

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.



University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313.761-4700 800.521-0600

Order Number 9130347

**Low-cost, high-performance, microcomputer-based vector
processor in a network environment**

Mahmoud, Said Mohamed, Ph.D.

City University of New York, 1991

Copyright ©1991 by Mahmoud, Said Mohamed. All rights reserved.

U·M·I

**300 N. Zeeb Rd.
Ann Arbor, MI 48106**

A

**LOW-COST, HIGH-PERFORMANCE, MICROCOMPUTER-
BASED VECTOR PROCESSOR IN A
NETWORK ENVIRONMENT**

**By
SAID MAHMOUD**

**A dissertation submitted to the Graduate
Faculty in Computer Science in partial
fulfillment of the requirement for the
degree of Doctor of Philosophy, The
City University of New York.**

1991

©1991

Said Mahmoud

All Rights Reserved

APPROVAL STATEMENT

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

..4/29/91.....

Date

..Jacob Rootenberg.....TW

Chair of Examining Committee,

Dr. Jacob Rootenberg, Queens College

..4/29/91.....

Date

..T.C. Wesselkamper.....

Executive Officer

T. C. Wesselkamper

Dr. Michael Anshel, CCNY

Dr. Jerry Waxman, Queens College

Dr. Reda Ammar, University of Connecticut

Supervisory Committee

The City University of New York

ABSTRACT

LOW-COST, HIGH-PERFORMANCE, MICROCOMPUTER-BASED VECTOR PROCESSOR IN A NETWORK ENVIRONMENT

By

Said Mahmoud

Adviser: Professor Jacob Rootenberg

The main characteristics of a class of microcomputer-based vector architectures in an **Open Network Environment (ONE)** are defined. Employing this definition, specific systems are then generated, modeled, and evaluated. The generated systems range from Microprocessor-based derivatives to high-performance versions suitable for solving large scale scientific application problems. Each generated system consists of several identical and independent vector processing units. The external functional behavior of the generated system does not depend on the number of vector processing units employed and the system normally works properly with any number of vector units. The microprocessor derivatives are micro-programmable while hardwired structures are considered for the high performance versions to ensure high throughput. After the system generation and modeling process the actual simulation of the complete class takes place. An experimental simulation facility for the systematic evaluation of this class of architectures is also presented.

ACKNOWLEDGEMENT

The author is indebted to Professor J. Rootenberg for his supervision throughout the research which led to this thesis, and for his comments and criticism that were invaluable for improving this manuscript.

I also wish to thank Professor T. C. Wesselkamper for his invaluable suggestions and comments that were also invaluable for improving this manuscript.

Gratitude is expressed to Prof. M. Anshel, Prof. J. Waxman, and Prof R. Ammar for their continued interest in various aspects of the research. Gratitude is also expressed to Prof. Lincoln Anitie for helpful and fruitful discussions about vector processing simulation techniques. I also gratefully acknowledge Prof Samir Saddawi, who provided the initial impetus and the moral support for this work.

I also express my thanks for the support and forbearance of all members of my family throughout this research and to the Computer Science Program for providing the research environment.

Undertaking a project such as this places an equal if not greater burden on the people closest to me. My wife, Nabila, provided her understanding, but above all she provided her love.

This dissertation is dedicated
to my wife **Nabila** and our children

Pacent,

Nissreen, and

Nouran.

TABLE OF CONTENTS

1.	INTRODUCTION	1
2.	PARALLEL PROCESSING ARCHITECTURES AND PROGRAMMING	11
2.1	Classes of Parallel Computers and their Limitations	12
2.1.1	Classification of Parallel Processor Systems	13
2.1.2	Practical Supercomputer Architectures .	19
2.1.3	Parallel Computer Architectures	25
2.1.4	Dataflow Architectures	51
2.1.5	VLSI System Architectures	52
2.2	Programming Language Requirements	58
2.2.1	Language Extensions for Parallel Processing	59
2.2.2	Machine Dependent Parallel Languages .	61
2.2.3	Machine Independent Parallel Languages	61
3.	APPLICATION CHARACTERISTICS AND BENCHMARK DESIGN PROBLEMS	64
3.1	Large Scale Scientific Computation Application	65
3.1.1	Applications Characteristics and Computational Requirements	66
3.2	Exploitation of Parallelism in Numerical problems	69
3.2.1	Multi-Operation Functions	70

3.3	NetVecX Oriented Parallel Algorithms for	
	Performance Evaluation	77
3.3.1	Matrix Multiplication	79
3.3.2	Parallel Gaussian Elimination	
	Algorithm	80
3.3.3	Parallel Gauss-Seidel Iterative	
	Algorithm	84
3.4	Parallel Benchmark Design Proposal	86
3.4.1	Setting the Benchmark Design	
	Objectives	87
3.4.2	Factors that Influence Benchmark	
	Techniques	90
3.4.3	Parallel Computer Benchmark Technique	
	(PACBET)	93
3.4.4	Evaluation of the Technique	103
4.	NetVecX Architectural Specification	107
4.1	System Architectural Frame	108
4.1.1	Instructions and Addresses	112
4.1.2	Memory System Interleaving	114
4.1.3	Pipeline Concurrency Principle	116
4.2	NetVecX Architectural Derivatives	117
4.2.1	Microprocessor-Based NetVecX	
	(NetVecX/M)	118
4.2.2	G-Machine Based NetVecX Vector	
	Architecture (NetVecX/G)	121

4.2.3	High Performance NetVecX Vector Architecture (NetVecX/HP)	124
4.3	NetVecX MC68000 Microprocessor-Based Implementation	130
4.3.1	Principal Hardware Components	130
4.3.2	Interface Registers Definition	135
4.3.3	Vector Types and Structures	144
4.3.4	Vector Instructions	144
4.4	Performance Analysis	148
5.	<u>NetVecX Programming and Specification Language</u> (NPSL)	155
5.1	General Characteristics of Vector Languages	155
5.1.1	NPSL Extensions to Parallel Pascal .	160
5.2	Parallel Pascal Overview	161
5.3	Built-In Vector functions and Procedures .	175
6.	NetVecX Modeling And Performance Evaluation . .	176
6.1	NetVecX Modeling	174
6.1.1	Control Stream Simulation	174
6.1.2	Simulation Level and Model Resources	179
6.2	Control Stream Generation	180
6.2.1	NetVecX Control Stream Generation .	181
6.3	Modeling of the Central Control Unit Memory System	182
6.3.1	The Central Control Unit (CCU) . . .	182

6.3.2	The Memory System	185
6.3.3	The Vector Processing Units (VPUs) .	189
6.4	Control Streams Used in the Experiments . .	189
6.5	Simulator Implementation	190
6.6	Discussion of Experiments Performed and Results	200
6.6.1	Sequential Model	201
6.6.2	Bufferless-System Performance . . .	201
6.6.3	Buffered-System Performance	206
6.6.4	Effect of the Memory Subsystems on Performance	207
6.6.5	Number of VPUs Effect on Performance	209
6.6.6	Matrix Size and Vector Start-up Time Aspects	215
6.6.7	Basic System Profile	216
6.7	Limitations of the Model	216
6.8	Parallelism in the Programs	217
7.	CONCLUSIONS	219
8.	APPENDIX 1	225
9.	APPENDIX 2	228
10.	APPENDIX 3	232
11.	BIBLIOGRAPHY	236
12.	AUTOBIOGRAPHY	243

List of Tables

3.1	Multi-Operation Vector Functions	74
3.2	Some Primitives and Other Relative Information . .	75
6.1	A Sampling of CPU and Floating-Point Arithmetic Operations Time	193
6.2	Characteristics of Some Supercomputers	194
6.3	Memory Contention in a Typical System Configuration	205
6.4	Basic System Profile	215

List of Illustrations, Charts, Diagrams, etc.

2.1	Flynn's Classification of Architectures	15
2.2	A Four Segment Pipeline	24
2.3	SIMD Structure	27
2.4	MIMD Structure	28
2.5	CDC STAR-100 Structure	38
2.6	TI ASC Pipeline	41
2.7	CRAY X-MP CPU Structure	49
3.1	Timing Diagram for Recursive Vector Sum	71
3.2	Comparison Between Vector and Scaler Performance	73
3.3	Multi-Operation Vector Pipeline	78
3.4	Parallel Gaussian Elimination Algorithm	83
3.5	Program Segment of a Parallel Gauss-Seidel Algorithm	85
3.6	Benchmark Efficiency Factors	89
3.7	PACBET Technique VS Actual Program Execution	94
3.8	Flowchart For Primitive Assembly Production	100
3.9	Primitive Assembly and Compilation	102
3.10	Techniques Sensitivity	105
4.1	(a) NetVecX Block Diagram	109
4.1	(b) NetVecX Central Control Unit (CCU)	110
4.1	(c) NetVecX Vector Processing Unit (VPU)	110

4.2	MicroProcessor-Based NetVecX Organization	119
4.3	G-Machine-Based NetVecX Organization	122
4.4	Basic Conventional Computer Organization	123
4.5	High Performance Vector Processor Organization	125
4.6	MC6800 Scalar Instruction Format	132
4.7	Interface Registers Layout	137
4.8	ILLIFE Vector Storage Techniques	138
4.9	Free Vector Unit Selection	139
4.10	BUSY/FREE Bit Setting for VPUs	140
4.11	Status Register Interpretation	142
4.12	Vector Unit Number Requireing Service	143
4.13	Vector Types	146
4.14	Vector Instructions	146
4.15	Allocated Sub-Functions Within Groups	147
5.1	A Vector of Mapped 2-Dimensional Array: Row Storage	165
5.2	ILLIFE and Value Vectors of 2-Dimensional Sparse Array: Row Order	164
6.1	Possible Control Flow Paths in The Central Control Unit (CCU)	184
6.2	Possible Control Flow Paths in The S-Store Subsystem	186
6.3	Possible Control Flow Paths in The D-Store SubSystem	187
6.4	Possible Flow Paths in the	

Vector Processing Unit (VPU)	188
6.5 System Performance VS CPU Clock Period	195
6.6 System Performance VS Memory Cycle Time	196
6.7 System Performance VS Number of VPUs	197
6.8 System Performance VS Memory Cycle Time	198
6.9 System Performance VS CPU Clock Period	199
6.10 System Performance VS Matrix Size	212
6.11 Effect of Vector Start-Up Time on Performance	213

CHAPTER 1

INTRODUCTION

The **Open Network Environment (ONE)** is the interface to all the resources on a network. For the user, a set of services which makes remote files, special processors, or computing capabilities appear as if they are on the local computer system. One consistent interface provides access to these resources, regardless of their location.

With the ONE, software developers can write programs locally and run them on machines that offer the optimal performance and characteristics for the program. The programmer does not need to become an expert in networking or know the vendor-specific characteristics of the systems on the network. It also provides facilities for distributing tasks over multiple computer nodes.

The main theme of the **NetVecX** research project is the design and implementation of software and hardware for a special purpose computer system. This system is to be integrated into an Open Network. A network of this kind might be appropriate for the needs of a large university, a company, or a research laboratory.

It is expected that the network will contain a variety of machines with characteristics which relate to their function. They may be classified as follows:-

- (1) Personal WorkStations (WS machines),
- (2) General Purpose Computers (GP machines),
- (3) Specialized File Servers and High Performance Computers (HP machines).

The personal workstations will probably employ standard 16-bit microprocessors. They will have networking and graphics capabilities and an operating system with several high level languages. With these facilities provided the personal workstations serve as local computing facilities located in laboratories and offices where they will be used mainly for running application programs while large or more complex programs will be sent to more appropriate machine in the network. In this way, they will be used as front-end processors to the machines in the network.

The medium size general purpose computers are normally a low cost relatively high performance computers. The machine will complement the WS machines as general purpose timesharing machines, and may provide back up processing power for running medium size jobs. These machines would be located in Departments or Branch Offices of a company.

In addition to the general purpose machines there would

be special machines for file handling, and high performance machines for executing large programs. These large programs are exemplified by number-crunching application problems for which the atmospheric weather problem is used to illustrate their general characteristics. The appropriate site for these machines will be in Regional centers and Head Offices. It is hoped that the power of these machines will be in excess of the power of the CRAY-1 [89], the CYBER-205 [43], or the CRAY X-MP [18].

In the light of the above high performance requirements, alternative approaches to the design of high performance machine architectures have been under investigation and the work described in this thesis relates to the design and evaluation of a class of high performance vector computer architectures.

It is very instructive at this point, to trace briefly the history of computer "power" since the early 1950's. The reason being that, in the past, technological progress has tended to dictate the increase in performance of computers but in recent years a number of physical constraints limit the performance of these computers. These constraints are discussed below and form the basis for alternative organizational improvements which introduce parallelism into computer architecture in order to increase performance.

The high demand for more computing power has led to a significant improvements in logic and memory technology. This, together with new circuit techniques, has resulted in the improvement in computing capability in both performance and complexity. This improvement in computing performance is , roughly speaking, the speed and thus the computing "power" increased in multiples of five every five years, starting with the Manchester MARK 1 in the 1950's and continuing until 1970's. From then on this trend diminished and a "power gap" emerged as a limiting factor on the increase in performance of computers through technology alone. These constraints include the speed of light for the propagation of a signal through an electrical conductor, the switching time of components and the dissipation of heat. Although, since 1960, significant development in technology, particularly Very Large Scale Integrated circuits (VLSI) have helped to reduce the number of components by up to hundreds of thousands, the stage has now been reached where increased performance from technology alone is limited by basic physical laws [50]. Therefore, alternative organizational improvements have become increasingly important in the search for higher performance in computer systems. The following paragraphs briefly discuss some of these alternative approaches.

The search for higher levels of computer performance, apart from technology alone, has inevitably led to the

introduction of parallelism into computer architecture to enhance performance of computations. For example, the concept of functional parallelism as depicted by the CDC-6600 [95] and pipelining techniques which have also produced further speed improvements as exemplified by the CDC-7600 [27], which eventually resulted in the design of the CRAY-1 [17,89], and the MU5 instruction pipeline [75] give credence to this assertion. Another divergent view which does not use the conventional control flow concept of machine design is based on the data-flow concept. Data-flow uses a finite directed graph to describe a computation. The arcs of the graph are of data directed from one node to another. The nodes represent single-valued functions which map input data onto output data. There are a few systems which form the hub of academic research into data-driven computation and are being developed for general purpose scientific applications [21]. This illustrates a radically different approach to computer design from the conventional control flow concept.

Another approach is to exploit the natural parallelism that exists in certain problems [97]; for example, image processing, weather forecasting and military applications. In doing this, parallel computation partly overcomes the inherent speed limitations of sequential computer systems, in principle, by allowing simultaneous operations to take place within the machine without affecting the validity of the final

result. This can only be made possible by the introduction of special architectural features which will allow the simultaneous operations to take place. Thus parallel and vector computers utilize their system resources in a better way and hence give increased performance. The incompatibility of the parallel and vector machines with the high level programming languages used in programming them also tend to reduce the expected increase in performance. therefore the design of high performance machines in an Open Network has followed the concurrent design of the software and hardware for this class of machines. The results of a comparative study of the two divergent approaches to the design of high performance computers, suggested that alternative high performance machines might be designed by combining some of the attractive characteristics of each parallel and vector architectures. The reason why an alternative approach is taken and the justification for designing a compatible architecture and language necessary for effective parallel processing in the network are discussed in **Chapter 2**.

The research described in this thesis is but a very modest part of the long term goal to advance the design of high performance computer systems. The aims of this research are as follows:-

- (1) to study the design of a standardized

- benchmark for performance evaluation of vector and parallel computers;
- (2) to investigate the design of a class of vector computer architectures suitable for use as a high performance machine in an Open Network Environment;
 - (3) to model and design an experimental system for evaluating the design in(2)

The methodology followed in pursuing (1) was to study the numerical algorithms underlying the major areas of application with a view to identifying the relevant characteristic which affect benchmark construction. This fundamental approach was designed to stimulate and generate ideas to aid the design of a benchmark. It also reveals some pertinent characteristics of the applications for which software may be used to map the problems into sets of parallel vector operations. This effectively leads to the realization of parts of objective (2). The general characteristics of the application problems and the benchmark design proposal are discussed in **Chapter 3**.

Another feature of the research involves the survey of parallel and vector architectures in order to identify factors which degrade their performance. These findings substantially provide ideas for the proposal of the alternative architecture within the class of high performance machines envisaged for

the network. **Chapter 2** surveys parallel and vector architectures, and discusses possible reasons for the relatively ineffective exploitation of the architectural features of these machines in solving large scientific application problems. The survey partly led to the fulfillment of objective (2) above. In pursuing objective (3), it became natural to develop a high level programming language by extending Parallel Pascal(PP) to exploit the vector processing features of the proposed **NetVecX** architecture. The features of this language, as discussed in **Chapter 5**, include formal explicit mechanisms for defining parallel computation which the architecture would implement by hardware. The language illustrates the flexibility of exploiting parallelism in this class of high performance machines. This contributes to the realization of objective (3) above. In pursuing this objective further, a simplified derivative of the proposed vector architecture is modelled for evaluating the class of vector architectures under investigation. The work involved in evaluating the basic **NetVecX** architecture consists of (1) designing a processor for simulating the vector instructions that were required in testing the application problems used in the performance evaluation exercise for validity and accuracy. In so doing, the program execution trace enabled procedural and data dependency information of typical workloads to be gathered as the programs were executed successfully; (2) having designed

and constructed the simulator, it then uses the data generated from the preprocessor to simulate those aspects of the system needed to evaluate its performance. A number of experiments were carried out by varying the system parameters to generate different **NetVecX** architectural configurations using identical workloads to study the effect of different architectural features on the performance of the **NetVecX** computer system. **Chapter 6** describes this model and the performance evaluation by simulation.

The detailed description of the proposed vector architecture is presented in **Chapter 4**, but, a brief description and its mode of operation is given here. The study of application problems' particularly those requiring high performance computers [90], shows that many involve the processing of arrays in which performance benefits are frequently to be obtained from having a single machine instruction process the whole of a one-dimensional array (or vector). Furthermore, it has been realized that a number of vector processing units can be conveniently controlled by a single, relatively conventional, scalar processor which uses software to map multi-dimensional array operations into a set of (frequently) parallel vector operations. Therefore, a class of high performance machines (**NetVecX/HP** type) envisaged for an Open Computer Network has been proposed based on the architecture involving a vector processor incorporating these

ideas.

In order to study the interactions of the various components of the **NetVecX** architecture, a functional level deterministic simulator was designed and implemented. It was exercised, as mentioned above, by two sample programs in a number of experiments for performance evaluation. The compatibility of the high level language features with the **NetVecX** architecture was also studied. **Chapter 6**, therefore, discusses the results of these experiments.

Finally, in **Chapter 7**, the findings of the research project are discussed in the light of the experimental results obtained from the simulation studies. The conclusion discusses whether the performance estimates for the architecture are convincing enough to be carried over to the eventual design of higher performance derivatives of this vector architecture.

CHAPTER 2

PARALLEL PROCESSING ARCHITECTURES AND PROGRAMMING

This chapter provides a brief description of major trends in parallel and vector computer architectures. Major trends which have influenced computer design and architecture are specifically highlighted. Availability of low-cost hardware and reduced communication costs have led to network-based architectures and distributed processing systems

This chapter also provides an architecture classification scheme. It should be noted how practical real life machines do not necessarily fall into one of these classifications, but usually span several classifications according to the mode of operation. Some examples of supercomputers are also provided. More specifically we survey current vector and parallel computer architectures and the high level programming languages used in programming them. The characteristics of the two different architectures are then highlighted to form the basis for the proposal of the class of high performance vector machines envisaged for an Open Network Environment.

2.1 Classes of Parallel Computers and their Limitations

Parallelism has appeared in various forms in many

computers developed to date and has proved to be a very effective approach in the way these computers achieve increase in performance over their sequential counterparts [2,30,43,45,60,85]. The styles of the various classes of parallel computers are determined by the way these computers are organized to achieve this increase in performance. Broadly speaking, this increase in computational speed is obtained by overlapping or reducing the number of operations performed by any machine. These operations may be broadly classified as (i) arithmetic and logical operations, (ii) organizational operations (iii) address generation, and (iv) memory accesses. An indication of the benefit of overlapping and reducing the four groups is exemplified by two divergent methods for organizing parallel computers from the choice of operational characteristics.

