 0
loop over bits {
        map PE memory onto controller bus
        read the word from PE memory containing the PE bit
        add bit to result word
}
```

4.4.2 Write a Word to a Vector

mm_simd_write(short address, short nbits, long index, long value)

This instruction writes a 32 bit scalar value into a single element of the vector. It operates by mapping PE memory onto the controller bus and setting the bits corresponding to the element of the vector to be written. It must read a word corresponding to the current values of bits in 16 PEs, insert the bit corresponding to the PE to be written, and write the 16 bit word back to PE memory for each of the bits in the vector element. This instruction takes seven controller cycles per bit of the vector element written.

```
loop over bits {
        map PE memory onto controller bus
        read word from PE memory containing bit to set
        insert bit into word
        write word back to PE memory
}
```

4.4.3 Write a Constant to a Vector

mm_simd_write_const(short address, short nbits, short nwords, long value)

This instruction writes the same value into all elements of a vector. The value may be up to 32 bits long. This instruction takes one PE instruction per bit written. Pseudocode is given below.

```
M = 0
loop over words {
        loop over bits {
                if (constant bit is 1)
                        mem(to) = ~M
                else
                        mem(to) = M
        }
}
```

4.4.4 Copy a Vector

mm_simd_copy(short source_address, short source_nbits, short result_address, short result_nbits, short nwords)

This instruction copies a vector to another vector. The source and destination vectors do not have to have the same number of bits. If the destination has fewer bits than the source, then the source will be truncated. If the destination has more bits, then the source will be sign extended. This instruction requires two PE instructions per bit copied. Pseudocode is given below.

```
loop over words {
    loop over bits {
        M = mem(source)
        mem(result) = M
    }
}
```

4.4.5 Arithmetic Shift of a Vector

mm_simd_ashift(short source_address, short source_nbits, short result_address, short result_nbits, short nwords)

This instruction arithmetically shifts the bits of a vector up or down. The direction and length of the shift is determined by the relative bit lengths of the source and destination vectors. The sign bits are always kept the same and the low order bits will be either truncated or zero extended depending upon the direction of the shift. This instruction takes two PE instructions per bit shifted. Pseudocode is given below.

```
if (source bit length shorter than destination) {
    loop over words {
        M = 0
        loop over low order extension bits {
            mem(result) = M
        }
        loop over bits {
            M = mem(source)
            mem(result) = M
```

```
                }
        }
}
else {
        loop over words {
                loop over bits {
                        M = mem(source)
                        mem(result) = M
                }
        }
}
```

4.4.6 Bitwise NOT of a Vector

mm_simd_bitwise_not(short source_address, short source_nbits, short result_address, short result_nbits, short nwords)

This instruction complements the bits of the source vector. It sign extends or truncates the result to fit into the destination vector. This instruction takes two PE instructions per result bit. Pseudocode for the instruction and is given below.

```
loop over words {
        loop over bits {
                M = mem(source)
                mem(result) = ~M
        }
}
```

4.4.7 Bitwise OR of a Vector and a Vector

mm_simd_bitwise_xory(short x_address, short x_nbits, short y_address, short

y_nbits, short result_address, short result_nbits, short nwords)

This instruction performs a bitwise OR between the words of x and y vectors. It sign extends the shorter of the two operands to match the size of the longer and sign extends or truncates the result to fit the destination. It takes four PE instructions per result bit. Its operation is described below in pseudocode.

```
loop over words {
        loop over bits {
                A = mem(x)
                M = mem(y
                 if M then A = mem(y)
                mem(result) = A
        }
}
```

4.4.8 Bitwise AND of a Vector and a Vector

mm_simd_xandy(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short result_nbits, short nwords)

This instruction is similar to the BITWISE OR instruction except it performs the AND operation. It takes four PE instructions per result bit. The algorithm is given below in pseudocode.

```
loop over words {
        loop over bits {
                A = mem(x)
                M = ~mem(y)
                if M then A = mem(y)
                mem(result) = A
        }
}
```

4.4.9 Bitwise XOR of a Vector and a Vector

mm_simd_xxory(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short result_nbits, short nwords)

This instruction is similar to the BITWISE OR instruction except that it performs the XOR operation and takes five PE instructions per result bit. The algorithm is given below in pseudocode.

```
loop over words {
        loop over bits {
                M = ~mem(x)
                A = M
                M = ~mem(y)
                if M then A = mem(x)
                mem(result) = A
        }
}
```

4.4.10 Bitwise OR of a Vector with a Scalar

mm_simd_xork(short x_address, short x_nbits, long scalar, short result_address, short result_nbits, short nwords)

This instruction performs the bitwise OR operation between the bits of a vector and the bits of a scalar. The scalar is a 32 bit quantity. If the operand vector has fewer bits than the result vector, then it is sign extended. The result of the OR operation is either sign extended or truncated so that it will fit into the result vector. The algorithm takes two PE instructions per result bit and is shown below.

```
loop over words {
        loop over bits {
                if (scalar bit is 1)
                        M = 0
                else
                        M = ~mem(x)
                mem(result) = ~M
        }
}
```

4.4.11 Bitwise AND of a Vector with a Scalar

mm_simd_xandk(short x_address, short x_nbits, long scalar, short result_address, short result_nbits, short nwords)

This instruction is similar to the BITWISE OR WITH SCALAR instruction except that it performs the AND operation. The algorithm takes two PE instructions per result bit and is given below.

```
loop over words {
        loop over bits {
                if (scalar bit is 1)
                        M = 0
                else
                        M = mem(x)
                mem(result) = M
        }
}
```

4.4.12 Bitwise XOR of a Vector with a Scalar

mm_simd_xxork(short x_address, short x_nbits, long scalar, short result_address, short result_nbits, short nwords)

This instruction is similar to the BITWISE XOR WITH SCALAR instruction except that it performs the XOR operation. The algorithm takes two PE instructions per result bit and is given below.

```
loop over words {
        loop over bits {
                if (scalar bit is 1)
                        M = ~mem(from)
                else
                        M = mem(from)
                mem(result) = M
        }
}
```

4.4.13 Logical NOT of a Vector

mm_simd_logical_not(short source_address, short source_nbits, short result_address, short result_nbits, short nwords)

This instruction performs the logical NOT operation on each of the words in a vector and returns a one bit vector with the result. If any bit in a word is set, then the result bit is clear, else the result bit is set. The algorithm takes two PE instructions per result bit and is given below.

```
M = 0
A = M
loop over words {
        M = ~mem(source)
        mem(result) = M
        loop over bits {
                M = mem(source)
                if M then mem(result) = A
        }
}
```

4.4.14 Negate a Vector

mm_simd_negate(short source_address, short source_nbits, short result_address, short result_nbits, short nwords)

This instruction negates the integers in a vector. The result vector has one more bit than the source vector. The algorithm takes five PE instructions and is given below.

```
loop over words {
        B = mem(constant_one)
        loop over bits {
                M = B
                mem(result) = ~M
                M = mem(source)
                if M then mem(result) = B
                if M then B = mem(constant_zero)
        }
        M = B
        mem(result_sign_bit) = ~M
        M = mem(from_sign_bit)
        if M then mem(result_sign_bit) = B
```

}

4.4.15 Absolute Value of a Vector

mm_simd_abs(short source_address, short source_nbits, short result_address, short result_nbits, short nwords)

This instruction takes the absolute values of the integers of a vector. It produces a result vector which is one bit longer than the source vector. The algorithms takes seven PE instructions per result bit and is given below.

```
loop over words {
    negate source and store in destination
    M = mem(destination sign bit)
    loop over bits {
        A = mem(source)
        if M mem(result) = A
    }
}
```

4.4.16 Add a Vector to a Vector

mm_simd_xply(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short result_nbits, short nwords)

This instruction adds the words of the x vector to the words of the y vector and puts the result in the destination vector. It sign extends the shorter of the two operands to match the size of the longer and sign extends or truncates the result to fit the destination. The algorithm takes nine PE instructions per result bit and is given below.

```
loop over words {
    B = mem(constant_zero)
    loop over bits {
        M = ~mem(x)
        A = M
        M = mem(y)
        if M then A = mem(x)
        M = B
        mem(result) = ~M
        M = A
        if M then mem(result) = B
        if M then B = mem(x)
    }
}
```

4.4.17 Subtract a Vector from a Vector

mm_simd_xmiy(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short result_nbits, short nwords)

This instruction subtracts the words of the the y vector from the words of the x vector and puts the result in the destination vector. It sign extends the shorter of the two operands to match the size of the longer and sign extends or truncates the result to fit the destination. The algorithm takes nine PE instructions per result bit and is given below.

```
loop over words {
    B = mem(constant_one)
    loop over bits {
        M = ~mem(x)
        A = M
        M = ~mem(y)
        if M then A = mem(x)
        M = B
        mem(result) = ~M
        M = A
        if M then mem(result) = B
        if M then B = mem(x)
    }
}
```

4.4.18 Add a Scalar to a Vector

mm_simd_xplk(short x_address, short x_nbits, long scalar, short result_address, short result_nbits, short nwords)

This instruction adds a scalar to the words of a vector. The scalar is a 32 bit quantity. If the number of bits in the source vector is less than the number of bits in the result, then it is sign extended. The algorithm takes five PE instructions per result bit and is given below.

```
loop over words {
    B = mem(constant_zero)
    loop over bits {
        M = B
        mem(result) = ~M
        if (scalar bit is one)
            M = mem(x)
        else
            M = ~mem(x)
        if M then mem(result) = B
        if M then B = mem(x)
        }
}
```

4.4.19 Subtract a Scalar from a Vector

mm_simd_xmik(short x_address, short x_nbits, long scalar, short result_address, short result_nbits, short nwords)

This instruction subtracts a scalar from a vector. The scalar is a 32 bit quantity. If the number of bits in the source vector is less than the number of bits in the result, then the source is sign extended. The algorithm takes five PE instructions per result bit and is given below.

```
loop over words {
        B = mem(constant_one)
        loop over bits {
                M = B
                mem(result) = ~M
                if (scalar bit is one)
                        M = ~mem(x)
                else
                        M = mem(x)
                if M then mem(result) = B
                if M then B = mem(x)
        }
}
```

4.4.20 Subtract a Vector from a Scalar

mm_simd_kmix(long scalar, short x_address, short x_nbits, short result_address,

short result_nbits, short nwords)

This instruction subtracts a vector from a scalar. The scalar is a 32 bit quantity. If the number of bits in the source vector is less than the number of bits in the result, then the source vector is sign extended. The algorithm takes five PE instructions per result bit and is given below.

```
loop over words {
    B = mem(constant_one)
    loop over bits {
        B = M
        mem(result) = ~M
        if (scalar bit is one) {
            M = mem(x)
            if M then mem(result) = B
            if M then B = mem(constant_zero)
        }
        else {
            M = ~mem(x)
            if M then mem(result) = B
            if M then B = mem(constant_one)
        }
    }
}
```

4.4.21 Multiply a Vector by a Vector

mm_simd_xmpy(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short result_nbits, short nwords)

This instruction multiplies two vectors together and puts the result in a result vector. The bit length of the result vector is equal to the sum of the bit lengths of the source vectors. This instruction takes 13 PE instructions per partial product and there are x_nbits*y_nbits partial products. Pseudocode for the instruction is given below. It uses two scratch PE memory locations, called carry and pp.

```
loop over words {
    set result bits to zero
    loop over y bits {
        M = 0
        mem(carry) = M
        loop over x bits {
            M = mem(x)
            A = mem(constant_zero)
            if M then A = mem(y)
            M = A
            mem(pp) = ~M
            M = mem(result)
            if M then mem(pp) = A
            M = ~mem(carry)
            mem(result) = M
            B = mem(carry)
            M = mem(pp)
            if M then mem(result) = B
            if M then mem(carry) = A
        }
        }
}
```

4.4.22 Multiply a Vector by a Scalar

mm_simd_xmpk(short x_address, short x_nbits, long scalar, short result_address, short result_nbits, short nwords)

This instruction multiplies a vector by a scalar. The scalar is a 32 bit quantity. It uses as many of the scalar bits as are required to produce the specified number of result bits. It takes 13 PE instructions per partial product and does not compute those partial products corresponding to a 0 bit in the scalar. Since, on average, half the bits in a scalar tend to be zero and half one, this instruction tends to take half as many PE instructions as the MULTIPLY VECTOR BY VECTOR instruction of the same bit precision. Pseudocode for the instruction is given below. It uses two scratch PE memory locations, called carry and pp.

```
loop over words {
        set result bits to zero
        loop over scalar bits {
                if (scalar bit is 1) {
                        M = 0
                        mem(carry) = M
                        loop over x bits {
                                M = ~mem(carry)
                                A = M
                                M = ~mem(x)
                                if M then A = mem(carry)
                                M = ~mem(result)
                                B = M
```

```
                    M = A
                    A = mem(result)
                    if M then mem(result) = B
                    if M then mem(carry) = A
                }
            }
        }
}
```

4.4.23 Subroutine to Divide

This section describes the divide subroutine, which is not a vector instruction, but is used by the divide and modulo family of instructions. The subroutine computes a one word quotient vector and a one word remainder vector from a one word divisor vector and a one word dividend vector, using repeated subtraction and unsigned arithmetic. Four scratch vectors are used, called quotient, remainder, q, and temp. The quotient vector accumulates the bits of the quotient. The remainder vector initially holds the dividend, then holds the successive differences as the division proceeds, and finally holds the remainder. The q vector holds the unsigned divisor and the temp vector temporarily holds the results of a subtracting the q from remainder. These vectors are located in PE memory in scratch space temporarily allocated by the host for this instruction. This subroutine takes approximately (divisor_nbits * dividend_nbits * 11) PE instructions and is described by pseudocode below.

```
remainder should be set to the unsigned dividend by the callerl;
q should be set to the unsigned divisor by the caller;
loop over the the number of bits in the dividend {
    temp = remainder - q
```

```
        if (temp < 0) {
                remainder = temp;
                set quotient bit = 0
        }
        else
                set quotient bit = 1
        shift remainder left by one bit
}
```

4.4.24 Divide a Vector by a Vector

mm_simd_xdvy(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short result_nbits, short nwords, short scratch_address)

This instruction divides a vector by a vector. It produces a quotient which is one bit longer than the dividend and uses the divide subroutine. Pseudocode for this instruction is given below.

```
loop over words {
        remainder = abs(x)
        q = abs(y)
        call divide subroutine
        result = quotient
        if (result sign bit should be negative)
                result = - quotient
}
```

4.4.25 Divide a Vector by a Scalar

mm_simd_xdvk(short x_address, short x_nbits, long scalar, short result_address, short result_nbits, short nwords, short scratch_address)

This instruction divides a vector by a scalar. It produces a quotient which is one bit longer than the dividend and uses the divide subroutine. Pseudocode for this instruction is given below.