Of the currently implemented high performance computers, the broad classes of architectures which can be identified are, (i) Vector computers for which examples are CDC STAR-100 [66], the CYBER 205 [43], TI-ASC [99], and the CRAY-1 [17,19,89], (ii) Parallel computers of which two main types are illustrated by the array processors exemplified by the ILLIAC-IV [6,61] and the ICL-DAP [86] while the other parallel class represents the class of multiprocessors, for example, the C.mmp [105]. The general characteristics and shortcomings of these machines are discussed in the following subsections.

2.1.1 Classification of Parallel Processor Systems

Numerous techniques have been proposed for an orderly, workable scheme of classification of parallel architectures as a result of the proliferation of processors with different disparities. Broadly speaking, there are three main kinds of classifications which are organized to achieve some form of parallel operation.

Of all the proposed classification schemes, Flynn's macroscopic taxonomy [30] is the most widely used. In this scheme, computer systems are classified according to the number of data or instruction streams that can be processed in parallel. Four broad classifications emerge, according to whether the instruction or data streams are single or multiple :-

- (1) SISD- Single Instruction stream/Single Data stream - which is the philosophy behind the conventional von Neumann computer and it is irrelevant whether pipelining is used to speed up the processing of instructions or the arithmetic.
- (2) SIMD - Single Instruction stream/Multiple Data Stream - is characterized by a computer that retains a single stream of instructions but has vector instructions that initiate many operations. This classification includes all machines with vector instructions. Again, it

is irrelevant whether the capability of vector processing is realized by pipelining or building arrays of processors.

- (3) MISD - Multiple Instruction stream/Single Data stream - is characterized by specialized streaming organizations using multiple instruction streams on a single sequence of data.
- (4) MIMD - Multiple Instruction stream - is characterized by several instruction processing units and therefore uses several data streams.

The argument against this taxonomy, **Figure 2.1**, is that it is broad, ambiguous and confusing without drawing any distinction between the pipeline computer and the processor array which have entirely different computers architectures. Even though Flynn was clear that the pipeline vector computer falls into the SIMD class, the above confusions have led many to think of other forms of classifications based on different criteria other than data and instruction streams.

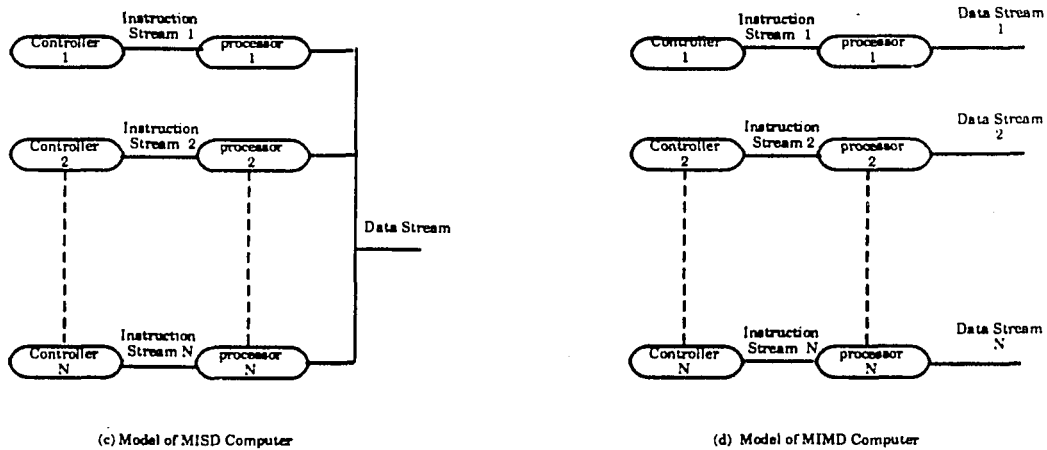
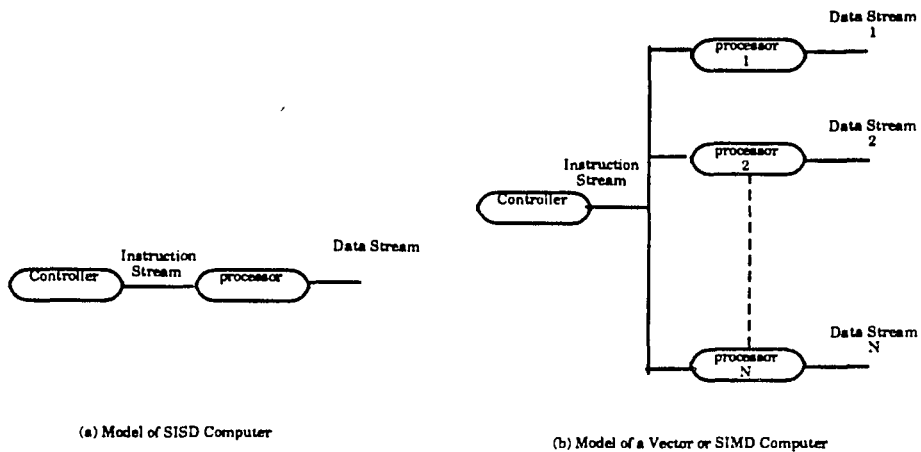


Figure 2.1 Flynn's Classification of Architecture

One other classification by Shore [91] is based on six different types of machines and distinguishes them by a numerical designator with no mnemonic or ordinal significance. The broad definitions pertaining to the Shore taxonomy are :-

- (1) Word serial, bit parallel (Type I)
- (2) Word parallel, bit serial (Type II)
- (3) Orthogonal computer (Type III)
- (4) Unconnected array (Type IV)
- (5) Connected array (Type V)
- (6) Logic-in-memory array (Type VI)

This classification also basically fails to place the pipeline vector computer in a class of its own. Thus, it can be seen that Shore's machines II to V are useful subdivisions of Flynn's SIMD class, and that machine I corresponds to the SISD class. This classification is by no means satisfactory and other attempts have been made at other classifications to incorporate pipeline designs. This is exemplified by the complex structural taxonomy by Hockney and Jesshope [43,47], who based their classifications on a detail machine architecture, interconnection and control. This classification, even though useful, is rather complicated and uses a notation akin to a chemical formula.

The taxonomy of computer structures, as seen above, has

inadvertently failed to uniquely classify pipeline vector computers and has unwittingly placed parallelism and pipelining at different logical levels i.e. program level, instruction level and word level.

By this taxonomy, the classification scheme mainly has to fulfil the following requirements :-

- (1) It must classify all existing as well as all foreseeable computer structures;
- (2) It must differentiate essential processing structures;
- (3) Any subject or computer structures must be assigned a unique classification.

It is clear that some of the existing taxonomies seem to violate just these rules. They present only rough categories to the exclusion of viable structures and are not able to classify any kind of pipelining.

The Erlangen Classification System (ECS) [39] which largely avoids these disadvantages is based on a lucid but rigid triplet as a characterization for basic structures. It introduces operations $+$, $*$, and v to facilitate flexibility. Thus an Erlangen triplet is defined as:

$$t(c) = (k, d, w)$$

where

k is the number of Control Units (CUs) interpreting a program;
 d is the number of Arithmetic and Logic Units (ALUs) controlled by one of the k Control Units;
 W is the word length or number of bits handled in one of the d ALUs.

This characteristic can be seen as a point in a three dimensional space which leads to more complicated structures since most control elements do not permit serialization without bringing about an additional complexity. By considering all kinds of pipelining, composite structures are defined using, as ingredients, a basic triplet or structure.

An elementary parallel - pipelining characteristics is an extended triplet exemplified by :-

$$t(c) = (k * k', d * d', w * w')$$

where k, d, w are as defined above, and k' is the number of Control Units (CUs) interpreting tasks of a program, whereby the data flow through these units (processor) is sequential : macro pipeline,

d' is the number of functional units (ALUs) controlled by one data stream :
 Instruction pipeline

w' is the number of levels or phases in an

arithmetic pipeline.

With this classification, the CDC STAR-100 is classified as

$$t_{(\text{CDC STAR})} = (1, 2, 64 * 4)$$

The notation 64*4 in $t_{(\text{CDC STAR})}$ means an arithmetic pipeline of four levels, working on 64-bit information. In the second position of the triplet, the 2 signifies that two such pipelines are allocated. Representing the NetVecX under this scheme, the notation NetVecX would be like this:-

$$t_{(\text{NetVecX})} = (1, 16*4, 64*8)$$

The 16*4 in $t_{(\text{NetVecX})}$ means sixteen vector processing units each is having four arithmetic pipelines while the 64*8 represents an arithmetic pipeline with 8 segments working on 64-bit information.

With the given triplets all elementary structures can be characterized according to its proponents; and because the three parts of a triplet refer to a specific homogeneous structure one has now to introduce composition rules to make possible the characterization of more complex structures.

2.1.2 Practical Supercomputer Architectures

This section briefly examines the configurations of existing supercomputers that introduce parallelism and pipelining into their architectures. In these computers the segmentation of a computation into several sub-process are executed by dedicated autonomous functional units. The

technique employed in executing these processes at speeds matching the store access rates are discussed whilst some of their advantages and disadvantages are also analyzed.

2.1.2.1 Vector Computer Architecture Features

Enhancing Performance

The Basic architecture features that largely enhance the performance of vector architectures are summarized below and are relevant in considering the design of alternative high performance machine for an Open Network Environment.

- (1) Very Large high speed central memory.

It is important for any high performance machine to employ the maximum amount of memory permitted by existing technology and engineering judgment at the time of design.

- (2) High speed memory-to-memory vector operations.

A vector as defined in the vector processing terminology is a contiguously stored string of data. The requirement for contiguous storage of data in memory arises from the engineering solution of achieving high bandwidth interleaved memory banks. In performing vector operations in the STAR-100, for instance, data in vector form is fed from

central memory to the arithmetic units and the results are returned to memory upon completion of the operation. Hence the term "memory-to-memory". The TI-ASC uses a variant of this system which creates vectors out of noncontiguously stored data with a consequent degradation in machine performance. The CYBER-205 employs its string unit which is fed streams of data by the stream unit to achieve its memory-to memory operations. The CRAY-1 provides arithmetic functions similar to the STAR-100, CYBER-205, and the TI-ASC, but its vectors are read from a set of high speed registers, which themselves must be loaded and unloaded from central memory at different points in the processing. Since the data arriving at these units is usually contiguous, every segment in a pipeline is performing useful work except at the very beginning and the end of a vector operation. Nevertheless, it increases the performance of these vector machines.

(3) Vast Virtual Memory Space.

This is used as a tool to assist in memory management where the operating system ensures

the contiguity of a user's work space. It also eliminates working space constraints which can inhibit algorithm development and provides the notion of an infinite work space for data and temporary results.

- (4) Overlapping of Input-Output and Arithmetic operations.

This is necessary so that the effective speed of the machine is not reduced drastically below its theoretical capacity, **Figure 2.2**.

- (5) Multi-operation functions.

These are used in addition to the standard arithmetic operations to increase the number of vector operations. Having adopted most of these functions from APL [46], they have been found to be very effective in providing data "mapping" or restructuring to produce vectors from otherwise non-vector constructs.

It is very clear that vector computers obtain their increase in performance through the use of pipelining techniques and vectors instructions that apply a single function, say, vector to vector multiplication as basis units of information. In doing so, they substantially reduce the

number of instructions for organizational and address generation operations and allow data look-ahead. However, most of these vector processors are limited to the predominantly serial processing of a single instruction stream. The highly serial nature of their class of problems and the lack of hardware facilities for sparse processing are some of the factors which degrade their performance capabilities. All these limitations contribute to reducing their potential increase in performance. Since a large class of problems do not appear to be well suited to this architecture, the central issues behind the development of any vector architecture must deal with the efficient execution of specific problems.

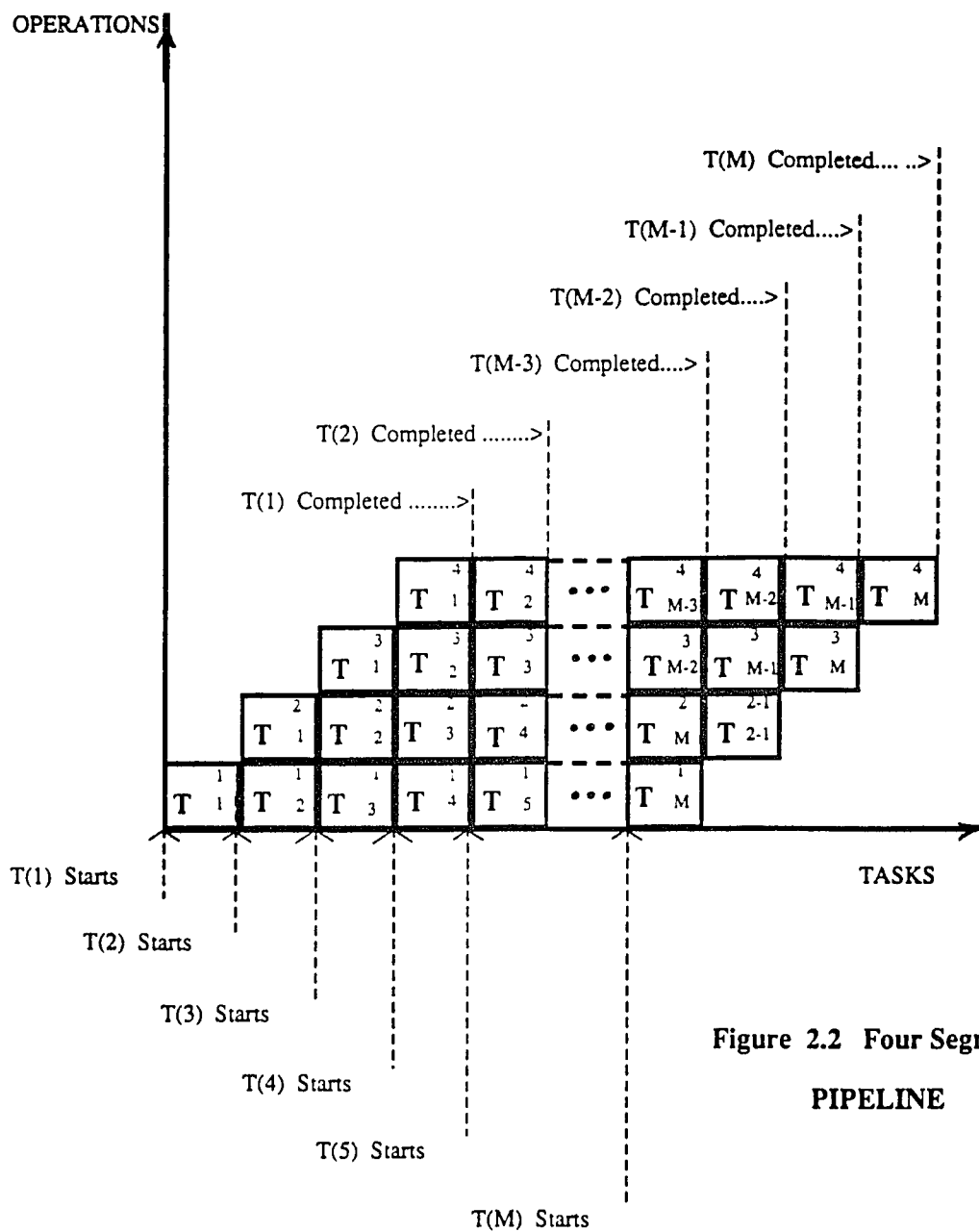


**Figure 2.2 Four Segments
PIPELINE**

2.1.3 Parallel Computer Architectures.

Parallel computers obtain their increase in performance through the use of a number of usually identical processing units that allow computations to be performed in parallel. Two sub-types are recognized. These are classified as the Single - Instruction - Multiple - Data stream (SIMD), Figure 2.3 using Flynn's terminology, and often called array processors, and the Multiple - Instruction - Multiple - Data stream (MIMD) computers, Figure 2.4. The following subsections discuss the strengths and weaknesses of these class of parallel architectures.

2.1.3.1 Array Processor Architectures

SIMD (array) computers, exemplified by the ILLIAC-IV, and the AP-120B [28,31,43,48] use a single control unit to drive a number of synchronous processing elements that perform the same function simultaneously on a number of data streams in lock-step mode. The main advantage in array computers is that a high degree of parallelism is introduced into its computational operations. Array computers, however suffer from the lack of effective data access and communication capabilities to facilitate effective parallel processing in order to make the computer system useful for a large variety of computations. Apart from this, array computers reduce their potential increase in performance when the vector length

exceeds the number of processing units [22,44]. It therefore exposes the fact that for effective and efficient processing on array processors, the problem must match the size of the array processor. This situation does not always arise as some of the application problems show in **Chapter 3**.

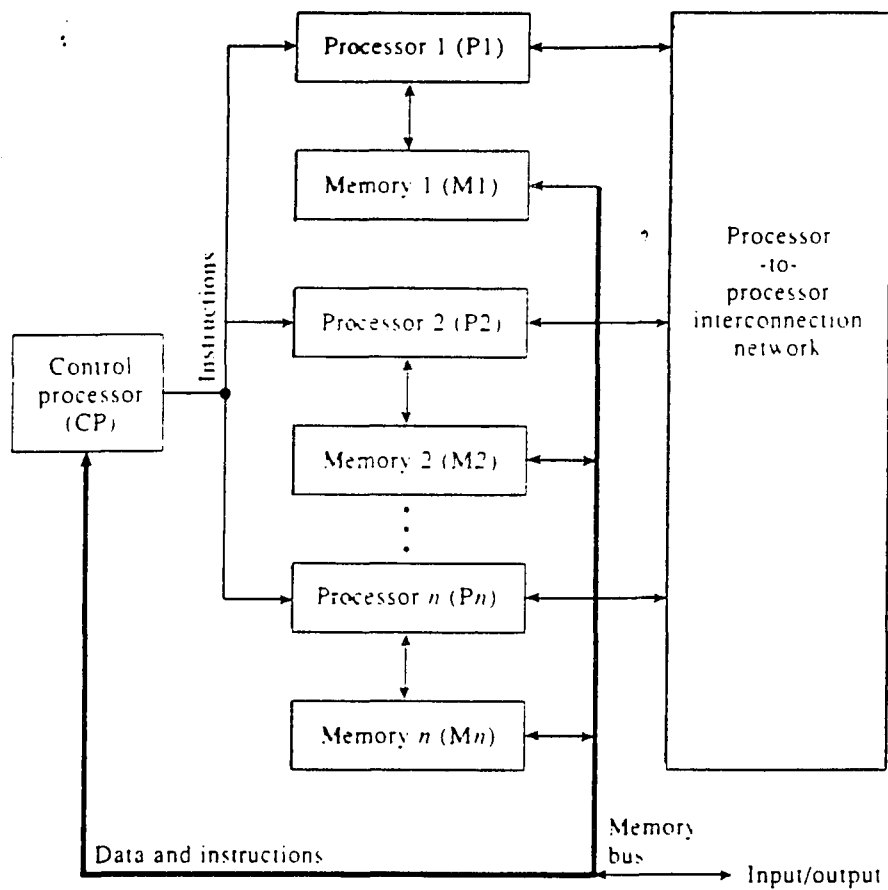


Figure 2.3 SIMD Structure

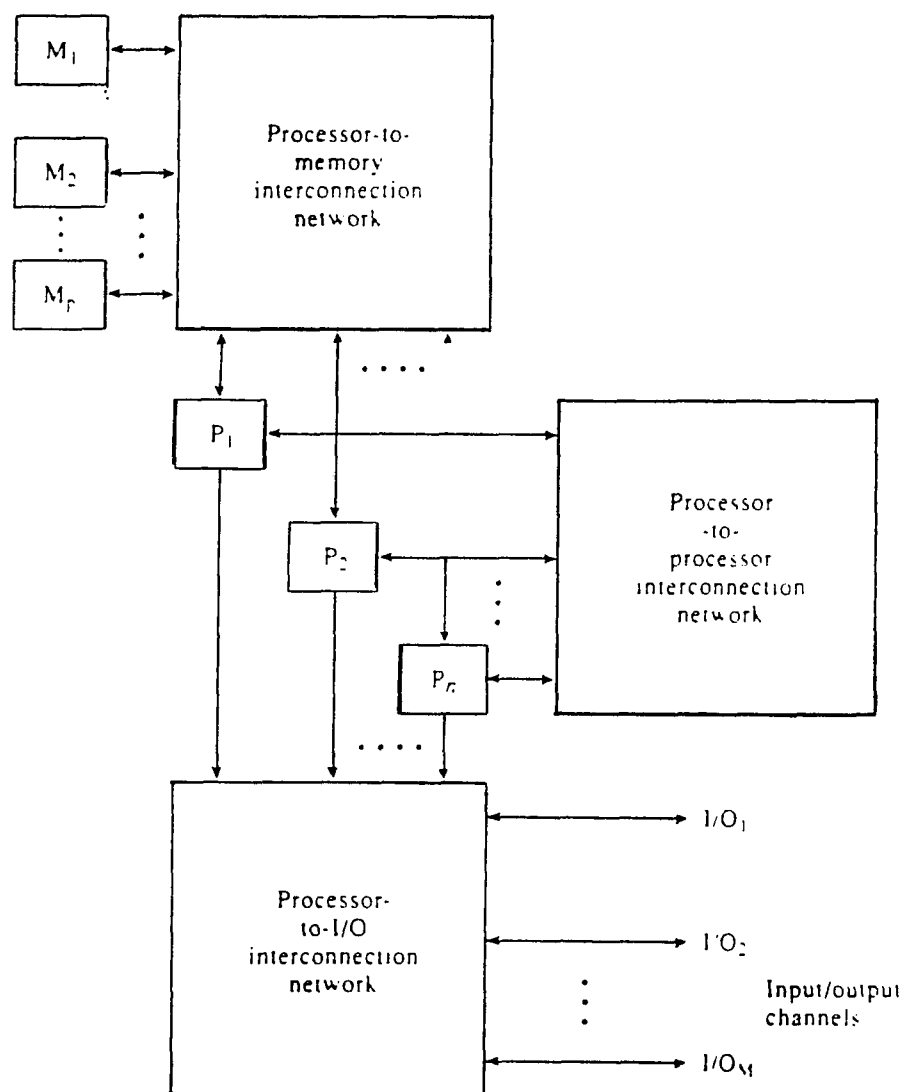


Figure 2.4 MIMD Structure

Unlike the ILLIAC-IV, the ICL-DAP has been integrated into a complete computer system and has been designed not only to emulate a memory module of an ICL mainframe computer, but also to process data autonomously in a highly parallel manner. Thus, the processing power is distributed throughout the memory of a conventional computer. In order to communicate with adjacent processing elements (P.Es), the DAP consists of bit-serial processing elements, all arithmetic is built up from software which implies that there will be strong performance dependency on the word length of the representation of the data. This inevitably leads to a degradation in DAP performance. Array computers also suffer from similar limitations as vector computers as discussed above. The NetVecX architecture is aimed at processing any length of a vector and does not necessarily rely on the size of the problem matching the number of vector processing units for increase in performance.

The array processor is increasingly employed in application areas where it is suitable for processing synchronously large amount of data with no inter-relationships [22,43]. The parallel capability of this class of computers depends on the number of processing elements and their application is limited since they are not suitable for certain vector oriented tasks, recurrence related, accumulative and table look-up operations. The array processor is also

unsuitable for executing components that are separated by non-unit increments because of the expensive cost of routing data [22,43]. Finally, it is difficult to program an array processor efficiently without being aware of the interconnectivity of the processors, whether orthogonal, rectangular or linear.

The philosophy underlying the design of array processors has been the relaxation of the restriction imposed on the processors to operate in lock-step mode. The processors are then given a block of code to execute in parallel. A typical example of such a machine is the Burroughs design for the NASF flow processor [22,43]. Their arithmetic processing units perform identical operations on a different parts of a homogeneous data structure, although not in a strict lock-step fashion.

The main efficiency problem for array processors is the maximal utilization of the processing elements [22,43]. With a highly branching program only a fraction of the processing elements will be active at any one time. This has led to writing different algorithms with less branching.

2.1.3.2 Multiprocessor Architectures

The two major classes of parallel computers are vastly different in how they attain parallelism of operation. As we

observed above, the array computer is well suited to computations that can be broken up into a sequence of vector operations. But a problem with no particular vector structure or natural iterative structure may have a great deal of potential parallelism that can be exploited by an MIMD computer, for example, the C.mmp. [105], because the MIMD computer is not constrained to a repertoire of vector instructions. Parallelism for this type of computer generally consists of doing independent tasks in parallel on independent computers. The results of the execution of the independent tasks are then combined. Thus, MIMD computers have the maximum potential for exploiting parallelism because each independent computer could be designed as a vector or array machine [18], for example, performance estimates for the NASF research machine has been conservatively put at least 1000 Mega Flops [43,105].

But one major weakness of most MIMD and other parallel computers is that they fail to exploit the maximum parallelism in a problem. For this we offer two main reasons: firstly; parallel computers are programmed in languages which have basically serial structures and by programmers who think serially; secondly, they most frequently lack equivalent hardware structures to support the mechanisms used in the language to represent parallelism.

While parallelism and pipelining are effective ways of improving system speed for a given technology, they have shortcomings. Some pipelines perform better, the longer the vectors are that they have to process whilst the performance of other pipelines depends on vector lengths matching high-speed register set lines. Parallel systems perform best on vectors whose length is a multiple of the number of processors available. Either type of system performs adequately if vector sizes are very large relative to the machine, but as these systems are used in wider application areas, short vector performance becomes more important. Other limitations of most pipeline processors have that, the arithmetic operations to be pipelined can reasonably be broken into only limited number of segments and that overlapping of several instructions in one pipeline leads to unreasonable control problems as in the TI-ASC.

Usually, there is no master-slave relationship between the processors and their interconnection network which would permit any processor to access any one of the system's memories. The time taken for all memory accesses may be the same as in the Carnegie-Mellon C.mmp which uses a cross-point switch: or different as in the upgraded version of the C.mmp, the Cm* which uses an hierarchical, distributed-bus-oriented interconnection scheme.

2.1.3.2.1 A Multiport Multiprocessor Architecture

The search for increased computer power has resulted in the design of a pyramid-like multi-processing system at the University of Erlangen-Nuremberg [39]. We now discuss the main architectural features of the Erlangen General Purpose Array project (EGPA). The pilot implementation consists of five processors, five memory units and five multiports which form an absolute minimum configuration.

This configuration employs a novel technique called 'Vertical processing', for greater efficiency since it allows the handling of an arbitrary word length while conventional or 'Horizontal processing' is essentially bound by the machine word [43].

There are some striking advantages with this mode of processing. A factor of improvement between three and fifty can be gained by 'Vertical Processing' according to the designers of the EGPA [9]. Most common applications, like the solution of partial differential equations by relaxation methods may be based on regular array, taking one processor for example to perform the iteration steps for one grid point. Boundary conditions are also taken into consideration, disabling processors on the boundary and introducing, instead, constant boundary values.

Decisive for the connection of the EGPA units is the use of the multiport, which is an integral part of an EGPA unit. Cross-connections make it possible either to change the workload or to dynamically "repair" a configuration under operation. Thus, each processor has access to its immediate neighbors in northern, eastern, southern, and western directions. Regarding the fact that a processor is connected to its "own" associated memory block, this topology results in a connection scheme where each processor has access to five memory blocks and each memory block is connected to five processors of the same layer. The EGPA is presented as a flexible structure, in the sense that this flexibility implies the multiple utilization of hardware in the form of different operation modes or different kinds of interpretation.

The main limitations of multiprocessors are the high cost of data communication and synchronization, which makes them unsuitable for large vector oriented tasks. This assertion is supported by Baudet [9]. In this work the iterative solution of the Laplace equation using an MIMD type parallel computer, the C.mmp is considered. Baudet showed that the usual solution methods of Gauss-Jacobi and Gauss-Seidel, involve a level of synchronization between the processors that significantly reduces the parallel performance. Using asynchronous iterative schemes, it was demonstrated theoretically that they perform better on the C.mmp than the

aforementioned synchronous methods. However, in running the synchronized algorithms on a similarly structured two processor parallel computer [5], a small performance loss was encountered due to synchronization overhead. The difference is attributed to a faster synchronization tool on the Loughborough computer than on the C.mmp machine. The Neptune system at Loughborough is a four processor parallel computer system with an even smaller performance loss due to synchronization overhead [4].

The above difference shows that there is a need to characterize overhead in parallel programs in a system-independent manner. In this way an algorithm can be matched to the machine, and thereby also avoid the mistake of rejecting an algorithm merely because it performs badly on one particular computer system.

From the onset of this research project, it has been borne in mind to design a system with high performance and reliability that is practical to implement. It was decided to investigate the suitability and practicality of designing a class of vector architectures which combine some of the attractive characteristics of vector and array architectures in a most cost-effective way.

2.1.3.3 Vector Supercomputers

This section briefly discusses the organization of first and second generation vector supercomputers that introduce parallelism into their architectures by pipelining the arithmetic units, or by replication of arithmetic units with pipeline instruction processing. These processor architectures are of particular relevance because of their architectural innovations in achieving high performance. The techniques for improving performance are analyzed and discussed vis-a-vis their strengths and weaknesses.

2.1.3.3.1 The CDC STAR-100

The CDC STAR-100 [66] was conceived as a processor for vectors with an instruction set based on Iverson's APL language [46] that would, with pipelining be able to sustain rates of 100 Mflops on long vectors that were common to many scientific problems. Many of the poor features of the machine (e.g. slow 1 Micro Sec. core memory) originated from its early start.

The logical layout of the machine is shown in **Figure 2.5**. There are two floating-point pipes connected to memory through a stream unit and a storage access control. All vector operations take two data streams from main memory and return a third data stream to memory. There are no vector registers. The stream and the storage access units have the job of organizing and smoothing the flow of the data. When

operating smoothly each pipe can generate one 64-bit or two 32-bit floating-point results per clock. Since the clock period is 40 nanoseconds (ns), The maximum rate is one 32-bit per 10 ns or 100 Mflops.

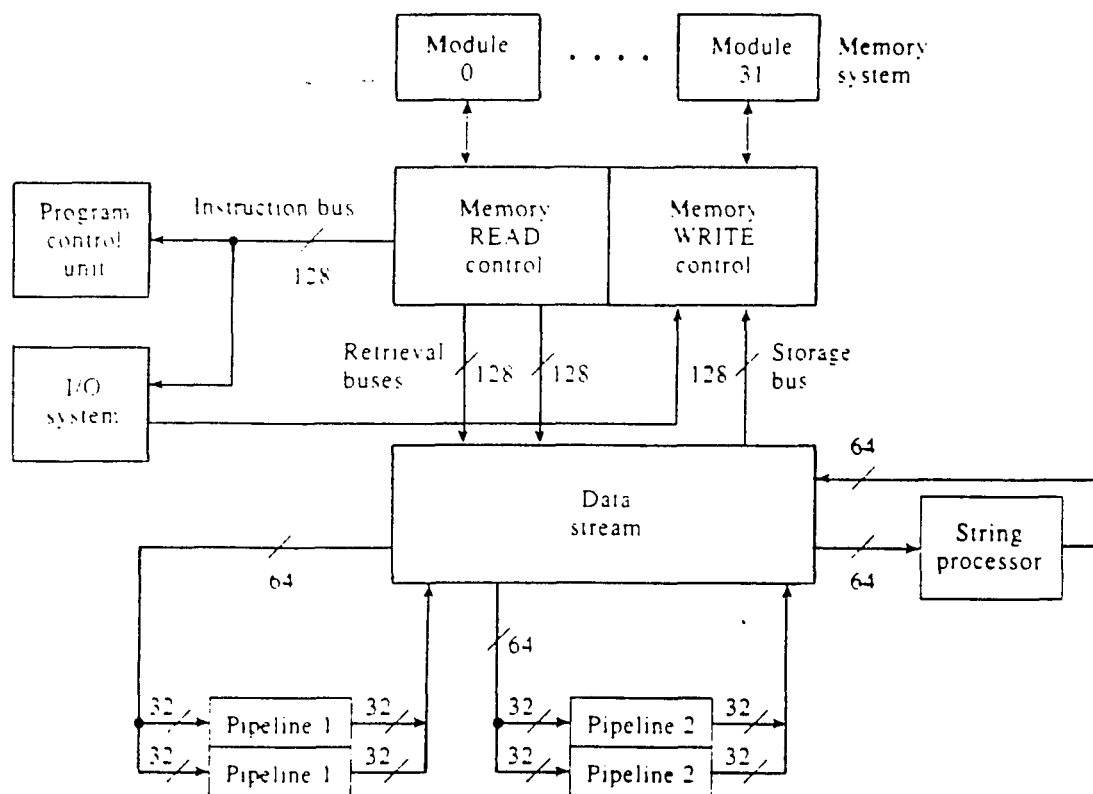


Figure 2.5 CDC STAR-100 Structure

With a slow main memory cycle time of 1.2 Micro Second, it is too slow to maintain the high data rates required to support the vector pipes, except in the case for exceptionally long vectors with successive elements stored in adjacent memory locations. Access to randomly spaced data for scaler operations takes the full 1.2 Micro Second which is ten times slower than the CRAY-1 machine. It has an exceptionally slow vector start-up time which allows the vector pipes to be effective only for vectors at least 100 long and therefore 30 times slower than the CRAY-1. Because of its unary vector increment, there is no provision for accessing vectors with an increment between elements, different from one.

The strength of the STAR-100 lies in the extensive software that has been developed during its long gestation period and the extensive instruction set provided in the machine. Nevertheless, it performs poorly on scaler arithmetic and is four times slower than arithmetic on the IBM 360/195 and the CDC 7600.

2.1.3.3.2 The TI-ASC

This pipeline computer was designed specifically with the processing of seismology data in mind. The logical basic configuration comprises a 4-16 megabytes memory in MOS technology with 160 ns access time [99]. Up to four pipes and two instruction processing units (IPU) may be supplied in

varying configurations. The memory is interfaced with pipes by the memory buffer unit (MBU), that has the job of ensuring a smooth flow of arguments and results to and from the pipes. Figure 2.6 shows the all-purpose pipe of the TI-ASC and how different sections of the pipe are used during the operations of addition and multiplication.

The vector instruction set of the TI-ASC is of particular interest. A single vector instruction is capable of performing the functions of a nest of three DO-Loops, whilst the innermost loop is restricted to single increments. The machine is therefore a unary stride system which can fetch several contiguous words in one memory request by increasing the bandwidth of the memory ports.

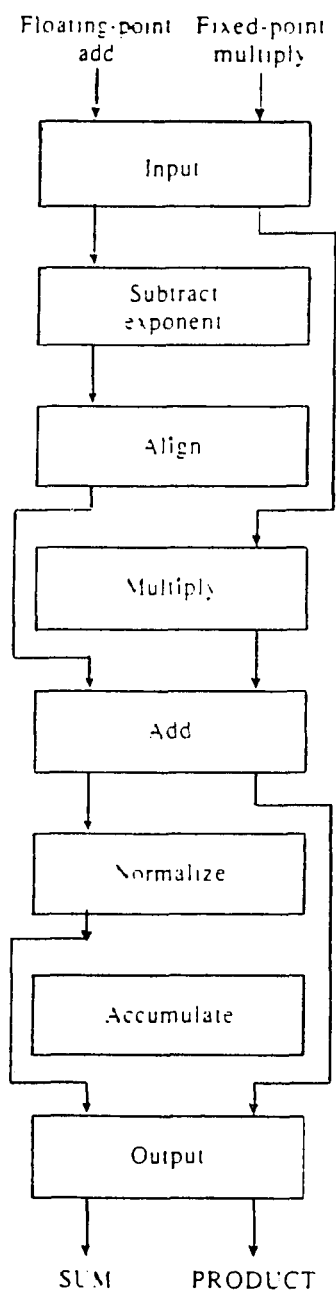


Figure 2.6 TI ASC Pipeline

The scaler performance of the TI-ASC is very similar to that of the STAR-100 and is, therefore, also slower than that of the CDC 7600 [44]. The vector start-up although long is shorter than that of the STAR-100. The maximum processing rate even with four pipes is 50 Mflops. Unlike the STAR-100, increments other than one are permitted in the outer loops of the three DO-Loops are efficiently implemented.

2.1.3.3.3 The CYBER 205

The CYBER 205 is a second generation vector supercomputer. It is a completely re-engineered STAR-100 in LSI with semiconductor technology and it is highly competitive with the other second generation vector supercomputer the CRAY-1. The basic configuration comprises one, two, or four general purpose pipelined arithmetic units and a stream unit, working with a clock period of 20 ns. The improved pipeline and memory speed have reduced the absolute value of the vector start-up from about three Microseconds for addition and multiplication on the CYBER 205, thus overcoming part of the long vector start-up disadvantage of the unit vector increment remains. This arises because both these features are properties of the architecture of the machine - that has not been changed - and not of the technology of which the machine is built.

Each one Mega-Word section of the memory is connected to the memory interface by a 512-bit wide data path (Superword or Sword). Thus, each data can be used to fetch an equivalent of eight contiguous 64-bit words in one memory request since each memory section is organized into 16 memory stacks and each stack is divided into eight independent banks.

The CYBER 205 uses a vector of bits to selectively create components without first expanding it. Unlike the CRAY-1, there are no vector registers, and all vector operations are main-memory to main-memory operations, necessitating data to travel a round trip of about 50 ft, compared to less than 6 ft on the CRAY-1. This difference partly explains the much longer vector start-up time on the CYBER 205.

The performance of the machine is slightly degraded when vectors whose components are non-unit distances are being processed. Before the operation is performed, the components have to be grouped into contiguous memory locations and re-stored consecutively into the correct result locations. The scatter and gather instructions can not be performed at arithmetic speeds, and they are therefore bottlenecks to the machine performance. In a two-pipe CYBER 205, the asymptotic performance rate of 64-bit vectors for most instructions is 100 Mflops, compared with 80 Mflops for such dyadic operations on the CRAY-1. This rate is reduced to 40-50 Mflops for the

scatter, gather, maximum/minimum and product of element instructions.

2.1.3.3.4 The CRAY-1

The architecture of the CRAY-1 comprises a main memory feeding data to and from a set of scaler and vector registers. The arithmetic and logic are performed by 12 independent functional units working to and from the registers. Central to the machine is its bipolar memory with a cycle and access time of 50 ns (3 times the speed of the TI-ASC and 20 times the speed of the CDC STAR-100). There are 64 buffer registers for (24-bit) addresses and 64 scaler buffers for (64-bit) floating-point numbers. These buffers lie between the main memory and the registers. There are 8 address registers, 8 scaler registers and 8 vector registers. The registers have a 6 ns access time. This is the only machine with vector registers that have the capacity to hold vectors of up to 64 elements long. Larger vectors must be broken up by program logic into vectors less than 64 long. All vector arithmetic takes place between the vector registers. There are 12 independent functional pipes which may perform arithmetic in parallel. An important feature of the design is the ability for a sequence of vector instructions to chain together and act like a single long arithmetic pipe. Intermediate results flow out of one pipe and into the next without reference to main memory thereby reducing memory accesses.

A 64-bit word is fetched from the store in 1 Clock Pulse (CP) using a single memory port. Therefore, if the vector registers cannot be used to contain the entire operands, the units have to be held up while the registers are being re-filled. This situation has been improved in the CRAY-X-MP series which use three uni-directional ports between the vector registers and the main memory.

CRAY-1 allows non-unit increments but cannot handle efficiently component accesses via an index list. Consequently, sorting algorithms fail to exploit the vector operations but still run faster than a good scaler algorithm designed for the CRAY-1. The machine has no facility for sparse vector operations.

The scaler operation rates are about twice that of the CDC 7600 or the IBM 360/195. The maximum vector capability is 160 Mflops. A combination of chaining and the parallel operation of the independent add and multiply pipes leads to an average operation of 7.5 ns, which is shorter than the clock time of 12.5 ns. The measured maximum rate is 133 Mflops for matrices that are a multiple of 64 (the vector register size). The rate degrades to 100 Mflops for matrices just larger than the register size [44].

2.1.3.3.5 The BURROUGHS BSP

The BSP system comprises sixteen unpipelined general purpose arithmetic processing units and pipelined instruction processing. The array section of the BSP contains four units which taken cyclically form a five-element pipeline. The memory-to-memory operations or tasks performed by this pipeline are as follows :-

- (a) read parallel memory operands;
- (b) align operand data with arithmetic units;
- (c) perform operation;
- (d) store results in parallel memory.

These operations are overlapped with successive sets of 16 elements taken from the vector instructions processed by the array control unit. The number of clock pulses required for each of these tasks is variable, depending on the number of operands and operations performed.

The BSP employs alignment network for providing conflict free accesses for the arithmetic units from the parallel memory. The parallel memory is used for storing data structures to be processed by the arithmetic units and consists of seventeen memory modules. The choice of a prime number of memory modules different from the number of arithmetic processing units allows the use of a mapping

algorithm that reduces the number of memory conflicts that may arise in common matrix manipulations. For instance, rows, columns and diagonals of a matrix can be accessed simultaneously [43].