```
loop over words {
    remainder = abs(from)
    q = abs(scalar)
    call divide subroutine
    result = quotient
    if (result sign bit should be negative)
        result = -quotient
}
```

4.4.26 Divide a Scalar by a Vector

mm_simd_kdvx(long scalar, short x_address, short x_nbits, short result_address, short nwords, short scratch_address)

This instruction divides a scalar by a vector. It produces a quotient which is one bit longer than the dividend and uses the divide subroutine. Pseudocode for this instruction is given below.

```
loop over words {
    remainder = abs(scalar)
    q = abs(x)
    call divide subroutine
    result = quotient
    if (result sign bit should be negative)
        result = -quotient
}
```

4.4.27 Modulo a Vector by a Vector

mm_simd_xmdy(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short nwords, short scratch_address)

This instruction divides a vector by a vector and returns the remainder. It produces a result which is one bit longer than the dividend and uses the divide subroutine. Pseudocode for this instruction is given below.

```
loop over words {
    remainder = abs(x)
    q = abs(y)
    call divide subroutine
    result = remainder
    if (result sign bit should be negative)
        result = - remainder
}
```

4.4.28 Modulo a Vector by a Scalar

mm_simd_xmdk(short x_address, short x_nbits, long scalar, short result_address, short nwords, short scratch_address)

This instruction divides a vector by a scalar and returns the remainder. It produces a result which is one bit longer than the dividend and uses the divide subroutine. Pseudocode for this instruction is given below.

```
loop over words {
    remainder = abs(x)
    q = abs(scalar)
    call divide subroutine
    result = remainder
    if (result sign bit should be negative)
        result = -remainder
}
```

4.4.29 Modulo a Scalar by a Vector

mm_simd_kmdx(long scalar, short x_address, short x_nbits, short result_address, short nwords, short scratch_address)

This instruction divides a scalar by a vector and returns the remainder. It produces a result which is one bit longer than the dividend and uses the divide subroutine. Pseudocode for this instruction is given below.

```
loop over words {
    remainder = abs(scalar)
    q = abs(x)
    call divide subroutine
    result = remainder
    if (result sign bit should be negative)
        result = -remainder
}
```

4.4.30 Compare if a Vector is EQUAL to a Vector

mm_simd_xeqy(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short nwords)

This instruction compares two vectors on an element by element basis and returns a one bit vector with a bit set if they are equal. If the bit lengths of the two vectors are not equal, then the shorter is sign extended to the length of the longer. The algorithm takes six PE instructions per source bit and is given below.

```
loop over words {
        B = mem(constant_one)
        loop over bits {
                M = ~mem(x)
                A = M
                M = ~mem(y)
                if M then A = mem(x)
                M = A
                if M then B = mem(constant_zero)
        }
        mem(result) = B
}
```

4.4.31 Compare if a Vector is NOT EQUAL to a Vector

mm_simd_xney(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short nwords)

This instruction compares two vectors on an element by element basis and returns a one bit vector with a bit set if they are not equal. If the bit lengths of the two vectors are not equal, then the shorter is sign extended to the length of the longer. The algorithm takes six PE instructions per source bit and is given below.

```
loop over words {
        B = mem(constant_zero)
        loop over bits {
                M = ~mem(x)
                A = M
                M = ~mem(y)
                if M then A = mem(x)
        M = A
        if M then B = mem(constant_one)
    }
    mem(result) = B
}
```

4.4.32 Compare if a Vector is GREATER OR EQUAL to a Vector

mm_simd_xgey(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short nwords)

This instruction compares two vectors on an element by element basis and returns a one bit vector with a bit set if the first is greater than or equal to the second. If the bit lengths of the two vectors are not equal, then the shorter is sign extended to the length of the longer. The algorithm takes six PE instructions per source bit and is given below.

```
loop over words {
    B = mem(constant_one)
    loop over bits {
        M = ~mem(x)
        A = M
        M = ~mem(y)
        if M then A = mem(x)
        M = A
        if M then B = mem(x)
    }
    mem(result) = B
}
```

4.4.33 Compare if a Vector is GREATER than a Vector

mm_simd_xgty(short x_address, short x_nbits, short y_address, short y_nbits, short result_address, short nwords)

This instruction compares two vectors on an element by element basis and returns a one bit vector with a bit set if the first is greater than the second. If the bit lengths of the two vectors are not equal, then the shorter is sign extended to the length of the longer. The algorithm takes six PE instructions per source bit and is given below.

```
loop over words {
        B = mem(constant_zero)
        loop over bits {
                M = ~mem(x)
                A = M
                M = ~mem(y)
                if M then A = mem(x)
                M = A
                if M then B = mem(x)
        }
        mem(result) = B
}
```

4.4.34 Compare if a Vector is EQUAL to a Scalar

mm_simd_xeqk(short x_address, short x_nbits, long scalar, short result_address, short nwords)

This instruction compares a vector with a scalar and produces a one bit vector with a bit set if the vector is equal to the scalar. This instruction sign extends the vector to match the number of bits in the scalar. The algorithm takes two PE instructions per source bit and is given below.

```
loop over words {
    A = mem(constant_one)
    loop over bits {
        if (scalar bit is set)
            M = ~mem(x)
        else
            M = mem(x)
        if M then A = mem(constant_zero)
    }
    mem(result) = A
}
```

4.4.35 Compare if a Vector is NOT EQUAL to a Scalar

mm_simd_xnek(short x_address, short x_nbits, long scalar, short result_address, short nwords)

This instruction compares a vector with a scalar and produces a one bit vector with a bit set if the vector is not equal to the scalar. This instruction sign extends the vector to match the number of bits in the scalar. The algorithm takes two PE instructions per source bit and is given below.

```
loop over words {
    A = mem(constant_zero)
    loop over bits {
        if (scalar bit is set)
            M = ~mem(x)
        else
```

```
                M = mem(x)
            if M then A = mem(constant_one)
        }
        mem(result) = A
}
```

4.4.36 Compare if a Vector is GREATER than a Scalar

mm_simd_xgtk(short x_address, short x_nbits, long scalar, short result_address, short nwords)

This instruction compares a vector with a scalar and produces a one bit vector with a bit set if the vector is greater than the scalar. This instruction sign extends the vector to match the number of bits in the scalar. The algorithm takes two PE instructions per source bit and is given below.

```
loop over words {
        A = mem(constant_zero)
        loop over bits-1 {
            if (scalar bit is set)
                M = ~mem(x)
            else
                M = mem(x)
            if M then A = mem(x)
        }
        M = ~mem(x)
        mem(scratch) = M
        if (scalar bit set)
            M = ~mem(x)
        else
                M = mem(x)
        if M then A = mem(scratch)
        mem(result) = A
}
```

4.4.37 Compare if a Vector is GREATER OR EQUAL to a Scalar

mm_simd_xgek(short x_address, short x_nbits, long scalar, short result_address, short nwords)

This instruction compares a vector with a scalar and produces a one bit vector with a bit set if the vector is greater than or equal to the scalar. This instruction sign extends the vector to match the number of bits in the scalar. The algorithm takes two PE instructions per source bit and is given below.

```
loop over words {
        A = mem(constant_one)
        loop over bits-1 {
                if (scalar bit is set)
                        M = ~mem(x)
                else
                        M = mem(x)
                if M then A = mem(x)
        }
        M = ~mem(x)
        mem(scratch) = M
        if (scalar bit set)
                M = ~mem(x)
        else
                M = mem(x)
        if M then A = mem(scratch)
        mem(result) = A
}
```

4.4.38 Compare if a Vector is LESS than a Scalar

mm_simd_xltk(short x_address, short x_nbits, long scalar, short result_address, short nwords)

This instruction compares a vector with a scalar and produces a one bit vector with a bit set if the vector is less than the scalar. This instruction sign extends the vector to match the number of bits in the scalar. The algorithm takes two PE instructions per source bit and is given below.

```
loop over words {
        A = mem(constant_one)
        loop over bits-1 {
                if (scalar bit is set)
                        M = ~mem(x)
                else
                        M = mem(x)
                if M then A = mem(x)
        }
        M = ~mem(x)
        mem(scratch) = M
        if (scalar bit set)
                M = ~mem(x)
```

```
    else
        M = mem(x)
    if M then A = mem(scratch)
    M = A
    mem(result) = ~M
}
```

4.4.39 Compare if a Vector is LESS OR EQUAL to a Scalar

mm_simd_xlek(short x_address, short x_nbits, long scalar, short result_address, short nwords)

This instruction compares a vector with a scalar and produces a one bit vector with a bit set if the vector is less than or equal to the scalar. This instruction sign extends the vector to match the number of bits in the scalar. The algorithm takes two PE instructions per source bit and is given below.

```
loop over words {
    A = mem(constant_zero)
    loop over bits-1 {
        if (scalar bit is set)
            M = ~mem(x)
        else
            M = mem(x)
        if M then A = mem(x)
    }
    M = ~mem(x)
    mem(scratch) = M
    if (scalar bit set)
```

```
        M = ~mem(x)
    else
        M = mem(x)
    if M then A = mem(scratch)
    M = A
    mem(result) = ~M
}
```

4.4.40 Select Elements from Two Vectors

mm_simd_select(short test_address, short test_nbits, short true_address, short true_nbits, short false_address, short false_nbits, short result_address, short result_-nbits, short nwords)

This instruction tests a vector and if any bit in it is set, it returns the corresponding element of the 'true' vector else it returns the corresponding element of the 'false' vector. The algorithm takes three PE instructions per result bit plus two PE instructions per test bit and is given below.

```
loop over words {
    A = mem(test)
    loop over test bits {
        M = mem(test)
        if M then A = mem(test)
    }
    M = A
    loop over result bits {
        A = mem(false)
        if M then A = mem(true)
```

```
                mem(result) = A
        }
}
```

4.4.41 Align a Vector

mm_simd_align(short source_address, short source_nbits, short result_address, short nwords, long distance)

This instruction does a circular shift across the elements of a vector by the specified distance. After an align instruction, the data which was in element i is in element (i+D)%L, where D is the distance of the shift and L is the length of the vector. Data which is shifted off the end of the source vector is shifted into the other end of the result vector. Shift distances can be positive or negative. The alignment is divided into two parts, an alignment across PEs and an alignment through the words of a PE. First all of the words are aligned so that they are in the correct destination PE, but not necessarily the correct word in that PE. This is done by rotating the words by D PEs using the switch. Then the words are moved to the correct word within each PE. This is done by rotating all words in PEs 0 <= id < D%32768 by D/32768+1 words vertically within the PE and by rotating all data in PEs D%32768 <= id < 32768 by D/32768 words vertically within the PE. At this point, the original vector has been aligned in an end around fashion. The operation moves data at a 64*35/8 = 280 megabyte per second rate. Pseudocode is given below.

```
loop over words per PE {
        align word in each PEs by D%32768
}
loop over words per PE {
        if (0<= id < D%32768) {
                rotate words in PE through PE by D/32768+1
        {
        else {
                rotate words in PE through PE by D/32768
        }
}
```

4.4.42 Subroutine to Test All Processors

This algorithm tests the A registers of all PEs and returns a flag if any bit in any PE is set. This is not an instruction in the vector instruction set, but is described here because it is used to implement vector instructions. The algorithm works in two steps. First, the bits in the A register on each page are ORed together using a tree. This is done by using the switch to align bits 256 through 511 with bits 0 through 255 on each page. The 256 bit long aligned segments are ORed together. This process is repeated with 128, 64, 32, 16, 8, 4, 2, and 1 bit long segments. At this point, the bitwise OR of all bits originally in the A register is located in the A register of the first PE on each page. In the second step, these PEs on all 64 pages are tested to determine if any bit was set in the A register over all 32768 PEs. The algorithm takes about 30 microseconds to execute. Pseudocode for this instruction is given below.

```
put value to test in the A register of each PE
set switch mapping to identity
loop for i = (256 128 64 32 16 8 4 2 1) {
        set tap pointer A to i
        set tap pointer B to 0
        enable switch for i cycles
        mem(scratch) = B
        M = B
        if M then A = mem(scratch)
}
mem(scratch) = A
map PE memory onto controller bus
test scratch bit of first PE of each page
if (any bit is set)
        return flag true
else
        return flag false
```

4.4.43 Global OR

mm_global_or(short address, short nbits, short nwords, long *scalar_result)

This instruction returns the bitwise OR of all words in a vector. It computes results one bit at a time, by ORing together the bits of all words in a long vector and then calling a subroutine to test them. It then inserts the result of the test into the scalar value to be returned. This instruction returns a scalar value which has as many significant bits as the input vector and is sign extended to be 32 bits long. The algorithm takes about 30 microseconds per result bit and is described below.

```
loop over bits {
        A = mem(constant_zero)
        loop over words {
                M = mem(address)
                if M then A = mem(constant_one)
        }
        call subroutine to test all processors
        if (a bit is set)
        insert a one bit into the scalar result
    else
        insert a zero bit into the scalar result
}
return scalar result
```

4.4.44 Global AND

mm_global_and(short address, short nbits, short nwords, long *scalar_result)

This instruction returns the bitwise AND of all words in a vector. It computes results one bit at a time, by ANDing together the of the bits of all words in a long vector and then calling a subroutine to test them. It then inserts the complement of the result of the test into the scalar value to be returned. This instruction returns a scalar value which has as many significant bits as the input vector and is sign extended to be 32 bits long. The algorithm takes about 30 microseconds per result bit and is described below.

```
loop over bits {
    A = mem(constant_zero)
    loop over words {
        M = ~mem(address)
        if M then A = mem(constant_one)
    }
    call subroutine to test all processors
    if (a bit is set)
        insert zero bit into the scalar result
    else
        insert a one bit into the scalar result
}
```

```
return scalar result
```

4.4.45 Global MAX

mm_global_max(short address, short nbits, short nwords, short scratch_address, long *scalar_result)

This instruction returns the value of the maximum word of all words in a vector. It finds the maximum by doing a binary search of the range of values represented by the words of the vector. The binary search has as many steps as the number of bits in a word of the vector. It uses a one bit vector the same length as the input vector to mark whether a particular word in the vector is still a candidate to be the maximum. This vector is only allocated for the duration of the instruction. It starts with the most significant bit and tests if the bit in any marked word is one. If so, then the corresponding bit in the maximum value is a one and all candidate words with a zero are marked to no longer be a candidate. If not, then the corresponding bit of the maximum value is zero. This process continues for all bits in the input vector. When checking the first bit which is the sign bit, the sense of the bit test is reversed. The algorithm takes approximately nbits*(30 +nwords) microseconds to execute. Pseudocode is given below.