There are two separate networks, an input network for fetching and an output network for data storing. The two networks contain full crossbar switching circuits, as well as hardware for broadcasting data to several destinations and for resolving conflicts. The consequence of the alignment network is that it was difficult to reduce the BSP's clock period of 160 ns to a smaller value comparable to the 12.5 ns of the CRAY-1. Thus, considerable improvement in performance could not be achieved even by fully pipelining the arithmetic units.

The file memory consists of a high-speed storage device based on Charge Couple Device (CCD) technology [43,95]. CCD devices are solid shift registers where data is stored as charged MOS capacitors. The characteristics of CCD based secondary stores are the twenty-fold increase in access speed over the state-of-the-art discs, and the very high transfer rates that can be obtained. The basic philosophy of the BSP design is to provide continuous high performance without interruption from I/O system. Thus, looking at the BSP performance figures without this perspective can be a little misleading; since Burrough has not aimed for massive

performance figures which cannot be sustained, but instead have designed a computer which can maintain 50-100 % of its maximum performance (50 Mflops) continuously.

2.1.3.3.6 The CRAY X-MP

The CRAY X-MP series [8,18], shown diagrammatically in **Figure 2.7** has major innovations in architecture and technology which offers overall system throughput up to five times that of a CRAY-1 S/1000 CPU, and a maximum burst rate up to eight times that of the CRAY-1 for specific cases. At the same time, software compatibility has been maintained between the CRAY X-MP and the CRAY-1.

Effectively, the CRAY X-MP is thus a powerful multiprocessor vector processing system where the mainframe features two identical Central Processing Units (CPUs) and a multiport memory. The dual CPUs allow for both multiprocessor jobs and concurrent independent jobs while sharing a two- or four-million word bipolar Central Memory. In this mode, four parallel memory access ports per processor provide over eight times the total usable memory bandwidth of the CRAY-1.

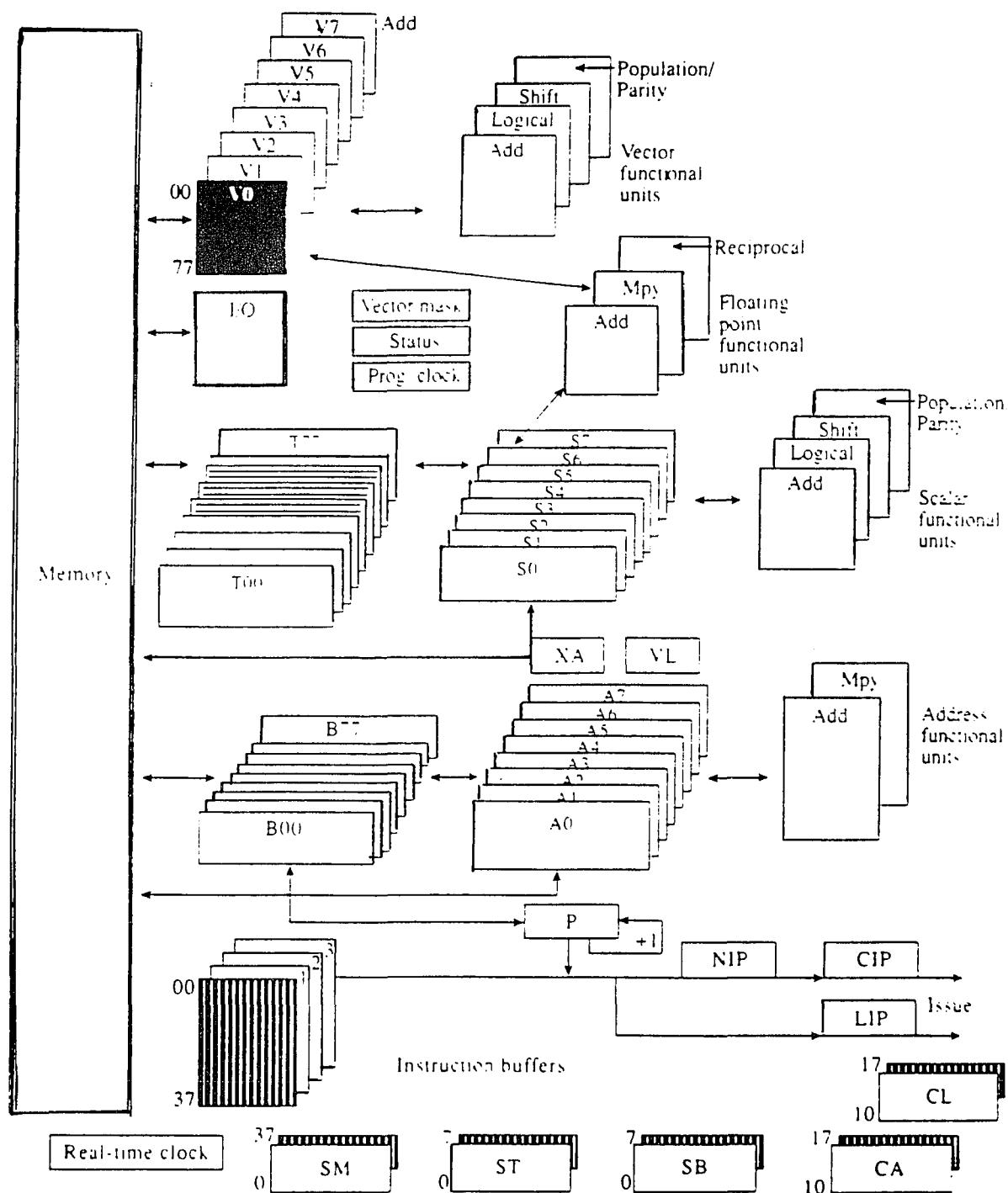


Figure 2.7 Cray X-MP CPU Structure

The CRAY X-MP computer system, with its 9.5 nsecs clock time, is the fastest general-purpose computer system commercially available today. It is capable of an overall instruction issue rate of over 200 million instructions per second (MIPS). Computation rates of over 400 Mflops are possible, and combine arithmetic/logical operations per second (MOPS).

Complementing the CRAY X-MP and designed with its demanding throughput requirements in mind, is the CRAY Research Solid-State Storage Device (SSD). The SSD is available in sizes of 64, 128, or 256 million bytes of on line storage. Using the latest memory chip technology, the SSD greatly reduces the access and transfer times over that for conventional storage devices. The SSD is placed such that it connects to the CRAY X-MP mainframe through the specifically designed 1250 Mbyte/sec channel so that theoretically the hardware can transfer 8 million bytes of data in milliseconds between the SSD and the mainframe.

The high performance of the CRAY X-MP is evident in both scalar and vectors. The scalar performance is improved through the faster clock, short memory access time, and larger instruction buffers, while a vector performance is improved through a combination of faster clock and, parallel memory ports and hardware automatic 'chaining' features. These new

features allow simultaneous memory and related vector instructions. Either long or short vector instructions, characterized by heavy register usage or heavy memory references, use these features to advantage.

2.1.4 Dataflow Architectures

The advocates for dataflow architectures contend that pipelined, vector and array architectures can not reach the performance goals that computer scientists would like to be met by the next generations of machines, whatever the improvements in VLSI (Very Large Scale Integration, **Section 2.1.5**) may be in the future. As a result of this assertion computer designers who share this view have turned to new architectures depicted by the data driven and demand driven concepts [9,21,38,43,97,99].

Dataflow uses a finite directed graph to describe a computation. The arcs of the graph are queues of data directed from one node to another. The nodes represent single valued functions which map input data onto output data. In general, a node is enabled for execution when there is a data token present on each of its input arcs. The node then executes and consumes the set of inputs removing one data token from each arc. It processes them according to the specified functions and releases a set of result tokens onto the output arcs which enable further nodes. The data token

provides a dual role in the dataflow model supporting both the data mechanism and control mechanism, whose flows of data control are therefore inseparable.

Computer architectures for dataflow schemata embody very different principles to those of the traditional Von Neumann systems. Specific architectures have been proposed in some theoretical papers referenced above and elsewhere [43,88]. It is, therefore, not surprising that dataflow has been studied carefully in the context of parallel computation.

2.1.5 VLSI System Architectures

From its origins in the early 1960's, integrated circuit technology has advanced to the point where Large Scale Integration (LSI) allows tens of thousands components (i.e., transistors, magnetic bubbles. other information holders) to be built into a single chip. Inevitably, this led to the design of microprocessors where a complete microcomputer system can be assembled on a semiconductor chip. The exact number of components in one chip depends on the speed of the devices and the regularity of the pattern used to lay them out on a chip.

The rapid advances in integrated circuits technology resulted in the design and manufacture of the Very Large Scale Integrated (VLSI) circuits where the number of transistors that the VLSI circuits contain will be increased by another

factor of 10 to 100 in the next 10 to 20 years [73]. By the late 1970's it became possible to design chips containing millions of transistors. The devices and interconnections in such VLSI systems have linear dimensions smaller than the wavelengths of visible light.

Chip design, generally, requires computer-aided design assistance because of the large number of components, packaging, and manufacturing problems which so far yet to be solved. This predictability suggests the VLSI will only be used in memories which are easier to design since the patterns are extremely regular. This is partially confirmed by the LSI experience which shows that the number of components in memory chips can be as much as five times that of a processor chip. This is evidently seen by the quadrupling every few years of the number of bits per memory chip.

The advanced technology involved, forecasts radical changes in the chip design and computer system architectures. Parallel on a single chip, in either MIMD, SIMD, or pipelining fashion is advocated. The design of these chips will rely heavy on the regularity of patterns for communication between active elements. Meanwhile, the study of VLSI circuits is not limited to the hardware aspects but also software tool become more and more important in the design and testing stages. This means that the production of a new chip requires software

as well as hardware engineering knowledge. Computer scientists, however, have developed and are still developing more new algorithms and new communication techniques to exploit the potentiality of the VLSI system.

More recently, a lot of research interest has been concentrated on the design and application of the VLSI circuit in parallel computation where the large systems of processing elements cooperate in the execution of parallel algorithms. The difficulty in selecting specific interconnections is frequently overcome by assuming a model (shared memory machine) where each pair of processors is connected. It has also been suggested that processor interconnection be limited to planar links between topologically neighboring cells (arrays, meshes) or hexagonal interconnection [33,63,73]. Such designs are certainly well suited to current VLSI technology, and have been used skillfully in implementing algorithms for matrices or graph problems [63]. This type of interconnection is, however, not suited for efficiently implementing algorithms for various fundamental problems such as sorting and convolution. More recently, an interconnection pattern of processing elements, the Cube-Connected Cycles(CCC) is proposed which can be used as a general-purpose parallel processor to solve efficiently a large class of problems that include the Fast Fourier Transform, sorting and permutation [63]. Since its design complies with present technological

constraints, the CCC can also be used in the layout of many specialized VLSI circuits. The operation of the CCC is based on the combination of pipelining and parallelism where the number of connections per processor is reduced by three.

As the current technologies offers the possibility of producing 32-bit floating-point arithmetic chips which perform an operation in 1 Micro Second, multi-layer circuit boards have been designed: holding 200 chips a piece and arithmetic units containing 10 circuit boards, therefor, Kung's proposed machine [63] can perform 2000 operations in 1 Micro Second rate of 2000 Mflops. This operation rate is however far in excess of that achievable by the fastest currently available machine, e.g. the CRAY Y-MP/1.

The system, nevertheless, faces two problems. The first one is that the decomposition problem, which can be reduced if the algorithm designer creates and expresses parallelism in problem formulation, algorithm design and program development. However, parallelism is crucial not only to the performance of a VLSI system but also the potential power of VLSI has to come from one large amount of concurrency that it may support. The second problem that limits the speed is the communication problem. The bottleneck for communication appears at all levels of the hierarchy of a computer system. For example, at the inter-chip communication level, pin bandwidth

limitation is observed. It is important to note that increasing chip density does not help to reduce the number of chips needed to cover a network. Therefore, networks that can be decomposed by removing only a small number of links are likely to be able to take advantage of the increase in chip capacity. The solution of the communication bottleneck is to design circuitry that can be partitioned among separate chips so that inter-chip communication is either completely avoided or reduced to a minimum. Inside a chip, however, the wiring for communication can easily dominate the power, chip area and the time required to perform a computation.

In order to solve the above problems, Kung [63] has introduced the "Systolic Array" in which the wiring in chip design is reduced to the minimum. Because of the regularity of the data and control flow, such systems were termed "systolic" and were found to be ideal for direct implementation using VLSI technology. A systolic system comprises identical simple cells that circulate data in a regular fashion. The array structure for such a system provide a neat and regular pattern that allows easy communication between the cells of many arithmetic units(ALUs). Thus, a much higher computation throughout can be achieved without increasing the memory bandwidth. Generally, the systolic array approach can be applied to all levels of a computer system.

In practice, many systolic arrays have been designed to solve problems in many area such as matrix computation, database manipulation and signal and image processing. In the design of the systolic arrays, the following ideas should be considered:

- (1) The design should use a few types of simple cells
- (2) Data and control flow should be simple and regular; there should be an extensive pipelining and parallelism where several data streams move at a constant rate over fixed paths in the network, interacting at cells where they meet. In this way a large number of cells are active at a time so a high computation can be achieved.
- (3) Multiple use should be made of each input data item. Therefore, high computation throughout can be achieved without requiring high bandwidth between the array and the memory.

In the implementation of a systolic array, each word processor can be partitioned into bit-serial implementation, it is possible however to implement about ten basic cells, performing 32-bit fixed operations on a single chip with MOS

technology. When many such chips are used the systolic array can achieve a high computation rate. This is because each word is used many times in the array.

This discussion concludes that VLSI technology offers a great reliability at the circuit level. It also has the advantages of simple and regular interconnections that lead to cheap implementation and high densities where high density implies both high performance and low overhead for support components.

2.2 Programming Language Requirements

Vector and parallel processors pose new problems in the area of design of high level programming languages since their vector architectures are radically different from the traditional Von Neumann machines. Most of the traditional languages, such as PL/1 and Fortran are based on the sequential flow of control originating from the Von Neumann concept of a machine.

2.2.1 Language Extensions for Parallel Processing

In order to perform parallel programming using current parallel computers, the traditional serial languages had special primitives added to extend them to parallel programming languages. FORTRAN, PL/1 and other languages [7,64,82,92,106] have been extended in this manner to support

vector declarations and operations. Multi-tasking in PL/1, for example, implements CAL and WAIT primitives by allowing more than one stream of instructions to be active at any instant. These are usually synchronized by co-operation between several processes. They, therefore, represent natural extensions to the sequential control flow concept. These extensions are, however, restricted, in the sense that there is a limit to the level of parallelism exploitation of these primitives without introducing errors and timing difficulties. In addition, the extended languages do not adequately exploit the advanced architectural features of these machines and therefore do not lead to any significant increase in performance.

Apart from the above extensions, APL [46] has been adapted to support vector and array processing on various computers. Even though it has been successfully applied in formulation of large scale scientific problem [29,34,43] it has never been universally accepted by the array and vector processing community for the following reasons. Firstly, it has a very large vocabulary which makes program logic and readability difficult to comprehend. Secondly, it contains no provisions for indicating parallel execution of at least two such statements. Thirdly, APL has been implemented as an interpreter as a result of its lack of explicit declarations. A variable may be scalar at one point in a program and a

vector in another. This consequently requires execution time testing and will probably make the language too slow for large models. Fourthly, APL lacks structures commands to control the program flow. It is, however, a very powerful language providing a comprehensive set of vector and array operators for effective vector and array processing units specialized environment.

One other limitation against the use of some of the current vector and array processing languages is the requirement that the user be aware of the machine instruction and/or the method of detection of parallelism employed by the compiler. In addition, the user is expected to restructure his program by using low level primitives to organize the transfer of data in order to exploit the parallelism of the computer. This, inevitably, inhibits the user from concentrating on the logic and formulation of his problem and hence results in an inefficient code which critically affects the performance of the program. One solution to this inefficiency lies in the provision of intelligent and optimizing compilers which relieve the user of low level of performance.

2.2.2 Machine Dependent Parallel Languages

A number of vector programming languages have been designed which introduced some of the advantage structure

programming to parallel and vector computers, e.g. SL/1 [7,43]. The level of the languages is largely vector architecture dependant and is directed at the level of the machine. It is such that the user is forced to express efficiently his executable algorithms by making maximum use of the specialized hardware without being exposed to the hardware peculiarities. Even though SL/1 has been directed at the level of the STAR-100, there are no existing hardware structures to support its micro-defined functions. The software implementation of these functions felt to further reduce the potential for an increase in performance.

2.2.3 Machine Independent Parallel Languages

The design of parallel machines independent languages with sufficient generalities provides the programmer with necessary for efficient construction of algorithms making optimize and maximum utilization of the hardware. ACTUS [82] comes within this category for both vector and array processors with very little hardware dependence. The advantages of this language lie in the fact that it enables the expression of variable parallelism to be exploited by both kinds of architecture. It also enhances rapid and efficient code generation by the compiler without the detection of inherent parallelism in the program. It also helps to produce algorithms of greater efficiency without resorting to sizing the data structure of the problem.

ACTUS suffers from the same limitations as SL/1 where the vector or the array processing functions are implemented through software without being supported by hardware mechanisms. It is felt that these limitations further reduce the abilities of computers like the STAR-100, CRAY-1 and ILLIAC-IV to exploit maximum parallelism. A high level language that is well matched to the source program can make compilation and control unit designs easier and also help to improve system performance.

The limitations of the current vector and parallel computers discussed above have made it expedient to look at alternative approaches to overcome these inadequacies by providing simple mechanisms for parallel computation that have a comparable representation between the architecture and language. The main attribute of that mechanism lie in the fact that parallelism is explicitly represented in the language for ease of programming and that all vector functions and operators have hardware structures for their implementation that will lead to an increase in performance. Moreover, the NetVecX supports sparse processing using hardware implemented mechanism and a new sparse data storage structure to enhance sparse processing performance. Finally, the main advantage of using multiple vector units is that the mapping of numerical problems onto the hardware involves a significant constraint in one direction as opposed to two in

an array processor, since vector lengths can be several orders of magnitude greater than the number of processors.

CHAPTER 3

APPLICATION CHARACTERISTICS AND BENCHMARK DESIGN PROBLEMS

The main objective of this chapter is to discuss the general characteristics of the large scientific application problems at which the **NetVecX** architecture is aimed. In addition, a discussion is presented on a proposed standardized technique for performance evaluation of the machine architectures described in **Chapter 2**. Two typical applications are considered in order to identify the major data and program structure characteristics which influence the design of parallel algorithms. Secondly, the discussion is also aimed at stimulating ideas for the design of new parallel algorithms to exploit the vector capabilities of the **NetVecX** architecture. Current parallel computer employ numerous serial and parallel algorithms in the solution of large scientific application problems [11, 14, 15, 40, 41, 43, 62, 67, 72, 74]. In order to study the interactions of the various components of the **NetVecX** architecture and to characterize a typical workload, three widely used algorithms with inherent parallelism are presented. The parallel versions of these algorithms are compared to their serial counterparts to illustrate the extent to which parallelism can be exploited in serial algorithms to take advantage of the advanced architectural features of parallel machines.

3.1 Large Scale Scientific Computation Application

The scientific nature of the problems means that parallel computers are being used to model some physical phenomena for which millions of words of data on backing store are required. This effectively implies that a high data bandwidth is required to be connected to the main processing store. Having achieved this, the power of the processor must be such that it can process data within "reasonable" time so that results are available for prediction or modelling simulations. Typical application areas for such machines are:

- (1) Structural stress analysis,
- (2) Electrical network analysis,
- (3) Numerical linear algebra,
- (4) Digital signal processing calculation,
- (5) Image and pattern recognition,
- (6) Weather forecasting,
- (7) Military applications, and
- (8) Large hydrodynamic problems.

The next section considers the general method of solution of the numerical weather forecasting and computational fluid dynamics problems in order to illustrate the computational requirements and the need for a more powerful computer.

3.1.1 Applications Characteristics and Computational Requirements

Numerical Weather Forecasting

A typical approach to this problem is the modelling of a grid to encompass the earth so that information is collected at each of the grid points [41,90]. This data can then be used to solve the equations which govern the interaction of the elements by using a technique based on finite differences. The processing of the data can be performed with half hemisphere or globally from pole to pole. Since the same set of instructions is to be applied independently to the data along a particular processing direction it is possible to perform much of the processing in parallel or to overlap operations. During the course of processing, communication between adjacent data points is established to enable the large amount of data which has to be updated frequently in order to obtain convergence to be processed in parallel. The net effect is that results are produced much faster than with a sequential machine. This, in turn, can mean the difference in predicting or not predicting the occurrence of a storm, hurricane or major abnormality in the weather. In many situations numerical weather simulation is the only tool available since controlled experiments cannot be carried out. It is therefore expedient to employ a very high performance computer system in such simulations.

Computational Fluid Dynamics

This application area also requires large amounts of processing power to advance its development. For instance, in the design of an aircraft, the main tool available is the costly wind tunnel in which the model of the proposed airplane is constructed, strapped with sensors and blasted with air in order to simulate flight conditions and therefore to detect design faults. However the amount of time required and the waiting time for an available wind tunnel are prohibitive and therefore not cost-effective means to simulate flight conditions. The heart of the matter therefore rests on the use of computer simulated flight conditions.

In the computational sense, the Navier Stokes equations which govern fluid dynamics take quite some time to solve in even two unknowns using known numerical techniques on the fastest processors [41,105]. The equations can also be solved by using a grid of data points which is updated in a parallel fashion. The grid spacing determines the amount of data required and the accuracy of the results. The types of techniques which are used involve the Fast Fourier Transform (FFT) and finite differences [32,78,79]. With a very high performance parallel computer, it is feasible to perform substantial aerodynamic simulations on a computer in order to eliminate unpromising designs. This is the view of the NASF project group [105]. In order to accommodate such calculations, a machine which is capable of solving the three

dimensional Navier Stokes equations with high Reynolds number within "reasonable" time is required. The CDC preliminary report [12,13] and the Burroughs Report [13] indicate that the NASA machine will require the ability to execute half a billion floating points operations per second. the architecture has been based on current parallel array processor with 1024 processing elements and a variant of FORTRAN. The machine therefore benefits from the latest technological hardware components but certainly not the latest software which may partly degrade its performance.

These problems significantly indicate that dynamically varying data structures are not essential but that conformable arrays, arrays with primitive, enumerated, subrange or record data items are desirable for mapping the numerical problem onto the hardware. The dimensions of the grid are important considerations, especially for array processors, where a high degree of efficiency is obtained by choosing one of the grid's dimensions to coincide with the hardware's dimensions as opposed to the problem's characteristics. In the proposed **NetVecX** architecture, the degree of efficiency does not necessarily rely on the grid size matching the hardware's characteristics since in theory any vector unit can process any length of vector.

3.2 Exploitation of Parallelism in Numerical problems

The development of parallel processors has stimulated continuous study of parallelism in computation [62,74,93,97].

Over the past decade the interest of numerical analysts in new algorithms to exploit vector and parallel computers has been on the increase. Most of this work took two forms; either the algorithm is designed with the computer architecture in mind or the algorithm was discussed in the context of a model hypothetical computer with unlimited number of ideal processors. The early review articles of Miranker [74] and Heller [40] is a summary of some of this work. On the other hand with the advent of the STAR-100, TI-ASC, CYBER-205 and the CRAY series the influence of vector computer architecture on numerical algorithms has been on the increase. A critical survey of these algorithms reveals that for most of these computers they must be developed with the specific architecture well in mind if anything approaching the full potential of these machines is to be realized. It is therefore the purpose of this section to discuss the broader concept of vector operations which will provide the opportunity for the exploitation of processing and for the design of new parallel algorithms.

3.2.1 Multi-Operation Functions

In vector computation, single-valued (scalar) operations are applied successfully to the elements of a vector (or a single result in the case of functions such as the dot product). Most of the numerical algorithms [40,74] have single inner-loops which account for a large fraction of the computational time. Hence the use of multi-operation functions

is highly desirable in order to eliminate this time consuming loop-mode feature. For instance, as an interesting example of recursive vector operations, consider the formation of a dot product of two vectors on the CRAY-1:-

$$A = (a_1, a_2, \dots, a_n)$$

$$B = (b_1, b_2, \dots, b_n)$$

such that:

$$A.B = \text{SEGMA } (k=1 \text{ to } n) [a_k \cdot b_k]$$

where $n = 64$

This product is achieved by a vector loop and a scalar loop. The vector loop, which contains a multiply-add chain first computes 64 partial sums such that element n of the result vector register contains the sum of every sixty-fourth product $a_i \cdot b_i$ starting with $a_n \cdot b_n$. The scalar loop then adds partial sums to complete the dot product. The operational sequence and timing diagram are shown in **Figure 3.1** for the formation of a recursive vector sum.

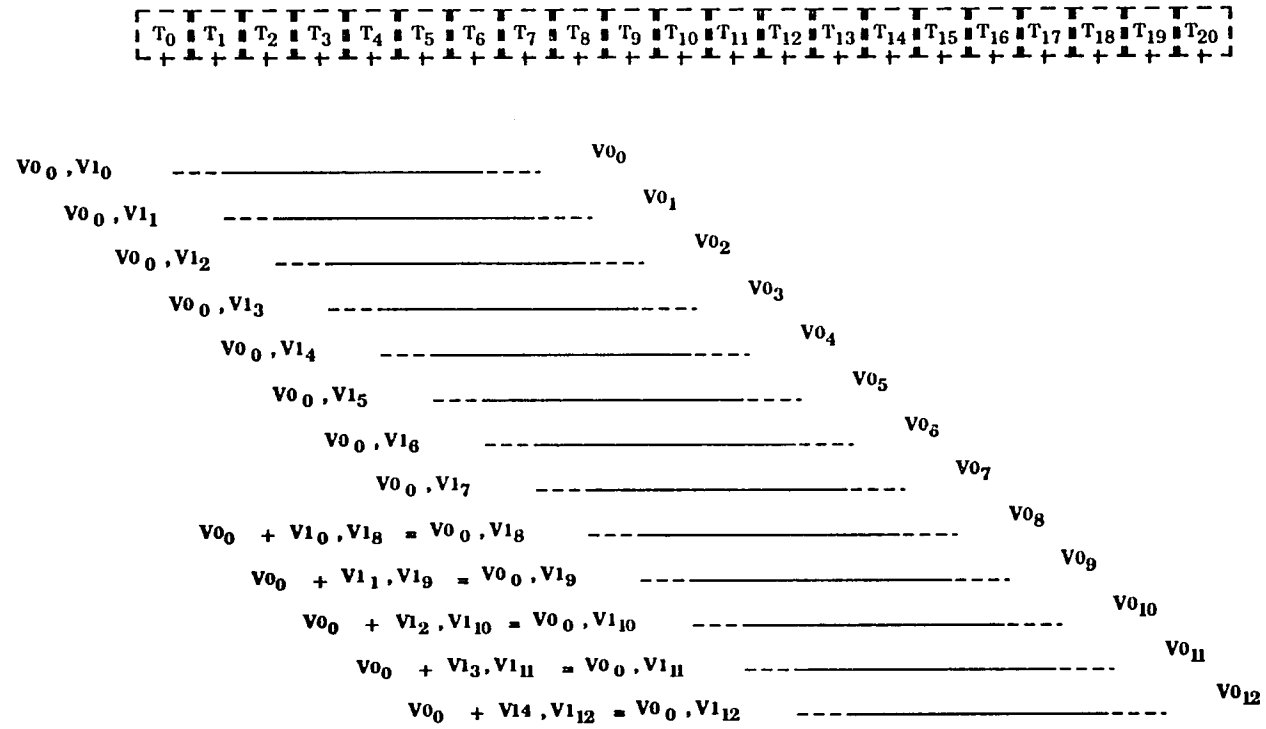


Figure 3.1 Timing Diagram for Recursive Vector Sum

The principle of improved performance with vector rather than scalar processing is shown in **Figure 3.2**. For the resultant production rates to approach the theoretical maximum, calculations must be highly interleaved and overhead loop instruction execution minimized. The use of multi-operation vector functions facilitates this by:-

- (1) Providing flexibility in scheduling the issuance of instructions;
- (2) Eliminating redundant store and re-fetch of intermediate results;
- (3) Amortizing loop start-up and control overhead over more calculations;
- (4) It also simplifies the programmer's task in developing his programs.

The STAR-100, TI-ASC and the CYBER-205 provide very large sets of vector operations while the CRAY-1 provides a basic (but well chosen) set of operations. It is also worth noting that the CRAY-1 has hardware features which make multi-operation vector computations possible, whereas other existing machines, except the CYBER-205, do not. This is a capability which should be required in any future vector computer.

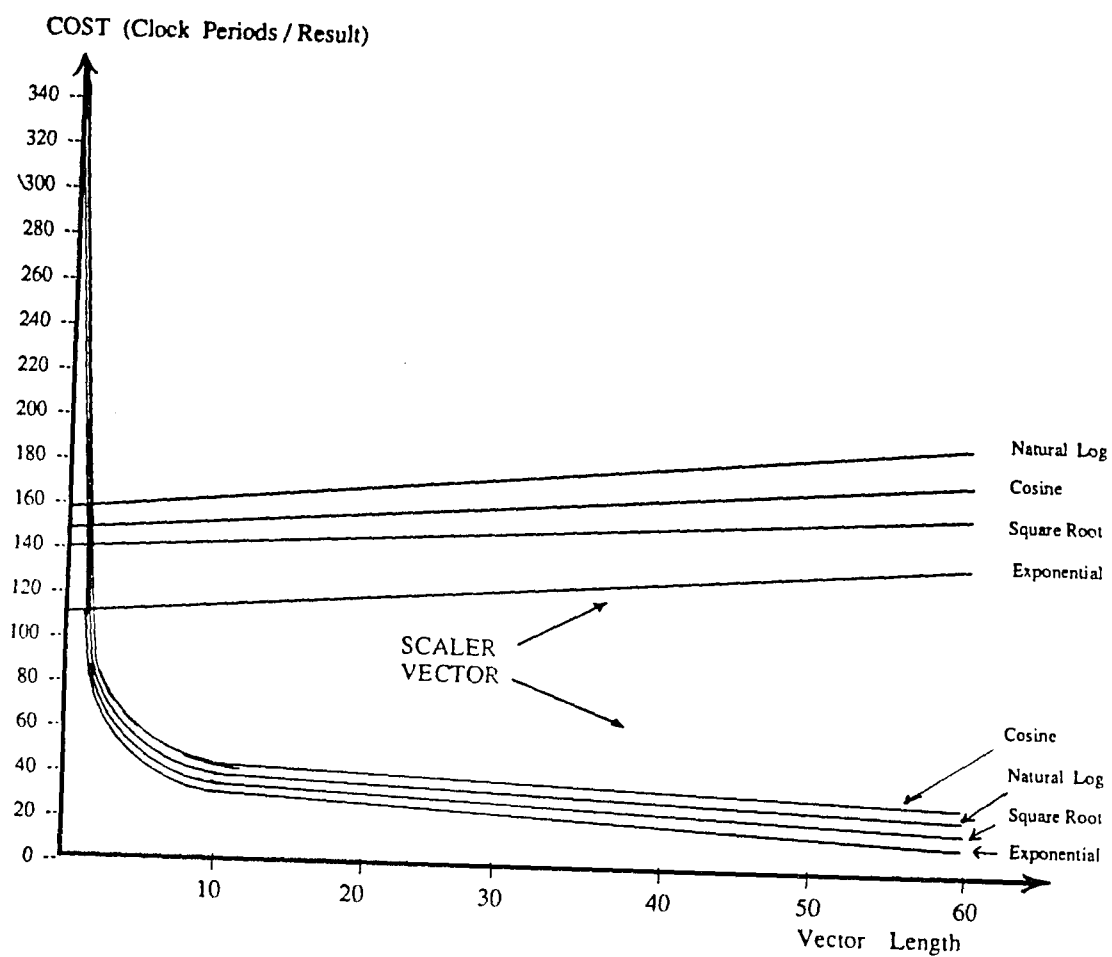


Figure 3.2 Comparison Between Vector and Scalar Performance

	GENERAL FORM	EXAMPLES
1	$V_0 = V_1 \text{ fn } V_2$	$V_0 = V_1 * V_2$
2	$V_0 = V_1 \text{ fn } S$	$V_0 = V_1 + SCL$
3	$V_0 = V_1 \text{ fn } (V_2 \text{ fnl } S)$	$V_0 = V_1 - (V_2 * SCL)$
4	$S = V_1 \text{ fn } V_2$	$SCL = V_1 * V_2$
5	$S = V_1 \text{ fn } S$	$SCL = V_1 - SCL_1$
6	$V_0 = V_1 \text{ fn } (V_2 \text{ fnl } V_3)$	$V_0 = V_1 * (V_2 + V_3)$

Table 3.1 Multi-Operational Vector Functions

PRIMITIVES INFORMATION

Definitions		Primitive No.	Weight Parameter	Parameter String
1	Data Assignment	PR01	W1	P1:1, P1:2
2	Tests	PR02	W2	P2:2, P2:2
3	Table Look-Up	PR03	W3	P3:1, P3:2, P3:3
4	Search List	PR04	W4	P4:1, P4:2
5	Computation	PR05	W5	P5:1
6	Proc. call/Linkage	PR06	W6	P6:1, P6:2
7	Space Creation	PR07	W7	P7:1, P7:2
8	Vectorize	PR07	W8	P8:1
Operating System Primitives				
9	Input-Output	PR50	W50	P50:1, P50:2
10	Store Request	PR51	W51	P51:1
11	Storage Return	PR52	W52	
12	Await Event	PR53	W53	
13	Inter-process comm	PR54	W54	P54:1, P54:2

Table 3.2 Some Primitives and Other Relevant Information

Table 3.1 shows the general form of multi-operation vector functions underlying most of the numerical algorithms which apply in the solution of scientific application problems.

It is quite apparent that the ability to effectively vectorize a problem and make optimal use of a vector structure such as our **NetVecX**, is dependent on the availability of means for data restructuring to produce vectors from otherwise non-vector constructs. APL [46] provided a conceptual basis and notational scheme for mapping these functions. The forms of the APL functions adopted and modified for this architecture are listed in [46]. The advantage of using multi-operation vector functions is exemplified by the use of such triadic operations such as:-

$$V_1 + S * V_2 \geq V_0(1)$$

$$(V_1 + S * V_2 \geq V_0(2)$$

which occur frequently in matrix problems (eg. the inner product of two vectors). These operate as single vector instructions with no intermediate reference to main memory as illustrated in Figure 3.3. This facility plays a similar role to chaining on the CRAY-1 and in the CYBER-205, but in a more restrictive way, because there, the special "select-link"

instruction is used to achieve linkage of the appropriate arithmetic units where one of the operands must be scalar. The **NetVecX** architecture imposes no such restriction.

3.3 NetVecX Oriented Parallel Algorithms for Performance Evaluation

As pointed out in **Section 3.2**, the algorithms for vector execution on vector machines should be designed with the specific architecture in mind in order to achieve anything near their full potential. Having put this into perspective, transformation have been applied to the following widely used sequential algorithms to yield new suitable parallel algorithms for execution on the **NetVecX** architecture. For simplicity, clarity has been favored at the expense of efficiency in order to demonstrate the effective exploitation of parallelism within the **NetVecX** architecture. The algorithms considered are:-

- (a) Matrix multiplication by the inner product method,
- (b) Gaussian Elimination with partial pivoting,
- (c) Gauss-Seidel method.

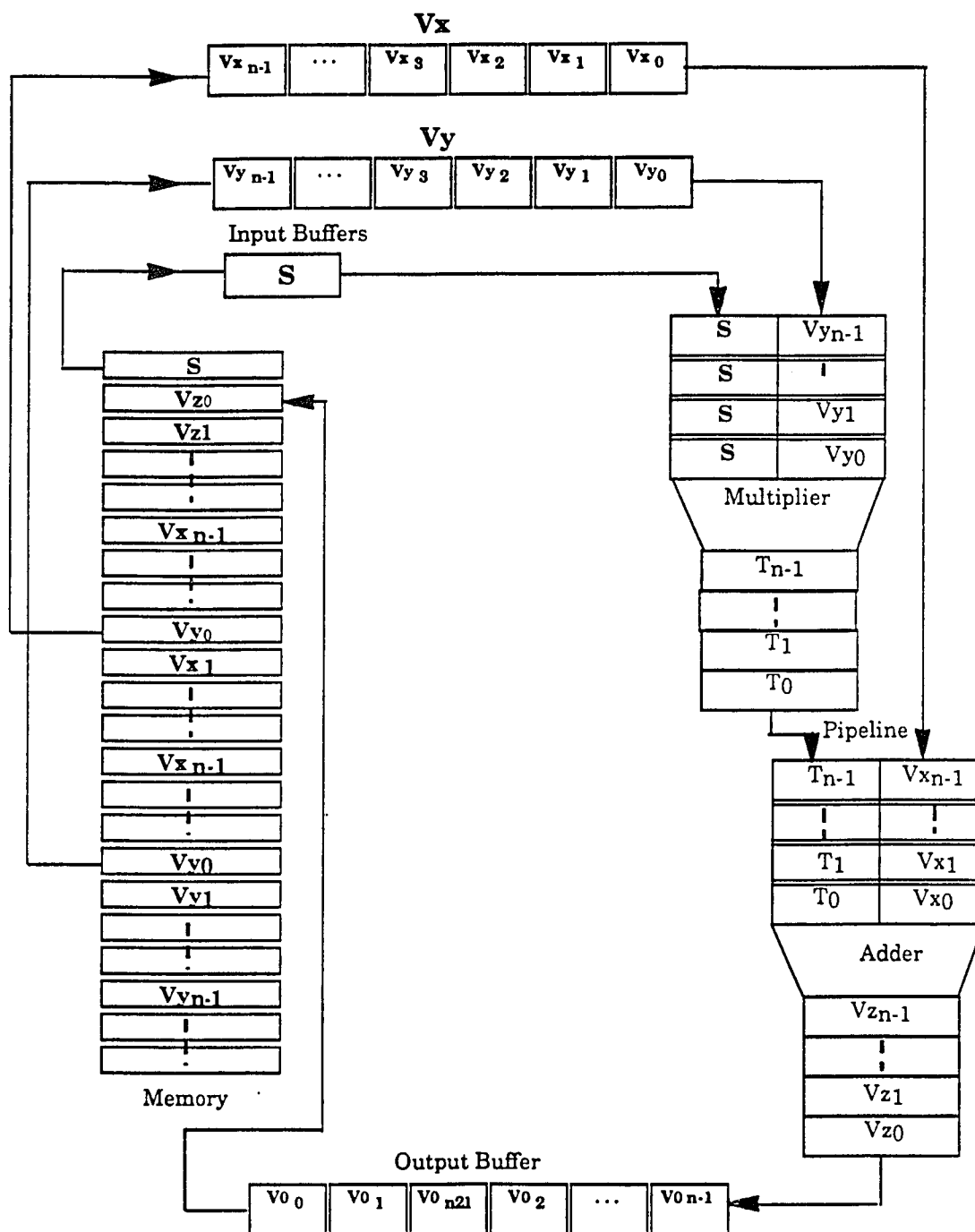


Figure 3.3 Multi-Operation Vector Pipeline With no Intermediate Reference to Central Memory

3.3.1 Matrix Multiplication

Matrix multiplication has been considered because the algorithm lends itself readily for transformation to suit the architecture on which it is to be executed and also amenable to vector processing. Matrix multiplication is typical of the large class of problems that can be vectorized. For these problems a significant increase in processing speed can be achieved over conventional scalar processing. The use of the multi-operation vector functions and the large amount of concurrency attainable through the use of the 16 vector processing units of the **NetVecX** architecture (see **Chapter 4**) will provide high processing speeds comparable to at least the CYBER-205 and the CRAY-1 computer systems when fully implemented. In mathematical terms, the relationship of the elements C_{ij} of the product matrix to the elements of the source matrices, A_{ij} and B_{ij} , being multiplied, is expressed by the equation:

$$C_{ij} = \text{SEGMA } (k=1 \text{ to } n) [A_{ik} \cdot B_{kj}], \quad 1 \leq i, j \leq n \quad (3)$$

Where the first subscript is the row number and the second subscript the column number. On serial machines matrix multiplication is performed using three nested DO-LOOPS as illustrated in the following FORTRAN code segment obtained by direct translation of the above definition, viz;-

```

DO 5 I = 1,N
DO 5 J = 1,N
DO 5 K = 1,N
5    C(I,J) = C(I,J) + A(I,K) * B(K,J) (4)

```

assuming all elements $C(I,J)$ have been initialized to zero.