```
set all bits in candidate vector
loop over bits {
    if (any bit of any candidate word is one) {
        clear candidate bit corresponding to candidate words with zero
        insert one bit into scalar result
    }
    else {
        insert zero bit into scalar result
    }
}
return scalar result
```

4.4.46 Global MIN

mm_global_min(short address, short nbits, short nwords, short scratch_address, long *scalar_result)

This instruction returns the value of the minimum word of all words in a vector. It finds the minimum by doing a binary search of the range of values represented by the words of the vector. The binary search has as many steps as the number of bits in a word of the vector. It uses a one bit vector the same length as the input vector to mark whether a particular word in the vector is still a candidate to be the minimum. This vector is only allocated for the duration of the instruction. It starts with the most significant bit and tests if the bit in any marked word is zero. If so, then the corresponding bit in the minimum value is a zero and all candidate words with a one are marked to no longer be a candidate. If not, then the corresponding bit of the minimum value is one. This process continues for all bits in the input vector. When checking the first bit which is the sign bit, the sense of the bit test is reversed. The algorithm takes approximately nbits*(30 +nwords) microseconds to execute. Pseudocode is given below.

```
set all bits in candidate vector
loop over bits {
    if (any bit of any candidate word is zero) {
        clear candidate bit corresponding to candidate words with one
        insert zero bit into scalar result
    }
    else {
        insert one bit into scalar result
        }
}
return scalar result
```

4.4.47 Global SUM

mm_global_sum(short address, short nbits, short nwords, short scratch_address, long *scalar_result)

This instruction returns the sum of all words in a vector. It operates in three steps. First, partial sums are formed by adding all the words of the vector in each PE. These sums are formed in each of the 32768 PEs using a tree. Second, these partial sums are added up on a per page basis giving 64 new partial sums, each located in the first PE of a page. The switch aligns partial sums 9 times. Third, the PE memory is mapped onto the controller bus and the 64 partial sums are added up serially by the controller. Scratch space must be allocated for the duration of this instruction in order to hold the partial sums. The algorithm takes approximately one millisecond for an 8 bit vector. Pseudocode is given below.

```
add all words in each PE to form 32768 partial sums
loop for i = (256 128 64 32 16 8 4 2 1) do {
        on each page and using the switch,
                align partial sums in PEs i through 2*i-1 with PEs 0 through i-1
        on each page,
                add the i*2 aligned partial sums
                to form i new partial sums in PEs 0 through i-1
}
map PE memory onto controller bus
add up 64 partial sums serially with the controller
return the scalar result to the host
```

4.4.48 First Nonzero Element in a Vector

mm_find_first(short address, short nbits, short nwords, long *index)

This instruction returns the vector index number of the first nonzero word in a vector. If all words in the vector are zero, it returns a -1. This instruction works in three steps. First, it checks if any bit in any word of the vector is set. If not, it returns a -1. Second, it divides the vector into subvectors 32768 element long, where each subvector is aligned with the others. It does a binary search across the subvectors to determine the first nonzero subvector. This second step determines the value of the index bits 15 and higher. In the third step, bits 0 through 14 of the index are determined. A scratch bit is used in each PE to mark the nonzero words of the first nonzero subvector. A binary search is then performed on the PEs to determine which one contains the first nonzero bit. The id number of the first nonzero PE supplies bits 0 through 14 of the index number. The algorithm takes approximately 30 microseconds per result bit. Pseudocode is given below.

```
check if all bits in vector are zero
if so, then return -1
/* do a binary search across rows to set bits above bit 14 */
loop for n = ( nwords/2, nwords/4, ... , 4, 2, 1 ) {
    check n words starting at address
    if ( any word is nonzero ) {
        set result bit to 0
    }
    else {
        set result bit to 1
        set address = address + n
    }
}
/* do a binary search across PEs to set bits 14 to 0 */
set a scratch bit in each PE to indicate if it contains nonzero words
loop for id = ( 14, 13, 12, ... , 3, 2, 1, 0 ) {
    set A register in each PE to scratch AND ~mem( PE_ID_BITS + id )
    call subroutine to test all processors
    if ( any A register bit is set ) { /* bottom half is nonzero */
        set result bit to 0
        set scratch = scratch AND ~mem( PE_ID_BITS + id )
    }
    else { /* top half is nonzero */
        set result bit to 1
        set scratch = scratch AND mem( PE_ID_BITS + id )
    }
}
return result
```

4.4.49 Create Vector with Element Indices

mm_simd_index(short address, short nwords)

This instruction returns a vector which contains a sequence of integers starting with 0, 1, 2, 3, and so on for all elements of the vector. It works by copying the 15 bit PE id numbers, which are constants stored in each PE, into bits 0 through 14 of all integers in the result vector. It then writes the PE word numbers into the bits 15 and higher in the case of vectors longer than 32768. Lastly, it adds a zero to make all the integers positive. The algorithm takes two PE instructions per result bit and is described below.

```
copy bits 0 to 14 of PE id numbers into lower 15 bits of result vector
```

```
write i/32768 into bits 15 and higher of result vector
write 0 into sign bit of result vector
```

Chapter 5

C++ PROGRAMMING ENVIRONMENT

The purpose of this chapter is to describe the C++ programming environment in which the MM32k is programmed. The logical machine described in the last chapter is abstracted by a C++ class called MM_VECTOR, which represents a vector of integers. The MM32k is programmed by writing an ordinary C++ program which manipulates MM_VECTOR variables using overloaded C++ arithmetic operators and functions. The storage and computation associated with these variables is allocated and performed on the MM32k. This program is compiled, linked, run, and debugged on the host using an ordinary C++ compiler and ordinary programming tools.

In the following sections, a description is given of the C++ language class mechanism, the MM_VECTOR class, the PE memory management, and the overloaded operators and functions. A short programming example is then given as a concrete example.

5.1 C++ Class Background

The C++ language is a superset of the C language and it provides a <u>class</u> mechanism which is a superset of the <u>typedef</u> mechanism in the C language. The class mechanism allows a programmer to extend the C++ language to implement variables representing abstract objects and allows the programmer to manipulate these new variables in a manner similar to traditional C language variables. This section provides a little background on the class mechanism of the C++ language. A variable whose type is a class is known as an <u>instance</u> of that class. The package of declarations and functions which implement the class is known as the <u>class library</u>.

One way in which a class is different than a typedef is that <u>constructor</u> and <u>destruc</u>-

tor functions can be defined for the class. The constructor function is called when an instance variable goes into scope and is used to initialize the variable. The destructor function is called when the instance variable goes out of scope and is used to terminate the variable. The important point about both the constructor and destructor is that the class programmer can define pieces of code which set up and take down the object.

Another feature in the C++ language and not in the C language is that of operator overloading. Operators such as + and * can be overloaded to work with variables of a newly defined class. The programmer can supply a function which can add two variables of a class and return the result. A program may then be written which uses the + operator to add two variables of the class together and the supplied function will be called. In this manner, arithmetic can be done on variables of the new class using the + and other operators.

Functions may also be overloaded in the C++ language. More than one version of a function may be supplied, each having different types of variables passed as input arguments. When the function is used in application code, the C++ compiler matches the argument types and calls the appropriate version of the function.

5.2 MM_VECTOR Class

The MM_VECTOR class represents a variable length array of variable precision integers. It allows MM_VECTOR variables to be declared and manipulated as if they were variables on the host computer, but the storage associated with them is in PE memory and computation is performed by the PEs. This class is implemented by a C++ class library which runs on the host and sends vector instructions to the logical machine. The data elements of the array are stored in the PE memory of the MM32k and are manipulated by the PEs at full speed.

The number of bits which represent the integers of an MM_VECTOR is variable. All elements of a particular MM_VECTOR have the same number of bits per element, but different MM_VECTORs may have different numbers of bits. The strategy is to keep enough bits in the MM_VECTOR to represent both the largest positive and the largest negative numbers without overflow. Whenever an arithmetic operation is performed between two MM_VECTORs, enough bits are allocated to represent the result without overflow. For some operations, addition for example, the number of bits required to represent the result cannot be computed from the bit lengths of the MM_VECTOR arguments, so enough bits are allocated to handle the worst case. The shortest bit length for an MM_VECTOR is one bit, which can represent the values 0 and -1. One bit vectors are frequently used to represent the boolean result of a test. The largest bit length of an MM_VECTOR is limited by the available storage in PE memory. The elements of an MM_VECTOR can be peeked and poked as 32 bit quantities. The class library dynamically computes the number of bits required to actually represent a 32 bit integer which is poked and resizes the MM_VECTOR to be larger if need be to contain the integer. The class library sign extends a peeked value if it is shorter than 32 bits. The size of PE memory is the absolute limit to the number of bits in an MM_VECTOR, although there is no mechanism to directly read or write elements larger than 32 bits.

The number of elements in an MM_VECTOR is also variable. The length is rounded up to a multiple of 32768. If a shorter vector is desired, the extra elements can be considered to be undefined. The elements of an MM_VECTOR are numbered starting with 0 up to the length-1.

5.3 PE Memory Heap

This section describes the PE memory heap, in which vectors are stored. The PE memory is divided into segments and is managed as a heap. Each segment is a contiguous block of bits and all PEs have the same set of segments at the same PE addresses. Each segment contains either free PE memory heap space or bit storage for an MM_VECTOR. If the MM_VECTOR is longer than 32768, then it has more than one word per PE and these words are allocated contiguously in a single segment. The segments are represented by C language structures and are kept in a linked list which is sorted according to the PE address of the segment. A definition of this structure is listed below.

```
struct mmvector {
        short addr;
        short nwords;
        short nbits;
        short refcount;
        struct mmvector *next;
};
```

The first member of the structure is a 16 bit integer and holds the PE address of the start of the segment. The next member of the structure is a 16 bit integer and contains the number of MM_VECTOR words per PE. The next member of the structure is a 16 bit integer and contains the number of bits per word of the MM_VECTOR. The next member of the structure is a 16 bit integer and counts the number of MM_VECTOR variables that refer to this segment. If the number is zero, then the segment is free space. The last member is a pointer to the next structure in the linked list of segments. All 512 bits of PE memory are represented by a segment in the segment list, including the memory allocated to constants.

5.4 MM_VECTOR Implementation

This section describes the data structure associated with a MM_VECTOR variable. A portion of the MM_VECTOR class declaration is given below.