The

inner loop expression forms the "inner product" of i^{th} row of A and the j^{th} column of B. As a result of the wide occurrence of an inner product in most algorithms, the "inner product or Dot product" instruction has been provided by most of the high performance computers. It is a form of the multi-operation functions defined in **Table 3.1**.

A great deal of parallelism is inherent in the evaluation of the matrix product because it involves the evaluation of inner products (for $n \times n$ matrices) which may be performed n at a time. It demonstrates that n vector processes can be set up simultaneously to achieve high performance.

3.3.2 Parallel Gaussian Elimination Algorithm

This method is used for illustrative purposes because it is known to be stable and one of the widely used algorithms in the solution of simultaneous linear equations and matrix inversion. The statement of the simultaneous linear

equation problem is expressed in matrix-vector notation, thus

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (5)$$

where $\mathbf{A}$ is an n by n matrix of known problem coefficients, $\mathbf{b}$ is a known n -vector and $\mathbf{x}$ is an unknown n -vector whose value is sought. A unique solution is then expressed as:

$$\underline{\mathbf{x}} = \underline{\mathbf{A}}^{-1} \underline{\mathbf{b}} \quad (6)$$

Basic Gaussian Elimination can be used to demonstrate the exploitation of parallelism in a serial algorithm for vector processing. The basic algorithm is stated briefly in PASCAL-like notation in Figure 3.4.

The underlined sections of Figure 3.4 show the statements which lend themselves to vectorization by using multi-operation functions for each iteration. The vector functions "MAX.INDEX" and "SWAP.ALL" used in the pivoting section are used to eliminate the loop mode features which are normally performed sequentially in Gaussian Elimination. The statements containing these functions are then assigned to vector processing unit in lock-step fashion. One major area of time saving occurs in the elimination section where a vector unit is assigned to compute the multipliers, m_i where $(k + 1$

$\leq i \leq n$), before computing the inner loop intermediate values. The significant thing here is the amount of parallelism exploited by initiation of $n-1$ vector processing units per iteration, each of which uses a variant of the triadic function illustrated in Table 3.1 to execute the inner loop expression :-

$$a_{pj} := a_{pj} - m_p * a_{ij} \quad (k \leq j \leq n+1) \quad (7)$$

The significant improvement in the performance of the parallel algorithm over the serial version is shown in the evaluation figures in chapter 6. It shows the wisdom in using the parallel algorithms to gain an increase in performance.

```

For i := 1 to n-1 DO
  BEGIN
    l := MAX..INDEX(ai,j) ( 1 <= j <= n-1 );
    If l > i THEN
      SWAP.ALL(ai,j, al,j) ( i <= j <= n+1 );
      k := i+1
      mj := aji / aji ( k <= j <= n );
      FOR p := i+1 to n DO
        ap,j := ap,j - mp * aij ( k <= j <= n+1 );
      END
    END

    xn := an,n+1 / an,n
    FOR i := n-1 to 1 DO
      BEGIN
        S := INNER.PRODUCT ( xj , ai,j) ( i+1 <= j <= n);
        xi := (ai,n+1 - S) / ai,i ;
      END
    END
  END

```

Figure 3.4 Parallel Gaussian Elimination Algorithm

Finally, use is made of the inner product function to calculate the final solution of x_{ij} 's in the back substitution section. The parallel implementation of this algorithm partly demonstrates the nature of vector processing abstraction expected to be used for the **NetVecX** architecture.

3.3.3 Parallel Gauss-Seidel Iterative Algorithm

The Gauss-Seidel iterative algorithm has been chosen because it converges more rapidly [93] and it is a special case of the general class of relaxation method which are described by :-

$$(\underline{I} - w\underline{L}) \mathbf{x}^{k+1} = [(1-w)\underline{I} + w\underline{U}] \mathbf{x}^k + w\mathbf{r}' \quad (8)$$

$$(\underline{I} - \underline{L}) \mathbf{x}^{k+1} = \underline{U} \mathbf{x}^k + \mathbf{r}' \quad (9)$$

where w is a suitably chosen relaxation parameter. Clearly the choice of $w = 1$ yields the Gauss-Seidel method (9). The important point to note about (9) is that when computing the value of the i^{th} component of $\mathbf{x}^{k+1}$ the previously computed value of the first $(i-1)$ components $\mathbf{x}^{k+1}$ may be used. Thus accounts for the rapid convergence rate since more and more accurate information is being used as the higher components of the approximation to the solution are being evaluated. A variant of the parallel version of the Gauss-Seidel method embodied in (9) has been implemented in terms of matrix-vector operations. The algorithm segment written in NPSL (**Chapter 5**) is shown below in **Figure 3.5**.

```
REPEAT
  BEGIN
    S := INNER.PRODUCT (aij, xj) ; (1 ≤ j < n)

    xi := xj - (bi - S) / aii ;          (aii ≠ 0)

  END
UNTIL Converged;
```

**Figure 3.5 Program Segment of a Parallel
Gauss-Seidel Algorithm**

In this version of the algorithm the "INNER.PRODUCT" function is used in the FOR...LOOP. The scalar, s, is computed using one vector unit with a vector length of n per iteration and therefore eliminates the time consuming loop mode feature.

3.4 Parallel Benchmark Design Proposal

The aim of this section is a brief discussion of the philosophy underlying a proposal for the design of a standardized benchmark technique for evaluating parallel and vector computers.

There is a need in the field of parallel processing, for standardized techniques capable of comparing the relative performances of machines performing dedicated and well defined tasks. These techniques have not been extensively studied by researchers in the field. The characterization of performance of parallel and vector computers using generic formulae has, however, been recommended by Hockney [44], Hockney and Jesshope [43] to characterize the effectiveness of vector operations. While these techniques only provide first order descriptions of any real computer, they hardly take into account overheads introduced by compilers. It is therefore expedient to adopt a constructive approach for constructing benchmarks for parallel and vector computers. In doing so, some degree of standardization may be introduced into the preparation of benchmarks for parallel and vector computers.

When studying the characteristics of programs, a static analysis is made of the measurements that can be performed at the source code level. This involves a study of the form of program presented to the compiler, for example, finding out how much work which parts of a compiler do and how to optimize them. Dynamic measurements relating to (i) source program features indicating how many assignment statements are executed at run time, what proportion of "IF" statements have true conditions etc. (ii) language features; for example, which features incur the most overhead during compilation and execution; (iii) hardware usage, which may be either use of the order code for which instructions are executed most frequently or simply use of the hardware.

3.4.1 Setting the Benchmark Design Objectives

Steps have to be taken and decisions have to be made when undertaking a benchmark. The first important thing is setting the benchmark objectives since there is no point in undertaking a benchmark if the reasons for doing it are not known. It is very important to identify the real objective for such a task.

The root of the problem here is about measuring the relative performance of the currently implemented and future vector and parallel supercomputer processor architectures by using a standardized benchmark technique for numerical

scientific work [35,42]. What the author is concerned with is the naked processor architecture with sufficient store to run relatively large scientific application programs.

It is emphasized that the discussion below is not a description of the benchmark implementation but rather an illustration of the technique.

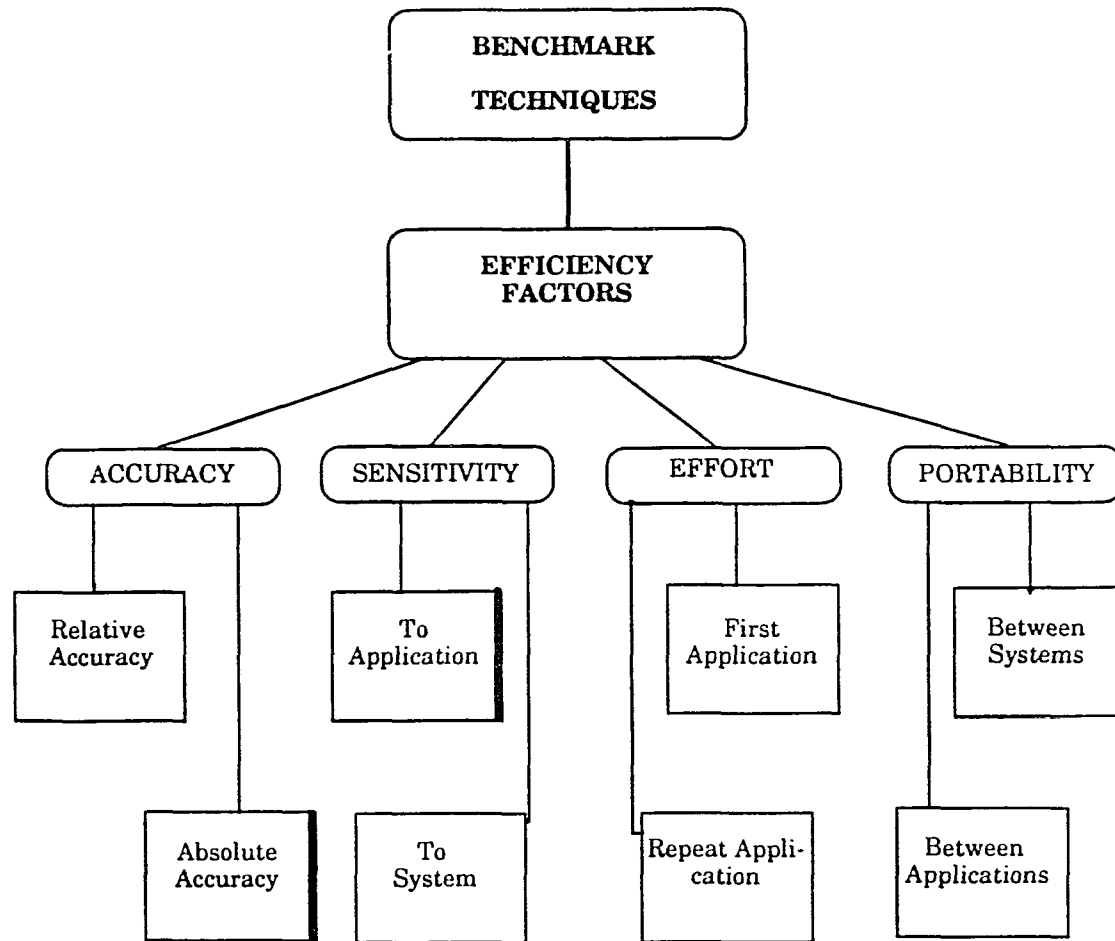


Figure 3.6 Benchmark Efficiency Factors

3.4.2 Factors that influence Benchmark Techniques

Four factors have been identified which form a basis in considering the design of benchmark techniques. The important point about these factors is that their relative and absolute importance is highly dependent on the type of measurement exercise to be performed, on the application to be modelled and on the time scales involved. These factors can be envisaged as in Figure 3.6 and are discussed below:

(1) Accuracy

This is often the most important factor, but not always, since accuracy can be defined in many ways according to user's objectives. Inaccuracy may arise from many sources. A benchmark specification may lack precision such that measured variation produced by programmer interpretation and ability is greater than variations in a machine's performance. Whether or not a particular source of inaccuracy is significant is highly dependent on both context and application.

(2) Accuracy

It is reasonable to expect a benchmark to be sensitive to changes in machine architecture because these are likely to affect its performance significantly in relation to the target application. Conversely, where a particular

hardware feature is of little consequence to the target application, insensitivity is required. Similarly, sensitivity to application is required, Thus, the benchmark technique should be capable of reflecting the distinguishing characteristics of different applications into benchmark itself. It is equally important to aim for minimum sensitivity to extraneous factors, programmer dependency and specification ambiguity.

(3) Effort

Given unlimited resources and sufficient time a perfect benchmark may be constructed. However, in our world, resources and especially time, are strictly limited. Therefore, there is the need to eliminate waste and duplication which results in the production of benchmarks. In a nutshell, there is the need for a degree of standardization to encourage resources sharing in this expensive area of computer evaluation.

(4) Portability

In most cases a benchmark is going to be used on at least two machines; portability between machines must therefore be assured. Naturally, an important consideration is the amount of effort

involved in transfer. For a benchmark implemented in a common high level language, that is, a language that has been implemented on the machines of interest, the problem becomes manageable; however, this does involve a further variable in the comparative efficiency of compiler implementation.

Portability between applications infers a minimization of the effort involved in implementing a benchmark which reflects at least two different applications on the same machine. That is a more complex problem than man-machine portability, no simple solutions are available but light is shed in this area later in this chapter.

3.2.4.1. Benchmark Design Requirements

Consideration of the available techniques [29, 30, 31, 32, 33, 32] with reference to the application of large scale scientific processing on parallel and vector computers indicted the following requirements:-

- (1) A technique in which both machine and application can be varied.
- (2) A sensitive technique, capable of exercising those machine features which are important for parallel and vector machines evaluation.

- (3) A technique which provides a reasonable accurate power measure which has absolute as well as relative meaning.
- (4) A technique which provides results within a reasonable time scale, with small effort.

Consideration of these criteria implies that in lieu of either suitable parallel synthetic program standard or suitable application technique a new tool is required: a tool capable of meeting these particular criteria and time scales. This tool named the "Parallel Computer Benchmark Technique"(PACBET) is proposed and described below.

3.4.3. Parallel Computer Benchmark Technique (PACBET)

This technique is a variant of an application approximator and is no different from other benchmark techniques which employ some degree of approximation to produce results. The important factor is the establishment of a standard to ensure that machines are being compared in their ability to perform the same job even if the job is not identical to the target application [32,33]. The level of approximation employed varies between that of a simple instruction mix to that of a fully implemented application program. The choice between techniques depends on the acceptable level of approximation, considering the fidelity to which the benchmark portrays the target application. The Proposed Technique is described below.

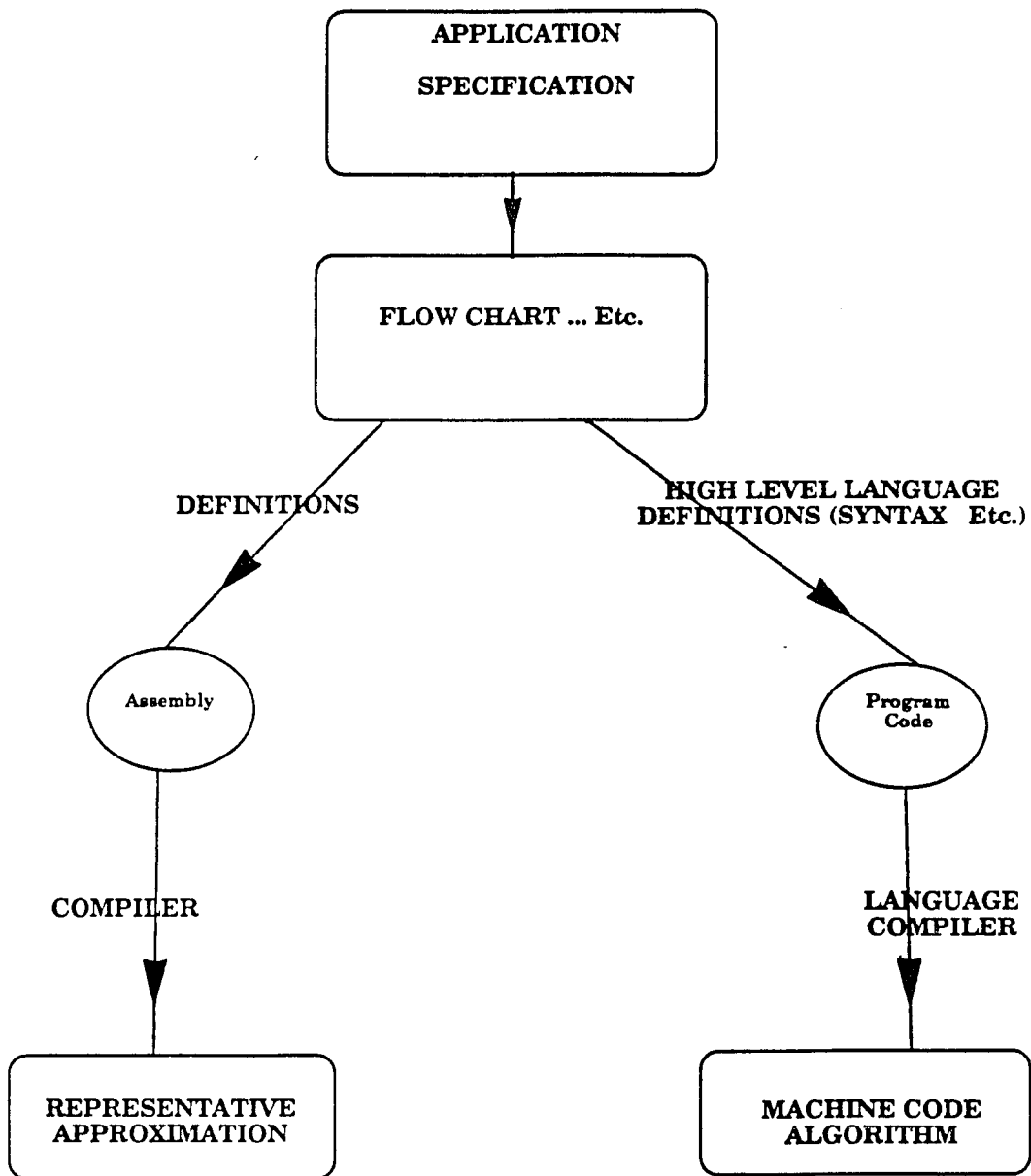


Figure 3.7 PACBET Technique Vs Actual Program Execution

3.4.3.1 The Proposed Technique

The Parallel Computer Benchmark Technique is best viewed in relation to its relation-program production using a high level language. That is illustrated in **Figure 3.7** in flowchart form. In both cases , it starts from a reasonably detailed specification of the target application. The right of the diagram portrays a high level language program produced by translating the flow charted algorithm with reference to the language facilities and definitions. On the left, the algorithm is approximated by choosing suitable primitives to reference the primitive definitions, assigning weights to them and other information dictated by the algorithm. Thus a primitive assembly is produced. This is further explained below.

The high level language program is processed conventionally by a compiler thereby producing a machine language version of the algorithm which is then executed on the machine and its performance characteristics measured. The primitive assembly is similarly processed by a primitive compiler; but the output far from being an executable algorithm, is merely a representative approximation of the original application, from which performance data can be obtained.

3.4.3.2 Definition of Primitives

The Primitives are entities which perform useful and definite programming tasks. They might be defined in rigorous high level language syntax, but for the present, it suffices to specify them in English for ease of explanation. These primitives are tied in with the weight parameters which are characterized with respect to the application problem in question. Some of the primitives which might be found useful in approximating application programs are shown in **Table 3.2**. In addition, "Primitives" which define operating system functions are necessary. These may map directly into an available operating system call as shown in **Table 3.2**. The format for using each primitive depends entirely on the logic of the algorithm used. The primitive number is first specified followed by the weighing and the relevant information string, if applicable.

The primitive definition is extremely important. Ambiguity and loose specification must be avoided at all costs since the code for a particular machine benchmark is produced directly from the primitive definition. Any inaccuracy introduced at this stage invalidates the whole exercise. To quote a specific example, the definition of the list search primitive might be defined as follows:-

PR04: List Search Definition

- (1) Search through a list of fixed sized records
until the first word of a record matches the

value found in a given location.

- (2) The list is one way linked, and a zero pointer indicated the last record on the list for which the program test.
- (3) The number of records on the list is P4:1 and is available in a given location.
- (4) On average, match is made on the P4:2th record.
- (5) The start of list address is found in a memory location
- (6) If necessary, assume that the list record is five words long including the pointer word, The record words are arranged in contiguous locations.
- (7) Execution time should be minimized.

Limiting the number of primitives represents a trade-off between minimizing programmer implementation dependence and accurate application portrayal. In the same vein, by restricting the number of primitives restricts available choice at the application flowchart level to primitive assembly translation stage; this necessarily increased the level of approximation but may reduce the amount of effort required in constructing the benchmark.

This is important because the primitive definitions are geared towards low level production of code which ensures

that the **effort** is minimized, **accuracy** is increased and **portability** is encouraged.

3.4.3.2.1 Statistical Description of the Workload

In order to examine the way in which a system reacts to architectural enhancements and changes in workload characteristics, a controlled environment must be produced. This means that the evaluation must be repeatable and that a steady state condition must exist. This is dependent upon the validity of the workload definition and the assumptions made in setting up the application programs.