```
class MM_VECTOR {
        // ... other parts of the declaration
        struct mmvector {
            short addr;
            short nwords;
            short nbits;
            short refcount;
            struct mmvector *next;
        };
        mmvector *mmv;
        // other parts of the declaration ...
};
```

The MM_VECTOR variable contains a pointer to a structure for a segment. It contains no other data members. More than one MM_VECTOR variable can point at a single segment. The reference count member of the structure counts how many MM_VECTOR variables refer to the segment. If no MM_VECTOR variables point to a particular segment, then its reference count will be zero and it is considered free space in the PE memory heap. The segment list is shown in the next figure.

A benefit of this system is that arguments can be passed to a function by value and results returned without allocating duplicate storage in PE memory and copying any data. The drawback is the overhead of maintaining the reference counter. Another advantage is that free PE memory can be located quickly by searching for zero reference counters.

When an MM_VECTOR variable goes into scope, the constructor function for the MM_VECTOR class is called. Unless given an initial value, all MM_VECTOR variables are created one bit long and initialized to zero. PE memory space for the variable is allocated as a segment in PE memory and a structure is set up to represent the segment. When an operator is used in a program, an MM_VECTOR variable is allocated to hold the result of the operation. Space for this variable is allocated as a segment in the list of segments.

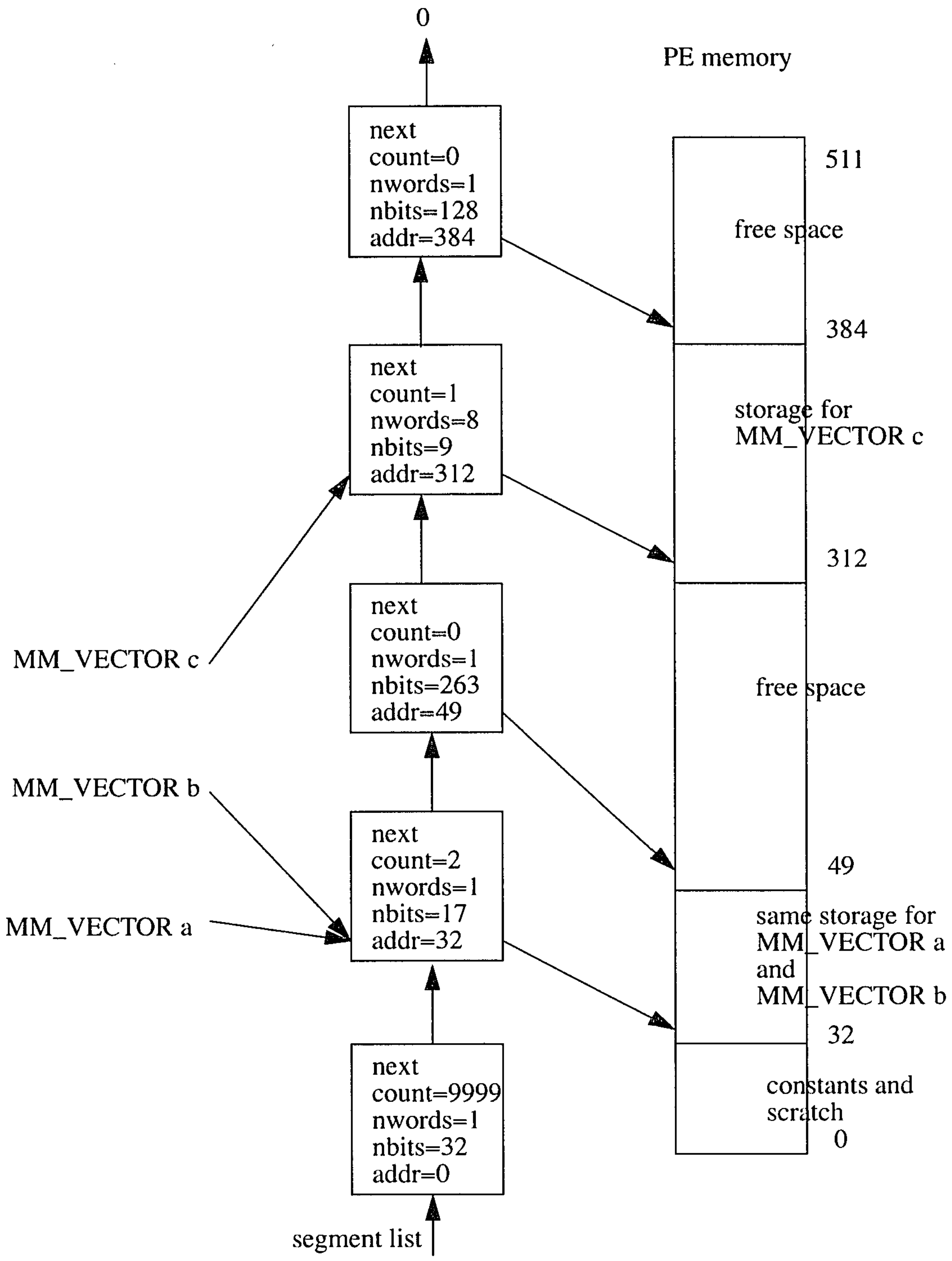


Figure 5.1 MM_VECTOR Heap

5.5 Heap Garbage Collection

As the MM32k executes, MM_VECTORs are created and freed and storage is allocated and deallocated in the heap in PE memory. Eventually, the heap may become fragmented in such a way that there is no single free segment of memory in the PE heap which is large enough to satisfy an allocation request, even though the sum of all free space segments is enough. In this case, the segments in the heap are relocated to place all the free space into a single contiguous segment. This is done by going through the segment list one segment at a time and deleting the segment if it is free space. Segments in use are copied to a lower address into space previously occupied by a free space segment. When the end of the list is reached, a new free space segment with all free space is created and placed at the end of the list. At this point the allocation request is reattempted.

A second technique can be used to recover unused PE heap space. Some MM_VECTORs may use more bits than are needed. For example, if an MM_VECTOR with 5 bits is incremented 10 times, then it will have 15 bits after the 10 increment operations because an extra bit is added to prevent overflow on each increment. However, the 15 bit MM_VECTOR really needs only 6 bits in the worst case and possibly less. This space can be recovered by checking each MM_VECTOR to determine how many bits are actually used and freeing the redundant sign bits. This technique is not currently implemented.

5.6 Overloaded Operators and Functions

The C++ class library overloads many operators and functions which allow arithmetic and other operations to be performed on MM_VECTOR variables. The following operators and functions are overloaded to work with MM_VECTORs. All of these operators and functions are implemented by functions in the C++ class library and the C++ declarations are given at the beginning of each description. The functions allocate PE storage needed to perform the operations and send vector instructions to the logical machine.

5.6.1 constructor

MM_VECTOR(const long);

This is the default constructor which is called whenever an MM_VECTOR variable goes into scope in a users program. The length of the MM_VECTOR variable is given as an argument. The number of bits is one and the MM_VECTOR is initialized to be zero. It sets the reference count in the newly allocated segment of PE memory to be one.

5.6.2 function helper constructor

MM_VECTOR(const int,const int);

This different form of the constructor is used by functions implementing the C++ overloaded operators and by the functions described here to allocate space for a result. It takes the first argument as the number of words per PE and the second argument as the number of bits per word and allocates space for an MM_VECTOR in PE memory. The PE memory is uninitialized. It sets the reference count in the newly allocated segment of PE memory to be one.

5.6.3 copy constructor

MM_VECTOR(const MM_VECTOR&);

This constructor is called the copy constructor because it copies the MM_VECTOR given as an argument to the MM_VECTOR variable is creates. It is called frequently by the compiler to pass arguments to and from subroutines. This function does not allocate any new PE memory, but sets the segment pointer to the same segment as the argument MM_VECTOR and increments the reference count.

5.6.4 destructor

~MM_VECTOR();

This function deinitializes an MM_VECTOR as it goes out of scope. It decrements the reference count of the segment of PE memory referenced. If this is the last pointer to the segment, then the reference count will be zero and the segment can be reclaimed as free space.

5.6.5 operator[]

long operator[](const long);

This operator allows the values of elements of MM_VECTORs to be referenced using [] syntax. The index number of the element is the argument. For example, the value of element 2680 of MM_VECTOR x would be x[2680].

5.6.6 set the value of an element

void set(const long,const long);

This member function sets the value of an element of a MM_VECTOR. The arguments are the index number and the new value. This will reallocate a larger MM_VECTOR with more bits per element if the value will not fit into the existing PE storage without overflow.

5.6.7 operator=

void operator=(const MM_VECTOR&);

void operator=(const long);

The = operator assigns the value of an MM_VECTOR to be the value of another MM_VECTOR. It frees the storage associated with the old assigned MM_VECTOR. If the assignment is to a scalar, then PE storage for an MM_VECTOR is allocated and initialized with the scalar. The assigned variable is then set up to reference the new MM_VECTOR.

5.6.8 operator+

MM_VECTOR operator+(const MM_VECTOR&const MM_VECTOR&);

MM_VECTOR operator+(const MM_VECTOR&,const long);

MM_VECTOR operator+(const long,const MM_VECTOR&);

The + operator adds MM_VECTORs to other MM_VECTORs or scalars and returns a MM_VECTOR. The vector length of the MM_VECTOR is the maximum of the vector lengths of the arguments. The bit length of the result is the maximum of the bit lengths of the arguments plus one.

5.6.9 operator-

MM_VECTOR operator-(const MM_VECTOR&const MM_VECTOR&);

MM_VECTOR operator-(const MM_VECTOR&,const long);

MM_VECTOR operator-(const long,const MM_VECTOR&);

The - operator subtracts MM_VECTORs from other MM_VECTORs or scalars and returns an MM_VECTOR. The vector length of the MM_VECTOR is the maximum of the vector lengths of the arguments. The bit length of the result is the maximum of the bit lengths of the arguments plus one.

5.6.10 operator- negate

MM_VECTOR operator-(const MM_VECTOR&);

The - operator negates an MM_VECTOR. The vector length of the result is the same as the argument. The bit length of the result is one more than the bit length of the argument.

5.6.11 operator|

MM_VECTOR operator|(const MM_VECTOR&, const MM_VECTOR&);

MM_VECTOR operator|(const MM_VECTOR&, const long);

MM_VECTOR operator|(const long, const MM_VECTOR&);

The | operator bitwise ORs MM_VECTORs to other MM_VECTORs or scalars and returns an MM_VECTOR. The vector length of the MM_VECTOR is the maximum of the vector lengths of the arguments. The bit length of the result is the maximum of the bit lengths of the arguments.

5.6.12 operator&

MM_VECTOR operator&(const MM_VECTOR&, const MM_VECTOR&);

MM_VECTOR operator&(const MM_VECTOR&, const long);

MM_VECTOR operator&(const long, const MM_VECTOR&);

The & operator bitwise ANDs MM_VECTORs to other MM_VECTORs or scalars and returns an MM_VECTOR. The vector length of the MM_VECTOR is the maximum of the vector lengths of the arguments. The bit length of the result is the maximum of the bit lengths of the arguments.

5.6.13 operator^

MM_VECTOR operator^(const MM_VECTOR&,const MM_VECTOR&);

MM_VECTOR operator^(const MM_VECTOR&,const long);

MM_VECTOR operator^(const long,const MM_VECTOR&);

The ^ operator bitwise exclusive ORs MM_VECTORs to other MM_VECTORs or scalars and returns an MM_VECTOR. The vector length of the MM_VECTOR is the maximum of the vector lengths of the arguments. The bit length of the result is the maximum of the bit lengths of the arguments.

5.6.14 operator<<

MM_VECTOR operator<<(const MM_VECTOR&,int);

The << operator shifts the MM_VECTOR left. The distance may be positive or negative. The vector length of the result MM_VECTOR is the same as the argument MM_VECTOR. The bit length of the result is greater than or less than the bit length of the argument MM_VECTOR by the size of the shift.

5.6.15 operator>>

MM_VECTOR operator>>(const MM_VECTOR&,int);

The >> operator shifts the MM_VECTOR right. The distance may be positive or negative. The vector length of the result MM_VECTOR is the same as the argument MM_VECTOR. The bit length of the result is greater than or less than the bit length of the argument MM_VECTOR by the size of the shift.

5.6.16 operator*

MM_VECTOR operator*(const MM_VECTOR&,const MM_VECTOR&);

MM_VECTOR operator*(const MM_VECTOR&,const long);

MM_VECTOR operator*(const long,const MM_VECTOR&);

The * operator multiplies MM_VECTORs by other MM_VECTORs or scalars and returns an MM_VECTOR. The vector length of the MM_VECTOR is the maximum of the vector lengths of the arguments. The bit length of the result is the sum of the bit lengths of the arguments.

5.6.17 operator/

MM_VECTOR operator/(const MM_VECTOR&,const MM_VECTOR&);

MM_VECTOR operator/(const MM_VECTOR&,const long);

MM_VECTOR operator/(const long,const MM_VECTOR&);

The / operator divides MM_VECTORs by other MM_VECTORs or scalars and returns an MM_VECTOR. The vector length of the MM_VECTOR is the maximum of the vector lengths of the arguments. The bit length of the result is one bit longer than the bit length of the dividend.

5.6.18 operator%

MM_VECTOR operator%(const MM_VECTOR&,const MM_VECTOR&);

MM_VECTOR operator%(const MM_VECTOR&,const long);

MM_VECTOR operator%(const long,const MM_VECTOR&);

The % operator takes the modulo of MM_VECTORs to other MM_VECTORs or scalars and returns an MM_VECTOR. The vector length of the MM_VECTOR is the maximum of the vector lengths of the arguments. The bit length of the result is one bit longer than the bit length of the dividend.

5.6.19 operator==

MM_VECTOR operator==(const MM_VECTOR&,const MM_VECTOR&);

MM_VECTOR operator==(const MM_VECTOR&,const long);

MM_VECTOR operator==(const long,const MM_VECTOR&);

The == operator compares MM_VECTORs to other MM_VECTORs or scalars and returns a one bit MM_VECTOR which indicates if they are equal. True and false results are represented by -1 and 0 respectively. The value -1 is returned because is corresponds to a one bit integer with the bit set. The vector length of the MM_VECTOR is the maximum of the vector lengths of the arguments.

5.6.20 operator!=

MM_VECTOR operator!=(const MM_VECTOR&,const MM_VECTOR&);

MM_VECTOR operator!=(const MM_VECTOR&,const long);

MM_VECTOR operator!=(const long,const MM_VECTOR&);

The != operator is similar to the == operator only the test is equality.

5.6.21 operator<=

MM_VECTOR operator<=(const MM_VECTOR&,const MM_VECTOR&);

MM_VECTOR operator<=(const MM_VECTOR&,const long);

MM_VECTOR operator<=(const long,const MM_VECTOR&);

The <= operator is similar to the == operator only the test is less than or equal.

5.6.22 operator<

MM_VECTOR operator<(const MM_VECTOR&,const MM_VECTOR&);

MM_VECTOR operator<(const MM_VECTOR&,const long);

MM_VECTOR operator<(const long,const MM_VECTOR&);

The < operator is similar to the == operator only the test is less than.

5.6.23 operator>=

MM_VECTOR operator>=(const MM_VECTOR&,const MM_VECTOR&);

MM_VECTOR operator>=(const MM_VECTOR&,const long);

MM_VECTOR operator>=(const long,const MM_VECTOR&);

The >= operator is similar to the == operator only the test is greater than or equal.

5.6.24 operator>

MM_VECTOR operator>(const MM_VECTOR&,const MM_VECTOR&);

MM_VECTOR operator>(const MM_VECTOR&,const long);

MM_VECTOR operator>(const long,const MM_VECTOR&);

The > operator is similar to the == operator only the test is greater than.

5.6.25 align function

MM_VECTOR align(const MM_VECTOR&,const long);

The align function takes an MM_VECTOR and realigns the elements such that element i is relocated to element (i+D)%length, where D is the shift distance given as an argument and length is the length of the vector. The bit length and vector length of the result vector is the same as the argument vector.

5.6.26 select function

MM_VECTOR select(const MM_VECTOR&,const MM_VECTOR, const MM_VECTOR);

MM_VECTOR select(const MM_VECTOR&,const long,const MM_VECTOR&);

MM_VECTOR select(const MM_VECTOR&,const MM_VECTOR&,const long);

MM_VECTOR select(const MM_VECTOR&,const long,const long);

The select function returns an MM_VECTOR with elements from either the second argument or the third argument depending upon whether the corresponding element in the first argument, an MM_VECTOR, is nonzero or zero, respectively. The vector length of the result vector is equal to the length of the longest argument MM_VECTOR. The bit length of the result vector is equal to the bit length of the longest of the second and third arguments.

5.6.27 truncate function

MM_VECTOR truncate(const MM_VECTOR&,const int);

This function forces the argument MM_VECTOR to have a specified bit length. The bit length of the argument MM_VECTOR may be increased or decreased. If the bit length of the MM_VECTOR is increased, then it is sign extended. If it is decreased, the most significant bits are lost and not all integers in the MM_VECTOR may be represented correctly.

5.6.28 operator~

MM_VECTOR operator~(const MM_VECTOR&);

This operator performs a bitwise not on an MM_VECTOR. The length and number of bits of the result vector is the same as the argument MM_VECTOR.

5.6.29 operator!

MM_VECTOR operator!(const MM_VECTOR&);

This operator returns a one bit MM_VECTOR indicating whether the argument MM_VECTOR is zero. The result MM_VECTOR has a value of -1 if the argument MM_VECTOR is zero. The vector length of the result is the same as the argument.

5.6.30 abs function

MM_VECTOR abs(const MM_VECTOR&);

This function returns an MM_VECTOR containing the absolute values of the argument MM_VECTOR. The vector length of the result is the same as that of the argument and the bit length of the result is one bit more than the bit length of the argument.

5.6.31 index function

MM_VECTOR index(const MM_VECTOR&);

MM_VECTOR index(const long);

This function returns an MM_VECTOR whose element values are the index numbers of those elements. The first element is 0, the second is 1, the third is 2, and so on. The length of the MM_VECTOR returned is either given explicitly as an argument or is the same as the MM_VECTOR given as an argument, depending upon the form of the function used. The bit length is enough to represent the largest index number in the vector as a positive integer. An MM_VECTOR 32768 elements long would have elements 16 bits long.

5.6.32 sum function

long sum(const MM_VECTOR&);

This function returns the sum of the elements of the MM_VECTOR given as an argument. The sum is sign extended if it is less than 32 bits, otherwise it is truncated.

5.6.33 or function

long or(const MM_VECTOR&);

This function returns the bitwise or of the elements of the MM_VECTOR given as an argument. The elements are sign extended if they are less than 32 bits.

5.6.34 and function

long and(const MM_VECTOR&);

This function returns the bitwise and of the elements of the MM_VECTOR given as an argument. The elements are sign extended if they are less than 32 bits.

5.6.35 minimum function

long minimum(const MM_VECTOR&);

This function returns the value of the minimum element of the MM_VECTOR given as an argument.

5.6.36 maximum function

long maximum(const MM_VECTOR&);

This function returns the value of the maximum element of the MM_VECTOR given as an argument.

5.6.37 first function

long first(const MM_VECTOR&);

This function returns the index number of the first nonzero element of the MM_VECTOR given as an argument. It returns -1 if all elements of the MM_VECTOR are zero.

5.7 C++ Programming Considerations

This section discusses C++ programming considerations for the MM32k. To get the maximum performance from a C++ program it is useful to understand what is actually happening on the MM32k as the program executes. This will be demonstrated by an example program, which is given below with line numbers.

```
1     #include "vector.h"
2
3     #define LENGTH 65536
4
5     MM_VECTOR gdata( LENGTH );
6
7     sample_data()
8     {
9             long i;
10            gdata = 0;
11            for ( i = 0 ; i < LENGTH ; i++ )
12                    gdata.set( i , atodin() );
13    }
14
15    MM_VECTOR filter_data( const MM_VECTOR & unfiltered )
16    {
17            MM_VECTOR left( LENGTH ), right( LENGTH );
18            MM_VECTOR result( LENGTH );
19            left = align( unfiltered, 1 );
20            right = align( unfiltered, -1 );
21            result = ( left + right + unfiltered ) / 3;
22          result = select( result > 1023 , 1023 , result );
23          result = select( result < -1024 , -1024 , result );
24          result = truncate( result , 11 );
25          return result;
26  }
27
28  main()
29  {
30          while ( maximum( gdata ) < 100 ) {
31                  sample_data();
32                  gdata = filter_data( gdata );
33          }
34  }
```

The program given above goes into a loop which reads data into an MM_VECTOR and filters it. It is described on a line by line basis below.

Line1 includes the declaration of the C++ class library and must be included in any C++ file which uses MM_VECTOR variables.

Line 3 defines the length of the MM_VECTORs which will be created. The length, 65536, is twice as long as 32768, so each MM_VECTOR will be wrapped around the 32768 PEs twice.

Line 5 declares a global MM_VECTOR variable which is 65536 elements long and is initialized to zero. When this variable goes into scope, the compiler calls the C++ class library constructor function, which allocates a segment of PE memory heap space to contain its data. The variable is represented by a pointer to the allocated structure in the segment list. Since the scope is global, the constructor is called before the main program is called and the destructor will be called after the main program exits.

Lines 7 through 13 are the declaration of the sample_data function, which reads integers using the function atodin and places them in the MM_VECTOR gdata.

Line 9 declares an index variable i which is used to count the integers read.

Line 10 sets the value of gdata to be zero initially. This is done here to effectively throw away the old integers contained in gdata. If not done here, then gdata might require more bits than necessary.

Lines 11 and 12 form a loop to read the integers and put them in the MM_VECTOR. The integers are put into gdata by calling the member function set with the index and value as parameters. The member function set calculates the number of bits required to represent the integer without truncation and will automatically increase the number of bits in the MM_VECTOR gdata if necessary. The member function set calls the mm_simd_write() macro which places data in the PE memory of the MM32k.

Lines 15 through 26 declares the function filter_data which takes an MM_VECTOR as an argument and returns a new MM_VECTOR which has been filtered as a value. This function demonstrates that MM_VECTORs can be passed to and returned from C++ functions like ordinary variables. The MM_VECTOR argument is passed by reference and therefore requires no MM32k activity.

Lines 17 and 18 declare three MM_VECTORS, left, right, and result, which are local in scope to the function filter_data, 65536 elements long, and initialized to zero. Similar to standard C++ local variables, space for those MM_VECTORs is automatically allocated on the MM32k upon entry to filter_data, and is released upon exit.

Line 19 sets MM_VECTOR left to be a realigned copy of the argument MM_VECTOR unfiltered. The compiler calls the align member function which returns a new

MM_VECTOR containing the same data as the old, except shifted from element i to element i+1. The align member function allocates PE heap space for the new MM_VECTOR and calls the mm_simd_align macro, which issues a vector instruction to the MM32k to realign the data in the left MM_VECTOR and put the result in the newly allocated space. The compiler then automatically calls the = operator member function, which frees the old value of the left MM_VECTOR and sets the new value to be the result of the align member function.

Line 20 is just like line 19, except that it shifts element i to element i-1 and sets the result in the right MM_VECTOR.

Line 21 computes the sum of the left, right, and unfiltered MM_VECTORs, divides the sum by 3, and assigns the result to the MM_VECTOR result. The compiler will automatically generate a series of temporary variables to hold the intermediate results of this computation. It will automatically call the + operator, the / opertor, the = operator, the constructor, copy constructor, and the destructor member functions as it deems necessary to evaluate the expression. These member functions will issue vector instructions to the MM32k, allocate and deallocate PE heap memory, change reference counts, and manipulate pointers to perform their individual operations. The programmer has no direct control over how the compiler evaluates this expression, beyond the explicit evaluation rules defined for the C++ language.

Line 22 limits the value of result to be less than or equal to 1023. The > operator returns a one bit MM_VECTOR which is passed to the select function as an argument. This line illustrates that MM_VECTOR expressions can be passed to a function as an argument. The compiler will create a temporary MM_VECTOR variable to hold the result of the comparison, and call appropriate MM_VECTOR class member functions to create it, compute it, and destroy it. The > operator member function and the select member function will issue vector instructions to the MM32k to perform the computation.

Line 23 is just like line 22 except that it limits the value of the result to be greater than or equal to -1024.

Line 24 forces the number of bits of the result MM_VECTOR to be 11. Since the lines 22 and 23 limited the values to be in the range 1023 to -1204, the result will require no more than 11 bits to represent without loss of accuracy. This line illustrates how the programmer can take explicit control of the number of bits used to represent an MM_VECTOR.

Line 25 returns the result MM_VECTOR to the calling program. The compiler automatically generates a call to the copy contructor of the C++ class library which copies the local MM_VECTOR into the scope of the calling function. This operation is performed by creating a second pointer to and incrementing the reference count of the element of the segment list which represents the local result MM_VECTOR. Then the local result MM_VECTOR variable is destroyed by calling the destructor member function, the reference count will be decremented. In short, the copy operation requires no MM32k activity, only pointer manipulation on the host.

Line 26 is the end of scope for the MM_VECTOR variables left, right, and result. When the return in line 25 is executed, the compiler automatically generates a call to the destructor member function for the MM_VECTOR class for each of these variables. The destructor function decrements the reference count in the segments of PE memory to which each of these variables refer. The destructor is called after the returned result variable has been copied into the scope of the calling function.

Lines 28 through 34 are the declaration of the function main, which is the body of the program.

Lines 30 through 33 are a loop which reads and filters data while the MM_VEC-

TOR gdata is less than 100. The function maximum takes an MM_VECTOR as an argument and returns a long integer value equal to the maximum element of the argument. The compiler calls the member function maximum and compares the returned value against 100. The function sample_data is called which reads and fills the MM_VECTOR gdata with integers. The function filter_data is called with the MM_VECTOR gdata as an argument. It returns an MM_VECTOR which is a filtered, limited version of the argument, which is assigned back to MM_VECTOR gdata. These lines illustrate a program using MM_VECTOR variables as like normal C language variable. They are passed to functions as arguments and returned from functions as argument.

Line 34 is the last line of the function main, which is the body of the program. When the main function exits and the program stops, the MM_VECTOR variable gdata goes out of scope. The destructor is called at this point for gdata.

Chapter 6

RESULTS

The purpose of this chapter is to describe the performance of the MM32k computer, to identify limitations, and to suggest improvements. The first section describes the raw performance of the vector instruction set. The second section shows how MM32k performance is effected by the host computer. The third section compares the MM32k against performance estimates for other SIMD parallel computers. The fourth section compares the MM32k against existing serial computers on neural network applications. The fifth section discusses performance limitations and how they might be improved.

6.1 Vector Instruction Set Performance

The purpose of this section is to describe the performance of the MM32k on the vector operators of the C++ class library. The number of millions of operations per second, or MOPS, measured for each overload C++ operator is listed in the table below. The measurements were made on a 66 megahertz 486DX2 PC. Each operation was repeatedly executed in a loop for 10 seconds and the performance reported corresponds to the average time per iteration of the loop. Both vector-vector and vector-scalar versions of the operators are listed and all operands have eight bits of precision. The performance of the MM32k is listed for two cases, corresponding to vector lengths of 32768 and 262144. The first case is the shortest vector which MM32k can work with and corresponds to one vector element per processor. The second case is the size of a standard image used in industrial machine vision applications and corresponds to eight vector elements per processor.

Another issue is how efficient the MM32k implementation is. A SIMD array utilization figure is listed with each absolute performance figure and indicates how close the

MM32k C++ class library, host, and controller together come to fully utilizing the computational resources present in the MM32k hardware. These figures are expressed as a percent of the theoretical maximum throughput when using the best known algorithm for each operation.

Table 6.1 MM32k Vector Performance

operation	actual MOPS with length of 32768	SIMD utilization with length of 32768	actual MOPS with length of 262144	SIMD utilization with length of 262144
v[s]	.042	8%	.042	8%
v.set(s,s)	.078	20%	.079	20%
v=s	2021	9%	14032	63%
v+v	1455	56%	2074	80%
v+s	1864	42%	3457	78%
s+v	1864	42%	3457	78%
v-v	1454	56%	2074	80%
v-s	1864	42%	3457	78%
s-v	1864	42%	3457	78%
-v	1989	45%	4127	93%
v|v	1945	31%	4260	68%
v|s	1864	15%	6065	49%
s|v	1864	15%	6014	48%
v&v	1945	31%	4285	69%
v&s	1864	15%	6014	48%
s&v	1903	15%	6065	49%
v^v	1945	31%	3651	58%
v^s	1826	15%	6065	49%

Table 6.1 MM32k Vector Performance

operation	actual MOPS with length of 32768	SIMD utilization with length of 32768	actual MOPS with length of 262144	SIMD utilization with length of 262144
s^v	1864	15%	6065	49%
v<<s	1955	19%	8226	78%
v>>s	1988	12%	11580	70%
v*v	206	86%	215	90%
v*s	426	90%	450	95%
s*v	406	86%	450	95%
v/v	249	72%	264	76%
v/s	271	71%	282	73%
s/v	271	71%	282	73%
v%v	249	72%	259	75%
v%s	263	68%	274	71%
s%v	263	68%	274	71%
v==v	1789	45%	3153	79%
v==s	1945	18%	6626	60%
s==v	1903	17%	6626	60%
v!=v	1789	45%	3167	79%
v!=s	1945	18%	6626	60%
s!=v	1945	18%	6565	59%
v<=v	1789	45%	3167	79%
v<=s	1903	17%	6014	54%
s<=v	1903	17%	6169	56%
v<v	1789	45%	3167	79%
v<s	1945	18%	6014	54%

Table 6.1 MM32k Vector Performance

operation	actual MOPS with length of 32768	SIMD utilization with length of 32768	actual MOPS with length of 262144	SIMD utilization with length of 262144
s<v	1945	18%	6223	56%
v>=v	1789	45%	3167	79%
v>=s	1903	17%	6223	56%
s>=v	1903	17%	6014	54%
v>v	1789	45%	3167	79%
v>s	1903	17%	6223	56%
s>v	1903	17%	6014	54%
align(v,s)	186	67%	213	76%
select(v,v,v)	1963	37%	3596	72%
truncate(v,s)	1936	14%	10372	73%
~v	1936	16%	9392	75%
!v	1936	17%	9858	84%
abs(v)	1863	57%	2909	89%
index(s)	1688	27%	4337	76%
sum(v)	52	50%	306	56%
or(v)	123	75%	923	85%
and(v)	123	75%	916	85%
minimum (v)	114	74%	754	81%
maximum (v)	114	74%	754	81%
first(v)	62	30%	393	47%

The v[s] operator is different in that it is not a parallel operation, but rather involves returning the value of a single element of a vector. It is slow because each bit of the returned value must be individually read from processing element memory, because the controller wastes cycles, and because the overhead associated with fetching the instruction and returning the value is large compared to the time required to read the bits. It is also slow because the next vector instruction cannot be issued until the value from the first is returned, which eliminates the effectiveness of the vector instruction queue in the controller. The time required does not depend upon the length of the vector, so the short and long vector cases are the same.

The vls operator is an example where the MM32k can execute vector instructions a little faster than the host can issue them. This operator requires just 2*8=16 SIMD processing element instructions per result and performs 1864 and 6065 million operations per second in the short and long vector cases, respectively. In the short vector case, the performance of the instruction is limited by the rate at which the host can issue instructions. In the long vector case, the performance is limited by vector instruction fetch and decode overhead. Processing element array utilization is poor and fair, respectively.

The v+v operator is an example where the host can issue vector instructions a little faster than the MM32k can execute them. This operator requires 9*8=72 SIMD processing element instructions per result and performs 1455 and 2074 million operations per second in the short and long vector cases, respectively. The host takes advantage of the vector instruction queue and issues many instructions in advance. In the short vector case, the performance is limited because of instruction fetch and decode overhead. In the long vector case, the performance is limited by the rate at which the processing element array can execute instructions. Processing element array utilization is fair and good, respectively.

The v*v operator is an example where the host can issue vector instructions much faster than the MM32k can execute them. The operator requires 13*8*8=832 SIMD processing element instructions per result and performs 206 and 215 million operations per second in the short and long vector cases, respectively. The vector instruction queue is largely irrelevant in this case because the host is so much faster than the MM32k. The performance is limited by the rate at which the processing element array can execute instructions and processing element array utilization is good.

The sum(v) operator is and example of a reduction operator. The host is much faster than the MM32k. The algorithm first forms sums in processing elements, second, forms sums across processing elements on each page, and third, forms sums across pages. The first operation is very fast compared to the second and third operations since there is no inter processor communication required. The second and third operations are independent of the length of the vector, since they must be done across all processor in all pages regardless. This means that the amount of time required to execute this instruction is largely independent of the length of the vector. Performance on the short and long vector cases is 53 and 306 million operations per second. This explains why performance is around six times greater for a vector which is eight times longer.

In actual application code, the vector instruction queue can sometimes improve the performance of some operations above that indicated in the table. If fast and slow vector instructions are intermixed in an application, then the host can issue the fast ones while the slow ones are executing, preventing the vector instruction queue from emptying and preventing the MM32k from waiting on the host. The effective operation rate of the fast instructions will be improved in this case.