The key to successful performance measurement is an adequate statistical description of the workload in the chosen high-level language. By this we mean that one has to know, for instance, the number of procedure calls, the number of array elements and vector accesses, number of stores, number of scalar elements and instruction fetches, number of conditional jumps, number of jumps, etc. Thus, a really detailed description of the use of the language must be made. Each particular source language is then translated into a single instruction or primitive so that they can be directly related to the use made of the features in the high-level language.

With this facility, statistics can then be collected from

the application program, where it becomes apparent that some features of the high-level language are used a lot more heavily than the others and, therefore they are needed to weigh those statements to produce a proper mix of the approximated application program.

3.4.3.3. Primitive Assembly Production and Compilation

Consider the flowchart segment shown in **Figure 3.8**. The Primitive assembly code which is produced from this flowchart, or the assumption that traffic flows down the right hand path of the diagram with a 70 % probability, is as follows:-

```
01,  1,  1, 100,
02,  1,  1,  50,  0,
05, 0.3, 100,   1,
03, 0.7,   0, 1000,  1,
00,   1,
```

The zero primitive number signifies the termination of the assembly and allows an overall weighing to be applied. This assembly can then be input to the primitive compiler for code generation and other processing, the entire procedure being exemplified in **Figure 3.9**

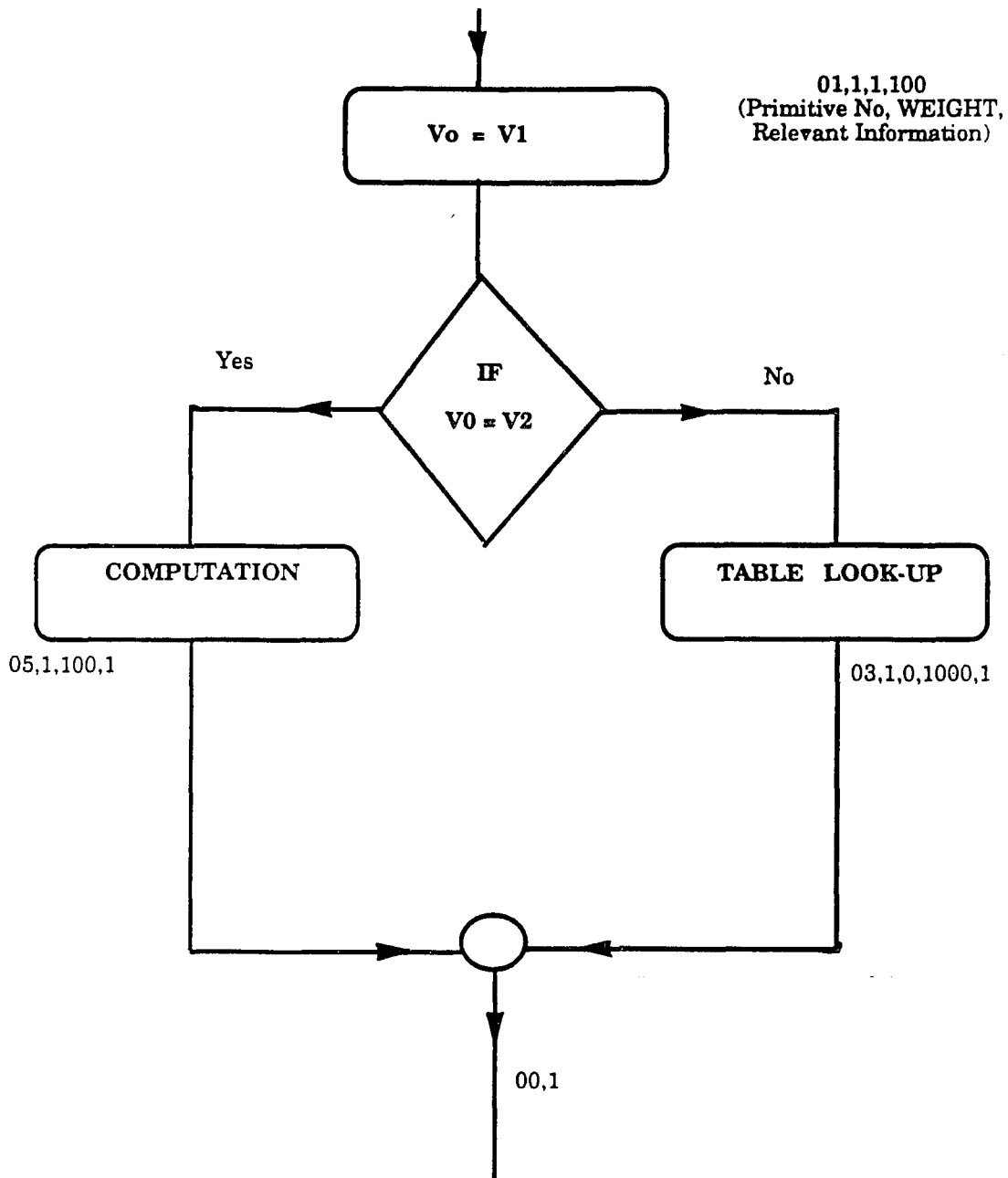


Figure 3.8 Flowpath for Primitive Assembly Production

The format analyzer merely detects errors, applies overall weighing, traps end of assembly conditions and so on. The primitive definitions can be used to produce code for a benchmark program to run on a machine or alternatively derive performances formulae based on the measured or calculated run time, etc. of each primitive code. Where an implementation does not generate code and for practical reasons, the performance formulae may be estimated from design parameters. This mode of operation has the advantage that the technique can be used as a design tool, that is, the performance of the machine can be estimated before the machine has been produced. Performance evaluation without recourse to practical tests on a machine, though economically advantageous, does introduce further inaccuracy, it is hoped that this technique will be tested on the hardware being implemented for the microprocessor derivative of the **NetVecX** architecture.

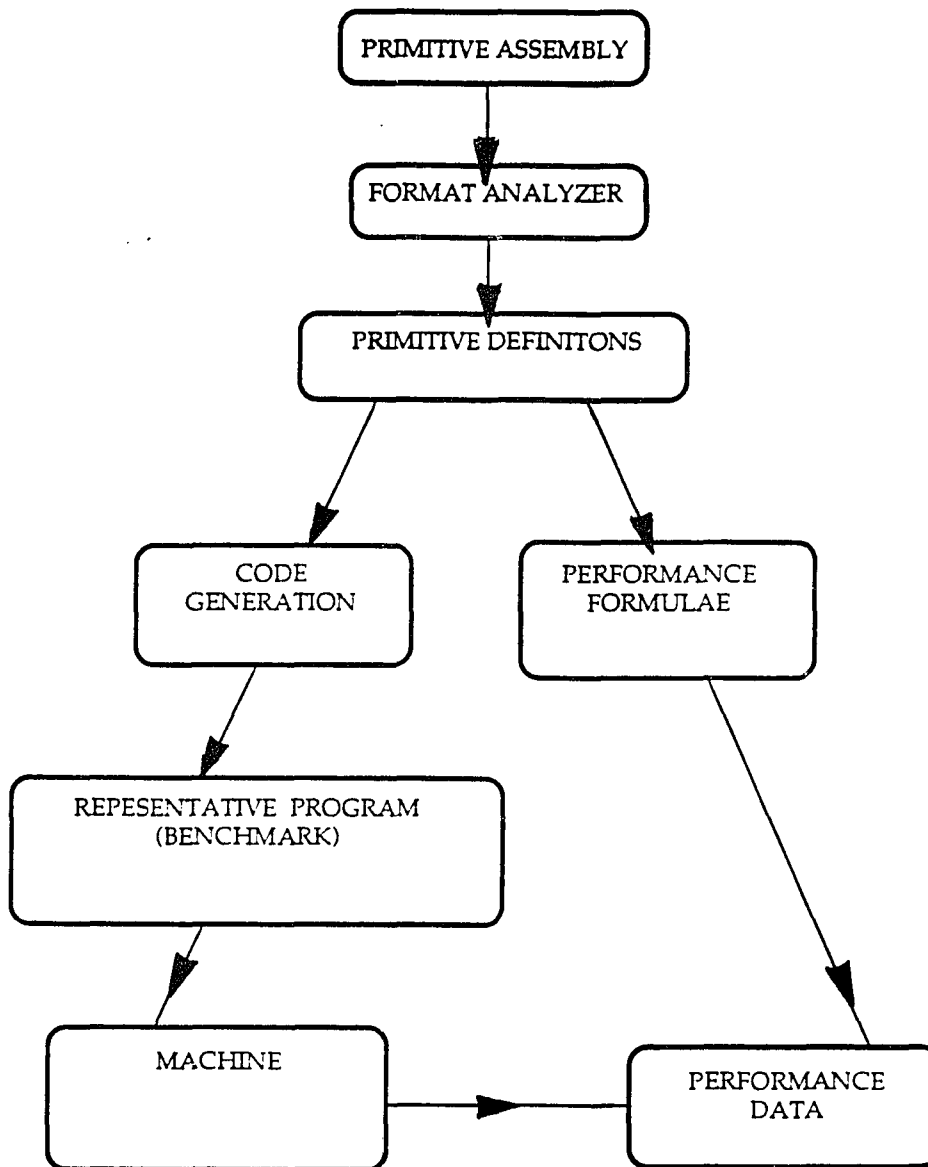


Figure 3.9 Primitive Assembly and Compilation

The performance data one expects to be output from an implementation of the Parallel Computer Benchmark Technique includes-:

- (1) Execution time
- (2) Memory Utilization
- (3) Average Instruction time
- (4) Storage requirements, etc.

3.4.4. Evaluation of the Technique

The PACBET technique rates well against the efficiency factors established in 3.4.2. The effort which may be expended in applying the technique may be minimized in the production of minimal instructions from relevant primitive definitions. A minimal programming effort. The effort expended in producing a primitive assembly from the application flowchart is by no means trivial but production times may be less in producing a working application program.

A primitive assembly, once produced, can be applied to any machine which has a primitive compiler with no further effort required. Similarly, a machine which has a primitive compiler can be examined in relation to any application for which a primitive assembly has been produced, again with little or no further effort.

Relative accuracy is assured since the application, as

represented by primitive assembly, is identical for each machine evaluated, so this technique serves as an effective standard. The low level of code production envisaged increases accuracy by minimizing programmer dependency, but this is only at the expense of a high approximation level stemming from limited primitive choice at the primitive-assembly production stage.

The sensitivity of the technique is reasonably high. The level of complexity of the primitives, though fairly low, is sufficient to take advantage of useful machine features. Sensitivity to application derives from the ability to approximate to the task to be performed at a detail level. To demonstrate the technique's sensitivity, **Figure 3.10** has been constructed.

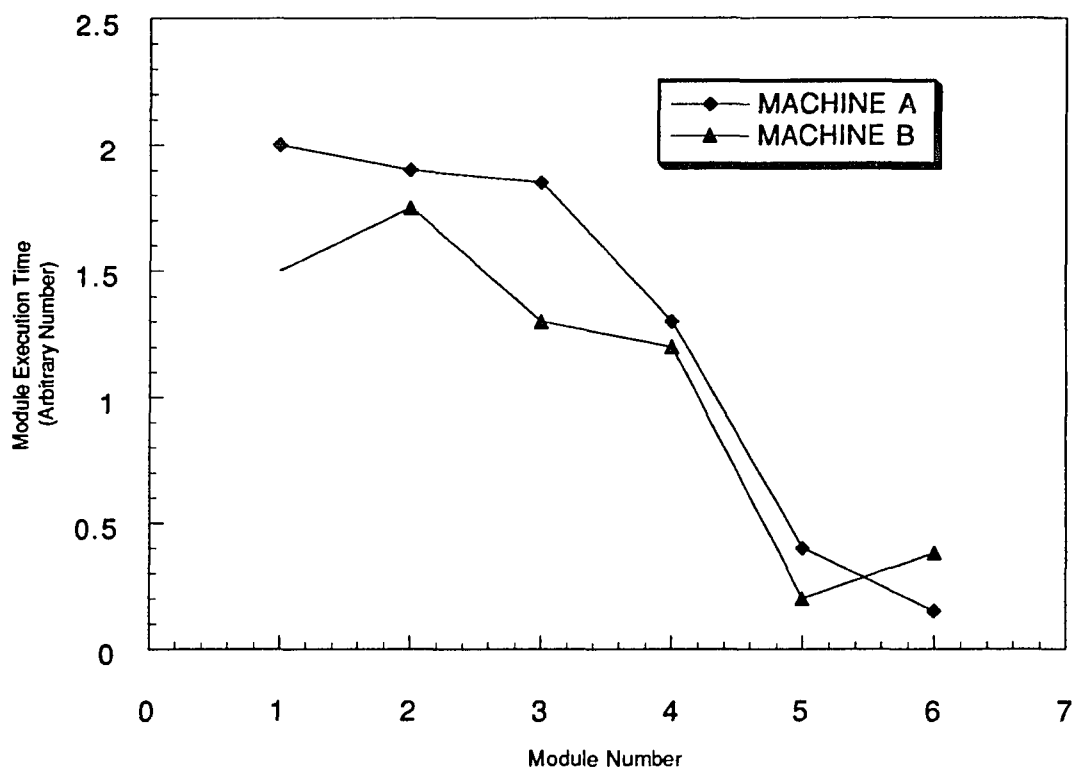


Figure 3.10 Techniques Sensitivity

Each module number relates to a sub-program within the overall application program. There is considerable variation in relative abilities of the two comparable machines to perform work on each module. In particular, module 6 reverses the trend of machine A's apparent superiority over machine B. This demonstrated the sensitivity useful for specialized tasks. In the same manner that the technique detected the superiority of machine B in performing the work of module 6 and its inferiority in executing the other modules, it can discriminate between the relative abilities of machines in performing the work of different applications.

Portability between machines relies heavily on the standardization of primitives. Production of a machine's primitive compiler is a reasonably trivial matter and, this done, an existing application as represented by a primitive assembly, can be arranged without involving the added variable of compiler efficiency, which results inevitably from the use of high level language benchmarks. The same primitive compiler can be used to translate many different applications thus allowing the same machine oriented effort, that is , the coding of primitive definitions to be re-used for a number of separate application studies.

CHAPTER 4

NetVecX Architectural Specifications

In this chapter the architectural specifications of the vector processor referred to as **NetVecX** are presented. The rest of the chapter describes the architectural frame from which any member of this class of vector computer system configurations can be derived. It also describes the modes of vector processing and discusses the performance characteristics of the proposed architecture.

The design presented here is based on the notion of a vector processor involving a multiplicity of vector processing units with full floating-point facilities. It has been recognized that a number of vector processing units can be controlled conveniently by a single relatively conventional scalar processor which uses software to map multi-dimensional array operations into a set of parallel vector operations. The advantage of using multiple vector units is that the mapping of multi-dimensional arrays onto the hardware only involves a significant constraint in one dimension, rather than two in an array processor, since vector lengths can be orders of magnitude greater than the number of processors. This structure is then capable of giving very high

performance, e.g., in Gigafllops, on some large problems. The vector processing units are designed to be relatively simple in structure and dedicated to processing vector operations given the relevant descriptors of the source operands and the vector function. The general characteristics of the **NetVecX** architectures are described below.

4.1 System Architectural Frame

Figures 4.1 (a-c) illustrate the block structure of the architectural frame which defines the class of vector computer architectures under consideration. Generating a specific system necessitates the definition of the characteristics of the class of architectures in question. The class in question will consist of the following:-

- (1) A central processing unit (Central Control Unit), which consists of floating-point arithmetic unit, address manipulation module, control logic, local storage, and central interface registers.
- (2) A number of vector processing units, each of which consists of a set of interface registers, an address manipulation module, vector floating-point arithmetic units and local memory.
- (3) System and data buses.
- (4) Main memory subsystem (MMS).

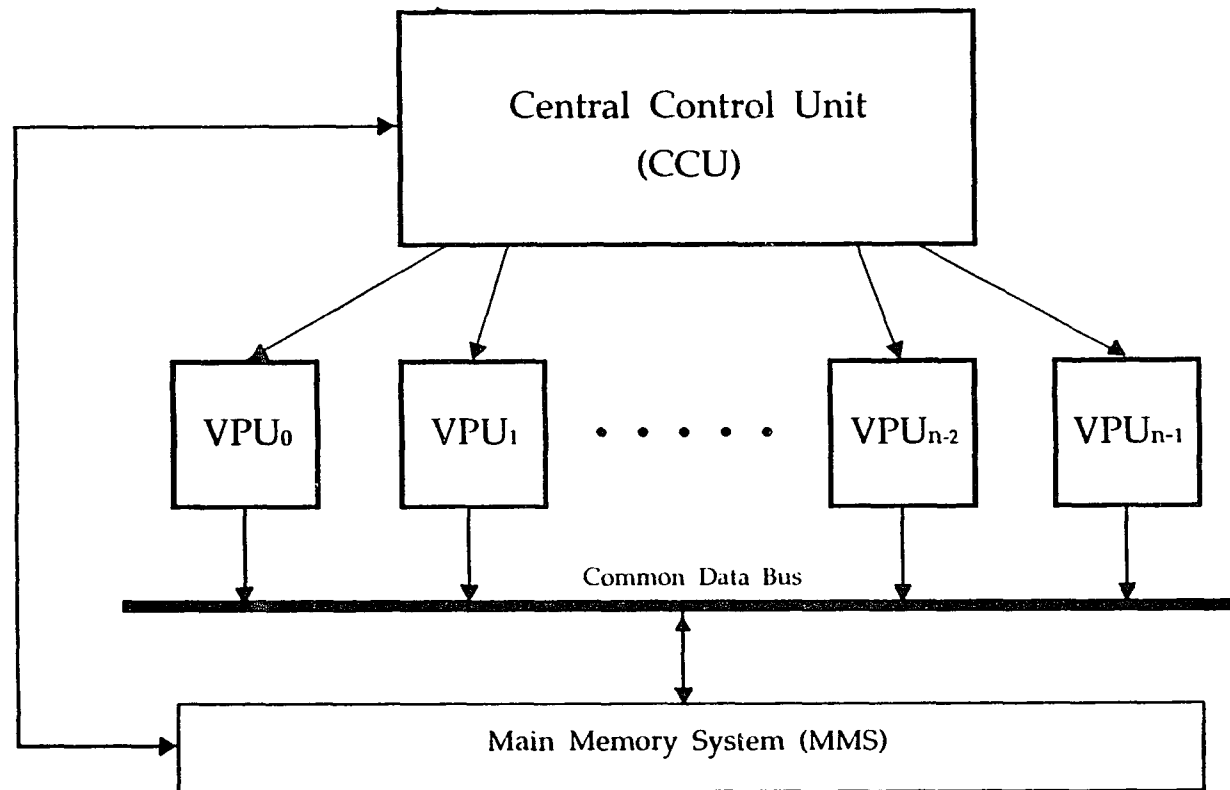


Figure 4.1(a) NetVecX Block Diagram

Floating-Point Arithmetic Unit
Address Manipulation Module
Control Logic
Local Storage
Central Interface Registers

Figure 4.1(b) Central Control Unit (CCU)

Interface Registers
Address Manipulation Module
Vecto Floating-Point Arithmetic Unit
Control Logic
Local Storage

Figure 4.1(c) Vector Processing Unit (VPU)

This architecture may be classified as Multiple Single Instruction Multiple Data Stream (MSIMD) system in Flynn's terminology. The scalar unit and the vector processing units consist of maximum of four addresses in the instruction format depending on the multi-operation vector functions. This encourage the definition of a new vector operations to take advantage of the new architectural features. The architectural frame provides for variable data and instruction word lengths to account for any particular system configuration which might be derived from it.

The interrelation of the various units and the corresponding bus connections are shown in **Figure 4.1 (a)**. The vector processing units (VPUs) share a common data bus and each has its own private local memory for code. In addition, each VPU contains a set of interface registers which contain information for initiating a VPU to run autonomously. The Address Manipulation Module (AMM) is used to generate the origin of the source vector operands for subsequent fetch and processing. The floating-point arithmetic units perform the various operations which they are called upon to execute. The addresses generated by the VPUs are used to access the common main memory via the common data bus.

The scalar unit is the focal point of the whole **NetVecX** architecture since it controls the scheduling of the VPUs and

the main memory. It is through the scalar unit that this notion of vector processing is implemented. The scalar unit drives the VPUs by passing descriptor and vector orders through the main set of interface registers of the scalar unit to the corresponding set in each of the VPUs. Once set up, a vector unit decodes its instruction and proceeds autonomously until it completes its vector operation, when finished it signals the scalar unit its readiness for the next instruction. The scalar unit itself executes all scalar instructions and organizational orders in order to control the correct logical flow of the executing program. In order to enhance the performance of the system, virtual memory is controlled by a separate autonomous hardware. Other important characteristic which increase the performance of high performance systems are discussed below in relation to the proposed **NetVecX** architecture.