In general, the overall performance and processing element array utilization is highest with longer vectors. This is because the vector instructions fetch and decode overhead is a smaller proportion of the total execution time and because the host is more likely to be faster than the MM32k. The performance and processing element array utilization seem to be good for most vector instructions with vectors 262144 elements long, which corresponds to the standard 512*512 pixel image used in industrial machine vision applications.

6.2 Performance and the Host

The purpose of this section is to show how the MM32k is affected by the performance of the host computer. In the table below, the performance of the v+v and v*v operators are listed for three different hosts. The hosts tested were a 20 megahertz 386 PC, a 33 megahertz 486 PC, and a 66 megahertz 486DX2 PC. The vector operands were 32768 elements long with eight bits per element. The performance is listed as millions of operations per second, or MOPS.

Table 6.2 Performance Across Hosts

operation	MOPS with 20 mhz 386	MOPS with 33 mhz 486	MOPS with 66 mhz 486
v+v	266	917	1454
v*v	206	213	206

If the MM32k is faster than the host, as it is in the v+v example, then the performance of the system scales with the performance of the host. If the host is faster than the MM32k as it in the v*v example, then the performance of the host is not important. In typical applications, there are usually a mix of fast and slow vector operations and the vector instruction queue will be allow the MM32k to attain the average performance of the mix. To obtain the maximum performance form the MM32k, the host should be faster than the average vector instruction.

6.3 Comparison to Other SIMD Machines

The purpose of this section is to compare the MM32k against other SIMD parallel computers. It attempts to place the MM32k into perspective with other SIMD parallel computers, but does not attempt to make exact performance comparisons on particular neural network algorithms.

The first parallel SIMD computer compared is the CNAPS/VME, which is manufactured by Adaptive Solutions of Beaverton, Oregon. It is a single board system implemented with four custom chips, each with 64 processing elements, and is optimized for multilayer perceptron neural networks. It can be configured with up to 16 megabytes of additional memory attached to the processors through a 40 megabyte per second bus. The bandwidth list for the switch was derived by counting the number of interconnection wires and multiplying by the clock rate. However, the practical bandwidth of the switch is believed to be much lower than the figure listed. For example, the system is roughly the same speed as the MM32k when doing a three by three image convolution, an operation dominated by inter processor communication. It is described by Hammerstrom [1990], and is discussed in chapter 2 of this thesis.

The second parallel SIMD computer compared is the Connection Machine CM-1, manufactured by Thinking Machines in Cambridge, Massachusetts. It is a complete supercomputer system with a host computer and an input-output subsystem. This machine supports individual random addressing of each data packet sent through the switch, although not at the bandwidth listed in the table, which refers to two dimensional mesh addressing. It is described by Hillis [1985].

The third parallel SIMD computer compared is the MasPar MP-1, manufactured by MasPar Computer Corporation, of Sunnyvale California. It is also a complete supercomputer system with a host computer and an input-output subsystem. This machine also supports individual random addressing of each data packet sent through the switch, although at a greatly reduced bandwidth, compared with that listed in the table which refers to two dimensional mesh addressing. It also supports random addressing of data by each processing element. A weakness is that the peak memory bandwidth is almost ten times slower than the peak processing rate and that the processing rate therefore refers to register to register operations, not memory to register operations. It is also capable of floating point arithmetic. It is described by Nickolls [1990].

The SIMD computers are compared on the basis of eight attributes. The first attribute is the number of processors. This is important because there must be enough parallelism to keep all processors busy for the SIMD computer to be efficient on a particular neural network. The next attribute is the width of the internal data paths, which determines the degree to which the processor takes advantage of the parallelism of bits across words. It is usually small when there are a lot of processors and larger when there are fewer. The next attribute is the number of chips which are present in the SIMD array. This is a rough indication of the implementation efficiency of the machine. The next attribute is power consumed. This is another indicator of the efficiency of a system. Both of these attributes, along with cost, influence the suitability of the system for use in dedicated applications. The next attribute is the peak number of eight bit additions per second which the machine can perform. This is a key operation in many neural networks and eight bits is a typical precision. The next attribute is the peak number of eight bit by eight bit multiply accumulate operations which the machine can perform. This is also a key operation in many neural networks and eight bits is typical. The next attribute is the total SIMD memory. This limits the size of the problem which the processor can address and includes only memory which is directly connected to the SIMD processing elements on a per element basis. The next attribute is the peak memory bandwidth. This limits the rate at which the processor can access all stored information and is an important strength of the SIMD architecture. The next attribute is the peak SIMD switch bandwidth. This indicates the peak rate at which the processors can exchange data and is also an important strength of the SIMD architecture. The next attribute is board cost, which gives the cost without considering the host computer. The CM-1 and MP-1 are not sold without a host and so this information is missing. The final attribute is total system cost. The MM32k is listed with a PC host added, the CNAPS machine is listed in a server configuration with a $10,000 workstation added, and the CM-1 and MP-1 machines are full supercomputers with a host computer and an input-output subsystem.

Table 6.3 Other SIMD Computers

attribute	MM32k	CNAPS	CM-1	MP-1
number of processors	32768	256	65536	1024
data path width in bits	1	16	1	4
chips in SIMD array	17	5	20000	512
power in watts	10	20?	12000	50
additions in MOPS	2500	3840	8000	6400
multiply accumulate in MOPS	200	3840	1000	1600
total mem-ory in megabytes	2	1	32	64
memory bandwidth in mega-bytes per second	25,000	10,240	32,000	680
switch bandwidth in mega-bytes per second	280	2560	6000	1400
board cost	$4000	$27,000	-	-
system cost	$7000	$64,950	$3,000,000	$150,000

The MM32k occupies a unique niche when compared against other SIMD parallel computers. It performs addition at a rate comparable with the other processors and therefore should have comparable performance on local basis function type neural networks and nearest neighbor search paradigms which utilize hamming distance metrics. It is less efficient at multiply-accumulate operations and is less well suited for networks for which the computation is dominated by dot products. It has a very high memory bandwidth, but cannot use this bandwidth effectively due to the inefficiencies of its instruction set. It has a low peak switch bandwidth but can use this bandwidth efficiently in many cases. It is by far the lowest in complexity, power consumption, and cost.

6.4 Miller-Scalera Neural Network Benchmarks

The purpose of this section is to discuss the performance of the MM32k on a series of actual neural networks. The neural network programs were developed to test the performance of the MM32k against some popular traditional computers in performing the same task.

Five different computers were tested for comparison. The first tested was the MM32k, which was installed in a 66 megahertz 486 host. The second machine was an ASTRIX 486 PC, manufactured by ASTRIX, with a 66 megahertz 486-DX/2 processor. The third machine was a DECstation 5000 Model 200, manufactured by Digital Equipment Corporation, with a 25 megahertz MIPS R3000A processor. The fourth machine was a DECstation 3000 Model 500 AXP, manufactured by Digital Equipment Corporation, with a 150 megahertz DECchip 21064 ALPHA AXP processor. The fifth machine was a SPARCstation 10 Model 30, manufactured by Sun Microsystems, with a 33 megahertz SuperSPARC processor. The neural networks were coded in C++ using the MM_VECTOR class library to execute on the MM32k and were coded in C for execution on the other computers. The applications were compiled and executed using the 32 bit memory model on the ASTRIX 486 machine and using the native C compiler on the remaining machines, all with execution speed optimizations enabled.

All benchmarks report both time per recall and speedup ratio. The speedup factor indicates how many times faster the MM32k was. All benchmarks, except the multilayer perceptron, report results for networks with both 32768 and 65536 neurons. The MM32k should perform better on the larger networks because the vector instruction fetch and decode overhead is a small part of total time. These benchmarks were performed by Professor W. Thomas Miller and Steve Scalera during the Spring of 1993 and are described in Miller [1993].

6.4.1 Nearest Neighbor Search

This benchmark compares the nearest neighbor algorithm during recall. This test uses the city block distance metric and has 16 eight bit elements in each neuron. There are 32 additions or subtractions, 16 absolute value operations, and one global search for the maximum. This neural network is ideal for the MM32k in that it is highly parallel and has very little inter neuron communication. In the best case, the MM32k achieves 338 million connections per second which corresponds to 1030 million operations per second. The poor performance of the MIPS machine is believed to be due to a subroutine call to implement absolute value, where the other processors used in-line code.

Table 6.4 Nearest Neighbor Performance

processor	time in msec using 32768 neurons	time in msec using 65536 neurons	speedup ratio using 32768 neurons	speedup ratio using 65536 neurons
MM32k	2.2	3.1	1	1
486	350	700	159	226
MIPS	970	1860	441	600
ALPHA	81	177	37	57
SPARC	410	820	186	265

6.4.2 Radial Basis Function Neural Network

This benchmark compares performance on the radial basis function neural network during recall. The network has nine features of eight bits each and three 24 bit outputs. It computes a true euclidean distance measure. This test is highly parallel, but includes multiply and divide operations at which the MM32k is less efficient. There are 21 additions or subtractions, nine multiplications, three divisions, and three global sums. In the best case, the MM32k achieves 94 million connections per second which corresponds to 280 million operations per second.

Table 6.5 Radial Basis Function Performance

processor	time in msec using 32768 neurons	time in msec using 65536 neurons	speedup ratio using 32768 neurons	speedup ratio using 65536 neurons
MM32k	5.1	8.4	1	1
486	820	1640	161	195
MIPS	920	1710	180	204
ALPHA	160	330	31	39
SPARC	480	960	94	114

6.4.3 Kohonen Neural Network

This benchmark compares performance on the Kohonen neural network during training. The network has an eight dimensional input space which is mapped into a two dimensional Kohonen layer. A Euclidean distance measure is used. The test is highly parallel while the closest neuron is being identified. There are 16 additions or subtractions, eight multiplications, one divide, one global search for the minimum, and one global search for the first nonzero neuron. The conventional processors have an advantage during weight update because only five percent of the neurons need be modified. During weight update, there are an additional 27 additions or subtractions, 10 multiplications, and eight divisions. In the best case, the MM32k achieves 26 million connection updates per second which corresponds to 325 million operations per second. If the network were modified to increase the size of the neighborhood affected by training, the traditional computers would be slower while the MM32k speed would remain the same.

Table 6.6 Kohonen Performance

processor	time in msec using 32768 neurons	time in msec using 65536 neurons	speedup ratio using 32768 neurons	speedup ratio using 65536 neurons
MM32k	10	20	1	1
486	760	1500	76	75
MIPS	690	1250	69	63
ALPHA	110	210	11	11
SPARC	490	970	49	49

6.4.4 Multilayer Perceptron

This benchmark compares performance on a multilayer perceptron during recall and training. The multilayer perceptron has four inputs, 8192 hidden units, and four outputs. There are 32768 weights in the hidden layer and 32768 weights in the output layer. The arithmetic was done in scaled fixed point with 15 bits to the right of the decimal point. The times reported are for one recall cycle and one training cycle, which take approximately one third and two thirds of the total time respectively. Since there are 65536 weights, the best MM32k performance corresponds to 4.9 million connections per second and 2.5 million connection updates per second. The MIPS machine was not tested.

Table 6.7 Multilayer Perceptron Performance

processor	time in msec	speedup ratio
MM32k	40	1
486	750	19
MIPS	-	-
ALPHA	110	2.8
SPARC	450	11

6.4.5 Discussion of Miller-Scalera Benchmarks

The Miller-Scalera benchmarks demonstrate that the MM32k performs well on real applications using the C++ class library. In comparison to the 486, the MIPS, and the SPARC microprocessors, the MM32k was 114 to 265 times faster on the nearest neighborhood and radial basis function neural networks. It was 49 to 75 times faster on the Kohonen network, which offers the serial microprocessors some advantages over the parallel MM32k. The MM32k was 11 to 19 times faster on the multilayer perceptron to which it is less well suited. The ALPHA microprocessor was four to seven times faster than the other three microprocessors compared in the benchmarks and the MM32k speedup factors were proportionately lower. However, the ALPHA workstation is much more expensive than workstations based on the other microprocessors or MM32k.

6.5 Discussion of Performance Limitations

The MM32k has several factors which adversely affect performance. The processing element instruction set, the switch, global operations, and the controller are each inefficient in certain situations. The host may not be able to keep the MM32k busy and in some applications, there may not be enough SIMD memory. There is no way to quickly get data into and out of the MM32k. These situations and possible remedies will be discussed in this section.

6.5.1 SIMD Instruction Set Inefficiencies

The instruction set of the MM32k processing elements was not designed to do arithmetic, but rather to do bit plane operations common to computer graphics. It can invert, copy, and conditionally copy bits, but can not perform arithmetic efficiently. As an example, it takes nine memory cycles to add two bits in memory and store the result in memory, when it should take only three, two to fetch the operands and one to save the result. The instruction set could be enhanced to allow addition to be performed in a single step.

The time to perform multiplication and division could be reduced as well. To multiply an M bit number by an N bit number requires M*N*13 steps. A more efficient instruction set could reduce the constant 13 to a lower figure. Additionally, up to M partial product circuits could be built to proportionally speed up the multiply. A similar improvement could be made for division.

6.5.2 Switch Inefficiencies

To shift a bit across all processors, the MM32k must move one bit from each processor off of the page and out of the chip, through the switch, and back into the destination page. Since there are 512 processors to each path through the switch, this requires 512 cycles. This is inefficient when the switch shift distance is less than 512. The most frequent shift distances are 1 and -1, which corresponds to one bit being moved from page to page and 511 bits being moved to neighboring processors on the same page. In this case, the MM32k does not take advantage of the potentially high intra chip bandwidth. This operation could potentially be speeded up by a factor of 512, although overhead would reduce this ratio.

The switch could also be given more bandwidth by making it wider. Currently, there are 512 processors per page, but a version of the chip exists with 256 processors per page. If this version of the chip were used, the same number of processors would be implemented with the same number of chips, but the number of data paths to the switch would double. This could potentially double the switch bandwidth, depending upon the operation being performed and the trade-offs made.

6.5.3 Global Operation Inefficiencies

Global operations are performed using the switch to move data around and the processing element ALUs to compute the results, requiring many processing element and switch cycles. Most global operations accumulate results on a per page basis in a single processing element, which is then read by the controller. If each page were given appropriate hardware, it could compute these results in a single step. An eight bit global sum operation, which currently takes around one millisecond, could be speeded up by a factor of 100.

6.5.4 Controller Inefficiencies

The existing controller design is not always able to keep the SIMD array busy. The duty cycle varies according to the type of operation and the number of bits and words which are processed per processing element. The SIMD processor array is idle as each vector instruction is being fetched and decoded by the controller, which can take up to 10 microseconds per instruction. The processor array is also sometimes idle while the instruction is being executed due to microcode program overhead in the controller. On the other hand, the controller must often waste microcode cycles while the SIMD processors are active. The SIMD processor array could be better utilized if there were a SIMD instruction fifo implemented in hardware between the controller and the processor array. Since the controller would be no longer tightly coupled to the processor array, it could run faster, and avoid null cycles, which would reduce the likelihood that the fifo would become empty. The duty cycle of the SIMD processor array can be as low as 50 percent when executing add instructions on 32768 element vectors of eight bits. A fifo could potentially double performance in this case.

6.5.5 Host Inefficiencies

The host computer may not be able to send vector instructions to the MM32k controller as fast as they can be executed. There are a number of ways to speed up the rate at which the host can issue instructions. First, the hardware interface between the host and the controller could be made cleaner and faster. It currently requires vector instructions to be issued in six word groups even though many instructions do not have word sizes which are multiples of six. Second, the host could also issue vector instructions without using the C++ class library. This would relieve the host of SIMD heap management by allowing the application programmer to allocate the heap in a fixed way at compile time. This is more complicated for the programmer. Third, the vector instruction set could be enhanced with more complicated instructions which take longer but do more work. A single complex instruction would replace many simpler ones. For instance, the inner loops of some neural networks or image processing operations could be replaced with a single vector instruction. Two disadvantages are that the new instruction might not be flexible enough to solve a particular problem and that the application programmer would have to know to use it. The last solution offered is to get a faster host computer and place the MM32k on a faster input-output bus. Microprocessors faster than the Intel 80x86 series are available and the PCAT bus is slow.

6.5.6 System Memory

Many applications need more than two megabytes of total system memory. A simple solution is to add more processing elements until the total memory is enough to accommodate the problem. However, this type of memory is expensive compared to commodity DRAM, which uses more dense technology and is produced in higher volume.

A different solution is to add commodity DRAM and attach it to the MM32k through the switch. The amount of memory which can be attached by this method is limited only by cost, and the peak memory bandwidth could be up to 640 megabytes per second. The C++ class library could be extended to handle this memory transparently by implementing virtual memory. Each MM_VECTOR variable would know whether it is in the processing element memory or in the external DRAM memory. If it is in the processing element, the computation could proceed unaffected. If it is in the external DRAM memory, then the needed data could be swapped into processing element memory. The operations analogous to address mapping and page faulting could be handled by the C++ class library. The C++ class library programming interface would remain the same.

6.5.7 Data Input-Output Bottleneck

There is no way to get large amounts of data into and out of the MM32k. Data must be moved into or out of the elements of a vector one element at a time. Due to the organization of data words in SIMD processor memory, each bit must be individually written or read. The bandwidth is an order of magnitude below what the PC bus is capable of. Block data transfer operations could be added and supported by microcode. The data could be transferred in or out in a block at speeds approaching the limit imposed by the PC bus. The block of data could be transformed to and from internal SIMD format by a single corner turn operation. This scheme might accomplish data transfer at a one megabyte per second rate.

Data can be transferred into and out of the MM32k at a faster rate through the switch. This data path could be up to 128 bits wide and could operate at a 40 megahertz rate for a peak bandwidth of 640 megabytes per second. A video rate interface based on the DT-Connect standard has been implemented in prototype form. It is capable of transferring data at a 20 megabyte per second rate.

Chapter 7
CONCLUSION

This chapter summarizes the description of the MM32k presented in this thesis. The MM32k is a SIMD parallel computer with 32768 one bit processing elements, each with 512 bits of memory and a connection to a switch. It can be programmed effectively using a C++ class library running on a host computer and is very low in cost when compared to other SIMD parallel computers. It can execute some types of local basis function neural networks over 250 times faster than traditional serial computers. The MM32k can make some applications practical when they would not otherwise be so due to their high computational requirements.

7.1 The MM32k SIMD Processor Array

The MM32k hardware is composed of a controller, a processor array, and a switch. The controller receives vector instructions from the host and implements them by sending the processor array a series of processing element instructions. The processor array consists of 64 pages, each with 512 processors and a connection to the switch. There are a total of 32768 processing elements, each of which contains 512 bits of memory and a one bit ALU. The pages each can shift data out of the contained processing elements, through the switch, and back into different processing elements on the same or different pages.

The MM32k is viewed by the host computer as an array of 32768 processing elements, each with 512 bits of memory and a bit serial ALU. The controller implements a vector instruction set which allows the processing elements to perform arithmetic operations on data stored in processing element memory. The vector instruction set allows the host to manipulate variable length vectors whose elements are variable precision integers.

A C++ class library abstracts the MM32k using the MM_VECTOR class, which represents a vector of integers. The length of the vectors and the bit precision of the integers is variable. Traditional C language arithmetic operators, such as +, -, *, and / are overloaded to operate on variables of the MM_VECTOR class. The class library, which runs on the host computer, automatically handles the allocation of space in MM32k processing element memory and keeps track of the number of bits required to represent each MM_VECTOR variable without overflow. The heap management strategy and data structures are described. The overloaded operators and functions of the C++ class library are described.

The performance of the MM32k has been measured to be in the giga operation per second range for many arithmetic operations when using the C++ class library. Due to controller inefficiencies, most operations do not fully utilize the processor array, particularly those on short vectors. When compared in general terms to other SIMD parallel computers, the MM32k was found to be similar in some measures of performance, although much cheaper. Some well known neural network algorithms were implemented on both the MM32k and on a variety of traditional serial computers, and the MM32k was found to be over 250 times faster in some cases.

The MM32k is not limited to neural network applications. It is a general purpose SIMD parallel computer with high level language support and can effectively address many problems which can be expressed as long vectors. It can also be connected to external input-output devices. For example, it could be programmed to preprocess speech or image data to produce a feature vector. The feature vector could then be analyzed using a neural network also implemented on the MM32k, allowing both computations to be performed on a single machine.

7.2 Future Work

The future work related to the MM32k is divided into two groups, improvements in the MM32k and neural network implementations. The first group deals with improvements to the hardware, firmware, and C++ class library of the MM32k. The second group deals with neural network algorithms which might exploit the MM32, but which were previously discounted due to low performance.

7.2.1 MM32k Improvements

The MM32k hardware can be improved by modifications to the chips which contain the processing elements. The instruction set of the SIMD processing elements could be made more efficient and the switch performance could be improved on short shifts. Modifications to the chip could also improve the performance of global operations.

The board level design could also be improved. A fifo queue for SIMD instructions could help the controller keep the processor array busy, particularly on short vector operations. The bandwidth of the host interface port could be improved. The size of the machine could be increased to increase the processing power, provided there is enough parallelism available in the problem. The width of the switch could also be increased to improve the performance of the align operation and some global operations.

The amount of memory in the system could also be increased. Extra bulk DRAM memory could be attached to the switch. The C++ class library running on the host could then implement a scheme similar to virtual memory to increase the apparent size of processing element memory, without disturbing the existing C++ programming model.

A high speed input-output path through the switch could also be added. A video rate interface is currently in prototype form and needs to be completed.

The vector instruction set for the MM32k can also be enhanced to implement more complicated vector instructions. These instructions might implement a significant portion of a neural network algorithm, reducing the overhead of issuing individual vector instructions. Also, many small changes could improve the controller microcode and the efficiency of the vector instruction set.

7.2.2 Neural Network Implementations

This thesis has not seriously addressed neural network algorithms which might be implemented on the MM32k. It is hoped that many variations on local basis function algorithms, which may have been previously discounted due to high anticipated computational requirements, will be revisited since a low cost implementation is available. For example, past implementations emphasized minimum numbers of high precision neural units, a technique well suited to sequential machines. The MM32k could be used to explore neural networks with large numbers of low precision neural units, since the MM32k has a large number of processing elements and execution speed increases as operand bit length decreases.

It is also hoped that new local basis function training algorithms, which take advantage of the massive parallelism of the MM32k, will be explored. For instance, redundant neurons could be identified and removed from a network at the same time new neurons containing new data samples from an on-line learning experiment are being allocated. Such a network could learn on line while it is performing recall.

It is also hoped that local basis function networks will be used in real time robot control experiments. Local basis function networks learn quickly and can learn on line, properties which are important for control experiments. The MM32k can implement nearest neighbor networks fast enough to implement control loops in the 100 to 1000 hertz range.

REFERENCES

D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, A learning algorithm for Boltzmann machines. cognitive Science, 9:147-169, 1985.

A. J. Agranat, C. F. Neugebauer, and A. Yariv, "A CCD based neural network integrated circuit with 64K analog programmable synapses," in Proc. IJCNN (San Diego), 17-21 June 1990, pp. II551-II556.

A. Aho, R. Sethi, and J. Ullman. Compilers: Principals, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.

J. S. Albus, "A New Approach to Manipulator Control: the Cerebellar Model Articulation Controller (CMAC)," Transactions of the ASME, pp. 220-227, September 1975.

I. Aleksander, ed., Neural Computing Architectures, The MIT Press, Cambridge, MA, 1989.

J. Anderson, J. C. Platt, and D. B. Kirk, "An Analog VLSI Chip for Radial Basis Functions," in Advances in Neural Information Processing 5, Morgan-Kaufmann, San Mateo, CA, pp. 765-772, 1992.

A. G. Andreou, K. A. Boahen, P. O. Pouliquen, A. Pavasovic, R. E. Jenkins, and K. Strohbehn, "Current-Mode Subthreshold MOS Circuits for Analog VLSI Neural Systems," IEEE Transactions on Neural Networks, vol. 2, no. 2, March 1991.

E. An, An Improved Multi-Dimensional CMAC Neural Network: Receptive Field Function and Placement, Phd Dissertation, University of New Hampshire, 1991.

K. Asanovic, B. Kingsbury, J. Beck, N. Morgan, and J. Wawrzynek, SPerT: A Microcoded SIMD Array for Synthetic Perceptron Training, International Computer Science Institute, Technical Report, TR-91-072.

G. Barnes, R. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes, [1968]. "The ILLIAC IV Computer," IEEE Trans. Comput. 17, 746-757.

K. Batcher, [1976]. "STARAN Parallel Processor System Hardware," Proc. NCC 43, 405-410.

J. Beetem, M. Denneau, and D. Weingarten, [1987]. The GF11 Parallel Computer. in J. Dongarra, editor, Experimental Parallel Computing Architectures. North-Holland, 1987.

G. Belloch and C. R. Rosenberg, "Network learning on the connection machine," in Proc. IJCAI, Milano, Italy, Aug. 1987, pp. 323-326.

V. Benes, Optimal rearrangeable multistage connecting networks. Bell System Technical Journal, 43:1641-1656, 1964.

R. G. Benson and T. Delbruck, "Direction selective silicon retina that uses null inhibition," Advances in Neural Information Processing Systems 4, pp 756-763, Morgan Kaufmann, San Mateo, CA, 1991.

K. A. Boahen and A. G. Andreou, "A constrast sensitive silicon retine with reciprocal synapses," Advances in Neural Information Processing Systems 4, pp 764-772, Morgan Kaufmann, San Mateo, CA, 1991.

B. Bosner, E. Sackinger, J. Bromley, Y. LeCun, and L. D. Jackel, "An analog neural network processor with programmable topology," IEEE J. Solid-State Circuits, vol. 26, pp. 2017-2025, Dec. 1991.

D. S. Broomhead and D. Lowe, "Multivariable function interpolation and adaptive networks," Complex Systems, 2:321-355, 1988.

G. A. Carpenter and S. Grossberg, "The ART of adaptive pattern recognition by a self-organizing neural network," Computer 21 pp 77-88, 1988.

G. Cauwenberghs, C. F. Neugebauer, and A. Yariv, "Analysis and Verification of an Analog VLSI Incremental Outer-Product Learning System," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

D. Chin, et al., "The Princeton Engine: A real-time video system simulator," In IEEE Trans. on Consumer Electronics, volume 34, page 285-298, 1988.

T. Chiueh, "Recurrent Correlation Associative Memories", IEEE Transactions on Neural

Networks, vol. 2, no. 2, March 1991.

M. L. Chuang and A. M. Chiang, "Simulation of the Neocognitron on a CCD Parallel Processing Architecture," Advances in Neural Processing Systems 3, pp. 1039-1045, 1990.

DAP Series Technical Overview, Active memory Technology, Inc., Oct 1989.

J. B. Dennis, "Data Flow Supercomputers," Computer (18): 42-56, 1980.

S. P. DeWeerth, L. Nielsen, C. A. Mead, and K. J. Astrom, "A Simple Neuron Servo," IEEE Transactions on Neural Networks, vol. 2, no. 2, March 1991.

M. R. DeYong, R. L. Findley, and C. Fields, "The Design, Fabrication, and Test of a New VLSI Hybrid Analog-Digital Neural Processing Element," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

R. O. Duda and P. E. Hart, Pattern Classification and Scene Analysis. New York: Wiley, 1973.

D. A. Durfee and F. S. Shoucair, "Comparison of Floating Gate Neural Network Memory Cells in Standard VLSI CMOS Technology," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

A. Falkoff, "Algorithms for Parallel-Search Memories," J. Assoc. Comput. Mach. 9, 488-511, 1962.

W.-C. Fang, B. J. Sheu, O. T.-C. Chen, and J. Choi, "A VLSI Neural Processor for Image Data Compression Using Self-Organization Networks," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

W. A. Fisher, R. J. Fujimoto, and M. M. Okamura, "The Lockheed programmable analog neural network processor," in Proc. IJCNN (San Diego), 17-21 June 1990, pp. II563-II568.

W. A. Fisher, R. J. Fujimoto, and R. C. Smithson, "A Programmable Analog Neural Network Processor," IEEE Transactions on Neural Networks, vol. 2, no. 2, March 1991.

A. Flynn, and J. Harris, [1985]. "Recognition Algorithms for the Connection Machine," Proc. 1985 IJCAI, pp. 57-60.

C. Foster, [1976] Content Addressable Parallel Processors, Van Nostrand-Reinhold, New York.

T. J. Fountain, K. N. Matthews, and M. B. J. Duff, "The CLIP7A Image Processor," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 10, no. 3, May, 1988.

Y. Fujimoto and N. Fukuda, "An enhanced parallel toridal lattice architecture for large scale neural networks," in Proc. IJCNN, vol. II, Wahsington, DC, June 1989, pp. 614.

Y. Fujimoto, "An enhanced parallel planar lattice architecture for large scale neural network simulations," in Proc. IJCNN, vol. II, San Diego, CA, June 1990, pp. 581-586.

N. Fukuda, Y. Fujimoto, and T. Akabane, :A transputer implementation of toroidal lattice architecture for parallel neurocomputing," in Proc. IJCNN, vol. II, Jan 1990, pp.43-46.

H. Graf and D. Henderson, "A reconfigurable CMOS neural network," in Proc. 1990 Int. Solid State Circuits Conf.

H. Graf, R. Janow, D. Henderson, and R. Lee, "Reconfigurable Neural Net Chip with 32K Connections," Advances in Neural Information Processing Systems 3.

Kamil A. Grajski,"Neurocomputing using the MasPar MP-1", in Parallel Digital Implementations of Neural Networks, Prentice Hall, Englewood Cliffs, NJ, 1993.

A. Hamilton, A. F. Murray, D. J. Baxter, S. Churcher, H. M. Reekie, and L. Tarassenko, "Integrated Pulse Stream Neural Networks: Results, Issues, and Pointers," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

D. Hammerstrom, "A VLSI architecture for high-performance, low cost, on-chip learning," in IJCNN, pages II:537-544, 1990.

H. Harrer, J. A. Nossek, and R. Stelzl, "An Analog Implementation of Discrete-Time Cellular Neural Networks,"

R. Hect-Neilsen, Neurocomputing. Addison-Wesley, Reading MA, 1990.

W. D. Hillis, The Connection Machine. Massachusetts Institute of Technology Press, 1985.

Connection Machine CM-1 with 64k PEs is described.

W. D. Hillis, and G. L. Steele, Data parallel algorithms. Communications fo the ACM, 29(12):1170-1183, December 1986.

Y. Hirai. A model of human associative processor (hasp). IEEE Trans. on SMC, SMC-13(5):381-388, 1983.

A. Hiraiwa et al., "A two level pipe line RISC processor array for ANN," in Proc. IJCNN, vol. II, Washington, DC, 1990, pp. 137-140.

Y. Hirai, K. Kamada, M. Yamada, and M. Ooyama. A digital neurochip with unlimited connectability for large scale neural networks. In Proc. IJCNN 1989., volume 2, pages 163-169. Washington, D. C., 1989.

M. Holler, et al. An electrically trainable aritificial neural network (ETANN) with 1024 floating gate synapses. In Proc. IJCNN-89, pages 11-191, Washington, D. C., June 1989.

K. Hwang and F. A. Briggs, Computer Architecture and Parallel Processing. New York: McGraw-Hill, 1984.

Intel, 80170NX Electrically Trainable Analog Neural Network (ETANN) Manual, June, 1991, Intel Corporation, Santa Clara, CA.

Intel, Ni1000 Preliminary Specifications, Intel Corporation.

A. Johannet, L. Personnaz, G. Dreyfus, J. Gascuel, and M. Weinfeld, "Specification and Implementation of a Digital Hopfield-Type Associative Memory with On-Chip Training," in IEEE Trans. on Neural Networks, vol. 3., no. 4, July 1992.

H. Kato, et al., "A parallel neurocomputer architecture toward billion connection updates per second," in Proc. IJCNN, vol. 2, Washington, DC, 1990, pp. 47-50.

D. E. Knuth, The Art of Computer Programming, 3 vols., Addison-Wesley.

P. Kohn, J. Bilmes, N. Morgan, J. Beck, "Software for ANN training on a Ring Array Processor," Advances in Neural Information Processing 4, pp 781-788,

T. Kohonen, [1978]. Associative Memory, a System: Theoretical Approach, Springer-Verlag, Berlin. Kohonen, T., Self Organization and Associative Memory, third edition, Springer-Verlag, 1990.

T. Kohonen, Self Organization and Associative Memory. Springer-Verlag, Berlin, 1984.

Y. Kondo and Y. Sawada, "Functional Abilities of a Stochastic Logic Neural Network," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

A. Krikelis and M. Grozinger, "Implementing neural networks with the associative string processor," presented at the Int. Workshop for Artificial Intelligence and Neural Networks, Oxford, UK, 1990.

A. V. Krishnamoorthy, G. Yayla, and S. C. Esener, " Scalable Optoelectronic Neural System Using Free-Space Optical Interconnects," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

S. Y. Kung and J. N. Hwang, "Parallel architectures for artificial neural nets," in Proc. ICNN, vol. 2, San Diego, CA, July 1988, pp. 165-172.

S. Y. Kung and J. N. Hwang, "A unified systolic architecture for aritficial neural networks," in Journal of Parallel and Distributed computing 6, 358-387, 1989.

S. Y. Kung, VLSI Array Processors. Prentice Hall, Englewood Cliffs, NJ, 1988. Kung, S. Y., Digital Neurocomputing. Prentice Hall, Englewood Cliffs, NH, 1992.

J. P. Lazzaro, S. Ryckebusch, M. A. Mahowald, and C. Mead, "Winner-take-all networks of O(n) complexity," Advances in Neural Information Processing Systems 1, D. Tourestzky, Ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 703-711.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Handwritten digit recogntion with a back-propagation network," in D. Touretzky, Ed., pp 396-404, Advances in Neural Information Processing Systems 2, San Mateo, CA: Morgan Kaufmann, 1989.

B. W. Lee, J.-C. Lee, and B. J. Sheu. VLSI image processors using analog programmable synapses and neurons. In IJCNN, pages II:575-580, 1990.

R. R. Llinas, Ed. The Biology of the Brain, From Neurons to Networks, W. H. Freeman and Company, New York, 1988.

W. Liu, A. G. Andreou, and M. H. Goldstein, Jr., "Voiced-Speech Representation by an Analog Silicon Model of the Auditory Periphery," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

J. B. Lont and W. Guggenbuhl, "Analog CMOS Implementation of a Multilayer Peceptron with Nonlinear Synapses," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

T. Lu and F. Lin, "Experimental demonstration of large-scale holographic optical nerual network," International Joint Conference on Neural Networks, pp. I-535-540, 1991.

H. Mada, Architecture for optical computing using holographic associative memories. Applied Optics, 24: 1985.

L. W. Massengill and D. B. Mundie, "An Analog Neural Hardware Implementation Using Charge-Injection Multipliers and Neuron-Specific Gain Control," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

N. Mauduit, M. Duranton, J. Gobert, and J.-A. Sirat, "Lneuro 1.0: A Piece of Hardware LEGO for Building Neural Network Systems," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

J. L. McClelland and D. E. Rummelhard, editors, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, volume 1, pages 318-362, MIT Press, Cambridge, MA, 1986.

C. Mead, Analog VLSI and Neural Systems. Reading, MA: Adison-Wesley, 1989.

C. A. Mead, X. Arreguit, and J. Lazzaro, "Analog VLSI Model of Binaural Hearing," IEEE Transactions on Neural Networks, vol. 2, no. 2, March 1991.

R. W. Means and L. Lisenbee. Extensible linear floating point SIMD neurocomputer array processor. IJCNN, volume 1, pages 587-592. Seattle WA, 1991.

M. S. Melton, T. Phan, D. S. Reeves, D. E. Van den Bout, "The TInMANN VLSI Chip," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

M. S. Melton, T. Phan, D. S. Reeves, D. E. Van den Bout, "VLSI Implementation of TInMANN," in Advances in Neural Information Processing Systems 3, R. Lippman, J. Moody, and D. Touretzky, Eds. Los Altos, CA: Morgan Kaufmann, 1991, pp. 1046-1052.

W. T. Miller III, B. A. Box, E. C. Whitney, and J. M. Glynn, "Design and Implementation of a High Speed CMAC Neural Network Using Programmable CMOS Logic Cell Arrays," Advances in Neural Informations Systems 3, 1990.

W. T. Miller III, F. H. Glanz, L. G. Kraft, Application of a General Learning Algorithm to the Control of Robotic Manipulators Using a General Learning Algorithm, International

Journal of Robotics Research, Vol. 6, No. 2, p84-98, 1987.

W. T. Miller III, P. W. Latham II, S. M. Scalera, Bipedal Gait Adaptation for Walking with Dynamic Balance, ACC Boston, 1991.

W. T. Miller III and S. Scalera, private communication on MM32k benchmarks, Spring 1993.

M. Minsky, and S. Papert, Perceptrons, MIT Press, 1969, second edition 1972.

J. Moody and C. J. Darken, "Fast learning in netowkrs of locally tuned processing units," Neural Computation 1, pp 281-294, 1989.

J. Moody and C. J. Darken. Fast learning in networks of locally-tuned processing units. Neural Computation, 1(2), 1989.

G. Moon, M. E. Zaghoul, and R. W. Newcomb, " VLSI Implementation of Synaptic Weighting and Summing in Pulse Coded Neural-Type Cells," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

A. Moore, J. Allman, and R. M. Goodman, "A Real-Time Neural System for Color Constancy," IEEE Transactions on Neural Networks, vol. 2, no. 2, March 1991.

N. Morgan, The RAP: A ring array processor for layered network calculations. In Proc. Conference on Application Specific Array Processors, pages 296-308, Princeton, NJ, 1990.

P. Mueller, J. van der Spiegel, D. Blackman, T. Chiu, T. Clare, J. Dao, C. Donham, T. P. Hsieh, and M. Loinaz, "A Programmable Analog Neural Computer and Simulator," Advances in Neural Information Processing Systems 1, 712-719.

M. L. Mumford, D. K. Andes, and L. R. Kern, "The Mod 2 Neurocomputer System Design," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

A.F. Murray, D. Del Corso, and L. Tarassenko, "Pulse-Stream VLSI Nerual Networks Mixing Analog and Digital Techniques," IEEE Transactions on Neural Networks, vol. 2, no. 2, March 1991.

J. R. Nickolls, The design of the MasPar MP-1: A cost-effective massively parallel computer, pages 25-28. In Proc. COMPCON Spring '90, San Francisco, CA, 1990.

F. J. Nunez and J. A. B. Fortez, "Performance of connectionist learning algorithms on 2-D SIMD processor arrays," Neural Information Processing Systems 2, Denver, CO, 1989, pp. 810-817.

C. Park, K. Buckmann, J. Diamond, U. Santoni, S. The, M. Holler, M. Glier, C. L. Scofield, L. Nunez, "A RadialBasis Function Nerual Network with On-chip Learning," Intel Literature Packet received Sept. 1993.

J. Platt. A resource-allocating neural network for function interpolation. Advances in Neural Information Processing Systems 3, 1991.

D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky, and H. T. Kung. Neural network simulation at Warp speed: How we got 17 million connections per second. Proc. IEEE International conference on Neural Networks, pages II143-II150, 1988.

J. L. Potter, [1985]. The Massively Parallel Processor, MIT Press, Cambridge, MA.

J. L. Potter, Associative Computing, Plenum Press, New York, 1992.

K. W. Przytula, V. K. Prasanna, Parallel Digital Implementations of Neural Networks, Prentice Hall, Englewood Cliffs, New Jersey.

U. Ramacher, Ed., Microelectronics for Neural Networks. New York: Springer, 1991.

U. Ramacher, J. Beichter, and N. Bruls, "Architecture of a general-purpose neural signal processor," International Joint Conference on Neural Networks, 1991, pp. I-443-446.

U. Ramacher, W. Raab, J. Anlauf, U. Hachmann, J. Beichter, N. Bruls, M. Webeling, E. Sicheneder, R. Manner, J. Glab, and A. Wurz, "Multiprocessor and Memory Architecture of the Neuroncomputer SYNAPSE-1", World Congress on Neural Networks, pp. IV 775-78, 1993.

M. E. Robinson, H. Yoneda, and E. Sanchez-Sinencio, "A Modular CMOS Design of a Hamming Network," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

R. M. Russell, "The Cray-1 Computer System," Communications of the Association for

Computing Machines 21 (1): 63-72, 1978.

E. Sackinger, B. E. Boser, J. Bromley, Y. LeCun, and L. D. Jackel, "Application of the ANNA Neural Network Chip to High-Speed Character Recognition," IEEE Transactions on Neural Networks, vol. 3, no. 3, May 1992.

M. G. Sami, Ed., Silicon Architectures for Neural Nets. New York: Elsevier, 1991.

L. A. Schmitt, S. S. Wilson, "The AIS-5000 Parallel Processor," IEEE Transactions on Pattern Analysis And Machine Intelligence, vol. 10, no. 3, May 1988.

C. L. Scofield and D. L. Reilly, "Into Silicon: Real Time Learning in a High Density RBF Neural Network," in Proceedings IJCNN 1991, pp I-551-556.

C. Seitz, Concurrent VLSI architectures. Invited paper, IEEE Trans on Computer, C-33, December 1984.

D. A. Sholl, The Organization of the Cerebral Cortex, Methuen, 1956.

H. Siegel, L. Seigel, F. C. Kemmerer, P. T. Meuller, H. E. Smalley, and S. D. Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition, "IEEE Trans. Comput. C-30, 1981, 934-947.

A. Singer. Implementations of artificial neural networks on the Connection Machine. Parallel computing, 14:305-315, 1990.

S. Su, L. Nguyen, A Emam, and G. Ripovshi, "The Architectural Features and Implementation Techniques of the Multicell CASSM," IEEE Trans. Comput. C-28, 430-445, 1979.

S. W. Tsay and R. W. Newcomb, "VLSI Implementation of ART1 Memories," IEEE Transactions on Neural Networks, vol. 2, no. 2, March 1991.

B. Svensson and T. Nordstrom, "Execution of neural network algorithms on an array of bit-serial processors," presented at the 10th Int. Conf. Pattern Recognition, Computer Architectures for Vision and Pattern Recognition, vol. II, Atlantic City, NJ, 1990, pp. 501-505.

Texas Instruments, TMS320 Family Development Support Reference Guide, 1992.

K. S. Trivedi, "On the Use of Continued Fractions for Digital Computer Arithmetic," IEEE Transactions on Computers, July 1977, 700-704.

K. Wagner, and D. Psaltis, Multilayer optical learning networks, Applied Optics, 26:5061-5076, December 1987.

T. Watanabe, K. Kimura, M. Aoki, T. Sakata, and K. Ito, "A single 1.5-V digital chip for a 10^6 synapse neural network," IEEE Transactions on Neural Networks, Vol. 4, No. 3, May 1993, pp. 387-393.

T. Watanabe, Y. Sugiyama, T. Kondo, and Y. Kitayama, "Neural network simulation on a massively parallel cellular array processor: AAP-2," in Proc. IJCNN, vol. II, Washington, DC, June 1989, pp. 155-161.

J. Wawrzynek. A VLIW/SIMD Microprocessor for Artificial Neural Network Computations. Banff Workshop on Neural Networks hardware, March 1992.

C. Weiszmann, editor. DARPA Neural Network Study, October 1987-February 1988. AFCEA International Press, 1988.

S. S. Wilson, "Vector morphology and iconic neural networks," IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-19 no. 6, Nov/Dec 1989.

D. C. Wunsch II, T. P. Caudell, C. D. Capps, and R. A. Falk, "An optoelectronic adaptive resonance unit," in Proceedings IJCNN 1991, pp I-541-549.

M. Yasunaga, N. Masuda, M. Asai, M. Yamada, A. Masaki, and Y. Hirai. A wafer scale integration neural network utilizing completely digital circuits. In Proc. IJCNN 1989, volume 2, pages 213-217. Washington, D. C., 1989.

M. Yasunaga et al., "Design, fabrication and evaluation of a 5-inch wafer scale neural network LSI composed of 576 digital neurons," in Proc. IJCNN, vol. II, San Diego, CA, July, 1990, pp. 527-535.

X. Zhang, et al. An Effecient implementation of the backpropagation algorithm on the connection machine CM-2. In Neural Information Processing System, volume 2, pages 801-809, Denver, CO, 1989.