4.1.1 Instructions and Addresses

One of the most important characteristics of an architecture of a computer is the number of addresses contained in its instruction format. Since most arithmetic operations generally require two source operands and produce one result, a three-address instruction format would seem natural. However, a number of arguments have been raised against this arrangement, and decisions about the number of addresses to be contained within one instruction are generally

based on the intuitive feelings of computer architects in relation to economic considerations, expected nature of the implementation, and the type of operand address and its size.

The **NetVecX** architecture proposed is intended to be used in solving large scale scientific application problems in a "reasonably" short time; and in order to achieve this end, a high degree of parallelism is introduced into the design of this vector architecture. An instruction set and processor maintaining the illusion of strict serial execution of instructions is proposed. A three-address instruction format provides this possibility, since successive instructions can refer to totally independent input and result operands, and therefore vector instructions are assigned to vector units and scalar instructions executed in the scalar unit. This processing technique naturally leads to parallel processing and high performance.

The order code of the central processor of the scalar unit does not include any peripheral handling instructions, since all peripheral operations are independently controlled by a front-end G-machines in an open network environment. These machines organize data transfer between main memory and input/output and bulk storage devices attached to the peripheral channels. This releases the central processor of the burden of dealing with input/output, a further

contribution towards high performance.

4.1.2 Memory System Interleaving

The technique of memory systems interleaving is essential in contributing to increased performance in the **NetVecX** and most other computer systems discussed in **Chapter 2**. Until parallel arithmetic hardware was introduced in the late 1950s, the speed of random access storage devices was a limiting factor on processor performance. The time for a floating-point addition in Mercury, for example, was 180 microseconds, with a core store cycle time of 10 microseconds. In **Atlas**, on the other hand, the time for a floating-point addition was 1.6 microseconds, with a core store cycle time of 2 microseconds [43,45]. Thus, in the absence of some special technique, the time required to access both an instruction and its operands would have amounted to 4 microseconds, leaving the floating-point unit idle for much of the time. Part of the solution to this problem was to overlap arithmetic and store accessing operations for successive instructions. This was achieved in the **Atlas**, where by having a store made for four independent stacks with two-way interleaving of addresses, the execution time of computational instructions was improved by a factor of two.

In the **CDC-6600**, the multiple functional units meant instructions could be issued at the much higher maximum rate

of one every 100 ns. Thus, a much higher level of store interleaving was required to sustain the instruction execution rate. In order to minimize the effects of store clashes on the overall performance, the 60-bit word main store of the CDC-6600, and also the 7600, are 32-way interleaved. This implies that in random accessing, there is a high probability that any one request is to a free storage bank, and will be serviced immediately. Much of the benefit of this arrangement comes not only from interleaving per se, but also from the fact that the store is made of large number of relatively small independent modules. The interleaving of these modules becomes more important when many successive accesses are to sequential addresses, during instruction accessing, for example, and more important during the processing of long vectors. Interleaving has been extended to higher levels in the CYBER-200 series of vector processors. The 32-bit word banks of the CYBER-205, for example, are 256-way interleaved and evidence for the necessity for such a high level of interleaving comes from the fact that reducing this interleaving to 128-way can cause a reduction in performance of nearly 40 percent for some types of operation.

Another aspect of memory hierarchy, quite different from that used in the Atlas, involves cache storage. First introduced into the IBM System/360 series with the Model 85 [45] and also used extensively in other IBM machines [43].

These are provided solely as a means of improving performance. The locality of store accessing exhibited by most computer programs means that most processor accesses are handled by the cache which most often operates at the processor rate rather than the main storage cycle time. This evidence suggests that any high performance derivatives of the proposed architecture must incorporate cache storage in the vector processing units in order to eliminate the disparity between the processor speed and store accessing speed, and thereby lead to further improvements in performance.

4.1.3 Pipeline Concurrency Principle

Another technique which increases the performance of computers is pipelining. In the execution of a single instruction, various activities such as instruction accessing, instruction decoding, operand accessing and arithmetic operating. Using separate units to carry out these activities, their operations can be overlapped to give an increased rate of completion of instructions. This technique known as "Pipeline Concurrency" has been successfully employed in many designs in both arithmetic and instruction processing units as typified by the pipeline arithmetic unit of the **TI-ASC** [99] and the **MU5** instruction pipeline [37,75] respectively. A combination of these two examples implemented in the proposed architectures further contributes enormously for an increase in performance.

For a successful operation of a pipelined arithmetic unit each instruction must reference at least two and preferably three operands. This arrangement surely requires complex control mechanisms to resolve dependencies between successive instructions. An example of this control technique is the scoreboard used in the CDC-6600 [95]. The ideal arrangement is for one instruction to cause two independent operand streams to be combined in the arithmetic unit to form a third independent result stream. This is achieved in the vector processing systems of the CYBER-205, the CRAY-1 and the TI-ASC vector computers. For a three-address instruction architecture, such as the one we are proposing, a variant of the pipelined arithmetic unit of the TI-ASC needs to be implemented in a high performance version of the architecture.

4.2 NetVecX Architectural Derivatives

This section examines three NetVecX configurations derived from the architectural frame described in Section 4.1. It also describes the hardware characteristics of a microprocessor-based derivative to be implemented as a natural future extension to the current efforts. Strengths and weaknesses of this system are identified. To isolate the key features of this architecture, certain elements less essential to the discussion such as data types and input/output are briefly discussed.

4.2.1 Microprocessor-Based NetVecX (NetVecX/M)

The main components of microprocessor based **NetVecX/M** architecture are identified and the operation of such components are described. The basic **NetVecX/M** processor organization, illustrated in **Figure 4.2**, incorporates the following functional units. A microprocessor-based scalar unit which initiates actions in each of the 16 vector processing units (VPUs) by writing into the set of interface registers as shown in the figure. The scalar unit selects a free vector unit by interrogating the contents of a status register giving the current state of all the VPUs. After a VPU is selected, the vector instruction is sent to it via the I-Bus for execution. Each VPU has its own local program store which contains a procedure of subroutine codes for representing a set of vector operations which are used to execute the vector instructions. At the completion of a vector operation, the corresponding bit of the VPU in the status register is reset and the VPU is made ready for another vector instruction.

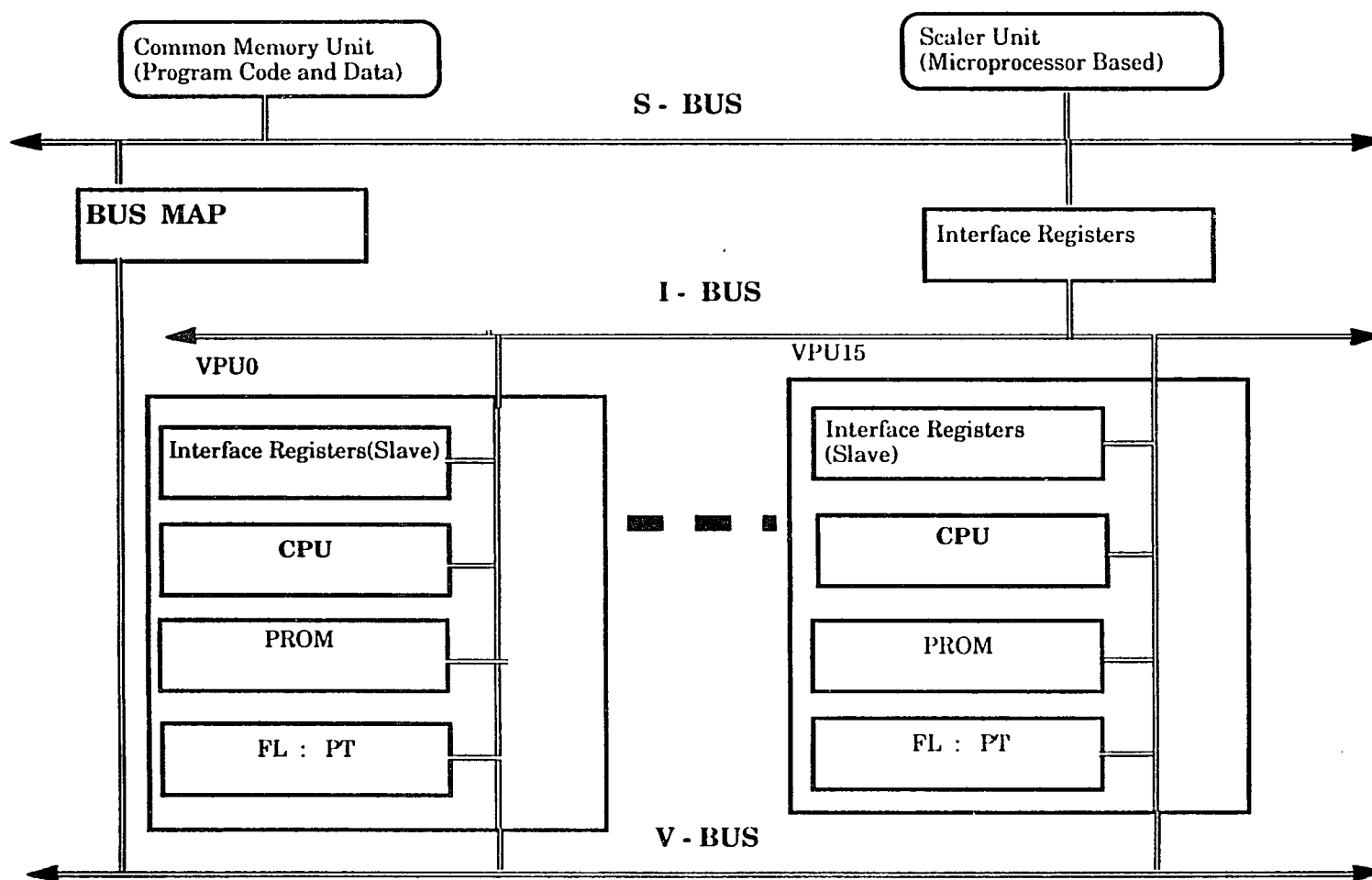


Figure 4.2 Microprocessor Based NetVecX Organization

The separation of the buses is intended to improve performance of the system as a whole. The S-Bus transfers from the S-Store while the I-Bus is used to send vector instructions to each of the VPUs. The V-Bus on the other hand supplies operands to all the VPUs via the Bus-Map which allows bi-directional access to the memory unit.

The major difference between **NetVecX/M** and other functional pipelined units of some current computers is its microprogrammability. Each of the modules being a general purpose microprocessor with each vector operation defined by a microprogram. A copy of each such program is stored in every microprocessor for vector processing. The gain here is that choice of vector operations can be made to fit the needs of the user and can be altered for different applications or when optimized microprograms are developed. The types of vectors such as dense, sparse, Boolean,..., etc. are distinguished by the mode or type part of vector instructions chosen by the system implementor. The microprogram vector units typically require several tens of microseconds per operation, whereas memories can currently be constructed to deliver words at intervals of tens of nanoseconds. Thus, several hundred floating-point microprogrammed vector processing units could in principle be supported by a single memory unit. Furthermore, the set-up time for a vector unit would be of the order of 1 per sec. at the most, so that, even

with 100 units, vector lengths measured in single figures would lead to efficient use of the vector units. A prototype **NetVecX/M** is described in **Section 4.3**.

4.2.2 G-Machine Based NetVecX Vector Architecture (NetVecX/G)

A Vector processor of much higher performance than the **NetVecX/M** can be configured from the architectural frame of reference. This may be based on a conventional general purpose computer shown diagrammatically in **Figure 4.3**. The mode of operation of the **NetVecX/G** is similar to the **NetVecX/M** except that the scalar unit and the vector processing units owe their basic structure to a conventional general purpose computer as illustrated in **Figure 4.4**. The organization of the **NetVecX/G** is made different than that of the **NetVecX/M** by incorporating an extra memory unit (D-Store) for holding data to be accessed mainly by the vector processing units. This is intended to improve performance by putting the program code into the S-Store and data into the main data store (D-Store). Another characteristic of relevance is that fast arithmetic units may be incorporated into the VPUs to speed up arithmetic operations.

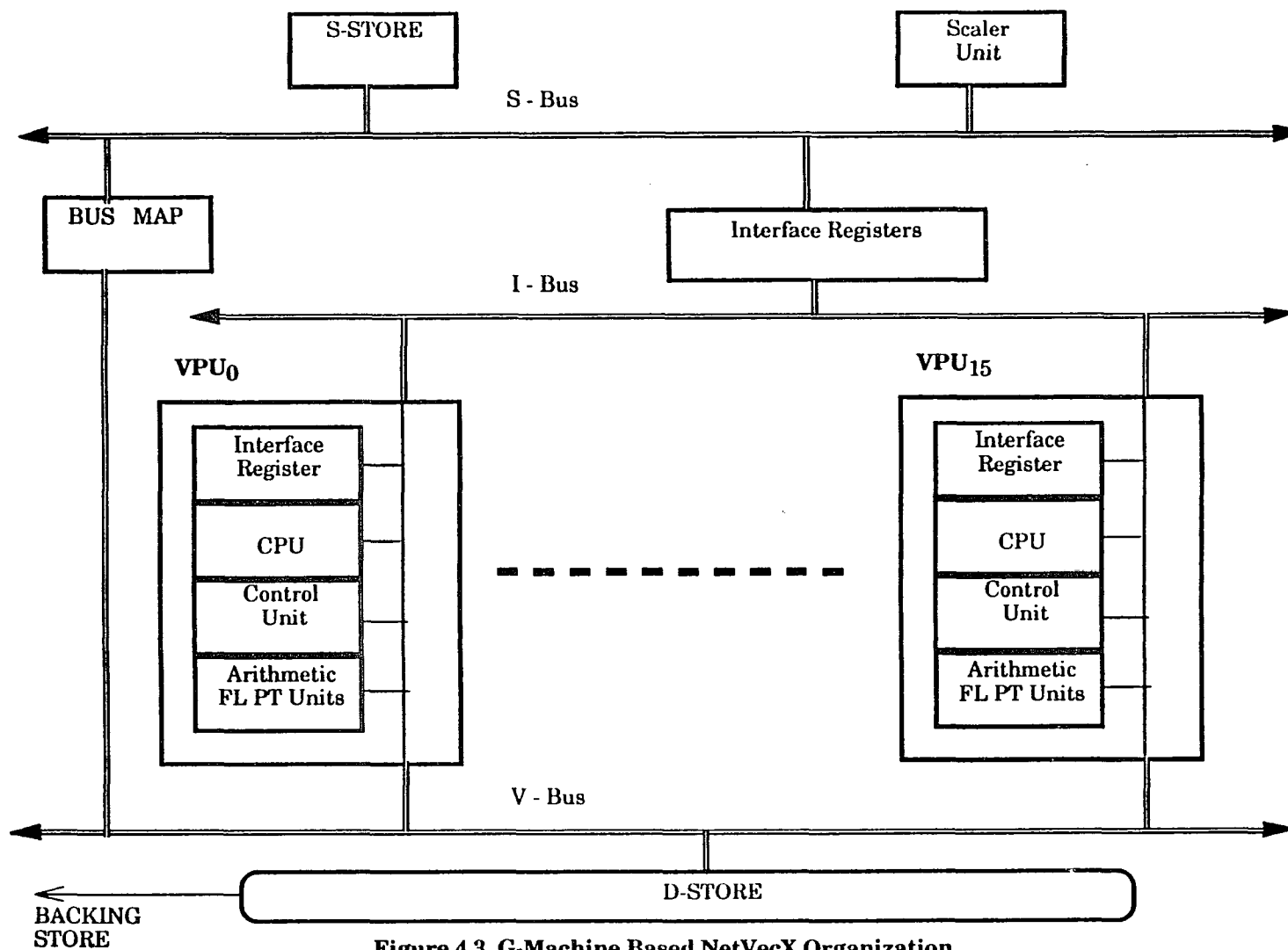


Figure 4.3 G-Machine Based NetVecX Organization

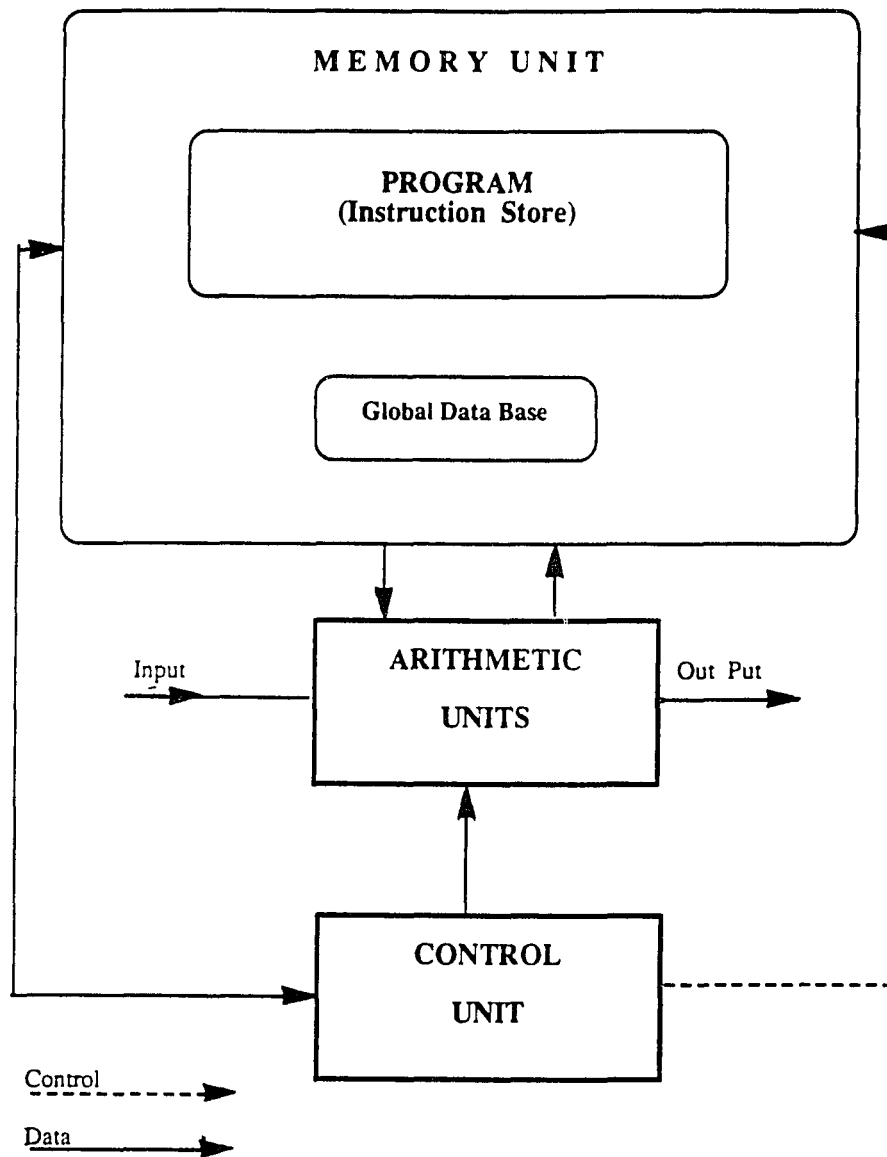


Figure 4.4 Basic Conventional Computer Organization

4.2.3 High Performance NetVecX Vector Architecture (NetVecX/HP)

The components of the **NetVecX/HP** are interconnected as shown in **Figure 4.5**, of which the basic architecture is similar to the **NetVecX/M** and **NetVecX/G** derivatives but with some major differences. The vector processor consists of 16 VPUs and a scalar unit which controls the VPUs. In order to improve scalar performance, in processing scalar operations, the scalar unit must be pipelined with a suitable number of stages depending on the number of concurrent activities per operation. The vector processing units communicate with the common main store (D-Store) and via the V-Bus whilst the Bus-Map allows bi-directional access between the S-Bus and the V-Bus. This allows the scalar unit access to the main data store and the VPUs access to the S-Store. Processed data may be input to or output from the D-Store or S-Store through the V-Bus and the S-Bus both controlled by the Bus-Map. The D-Store is supported by a very large backing store with a capacity in excess of say, 256 million 32-bit words to accommodate very large data for the application problems mentioned in **Chapter 3**.

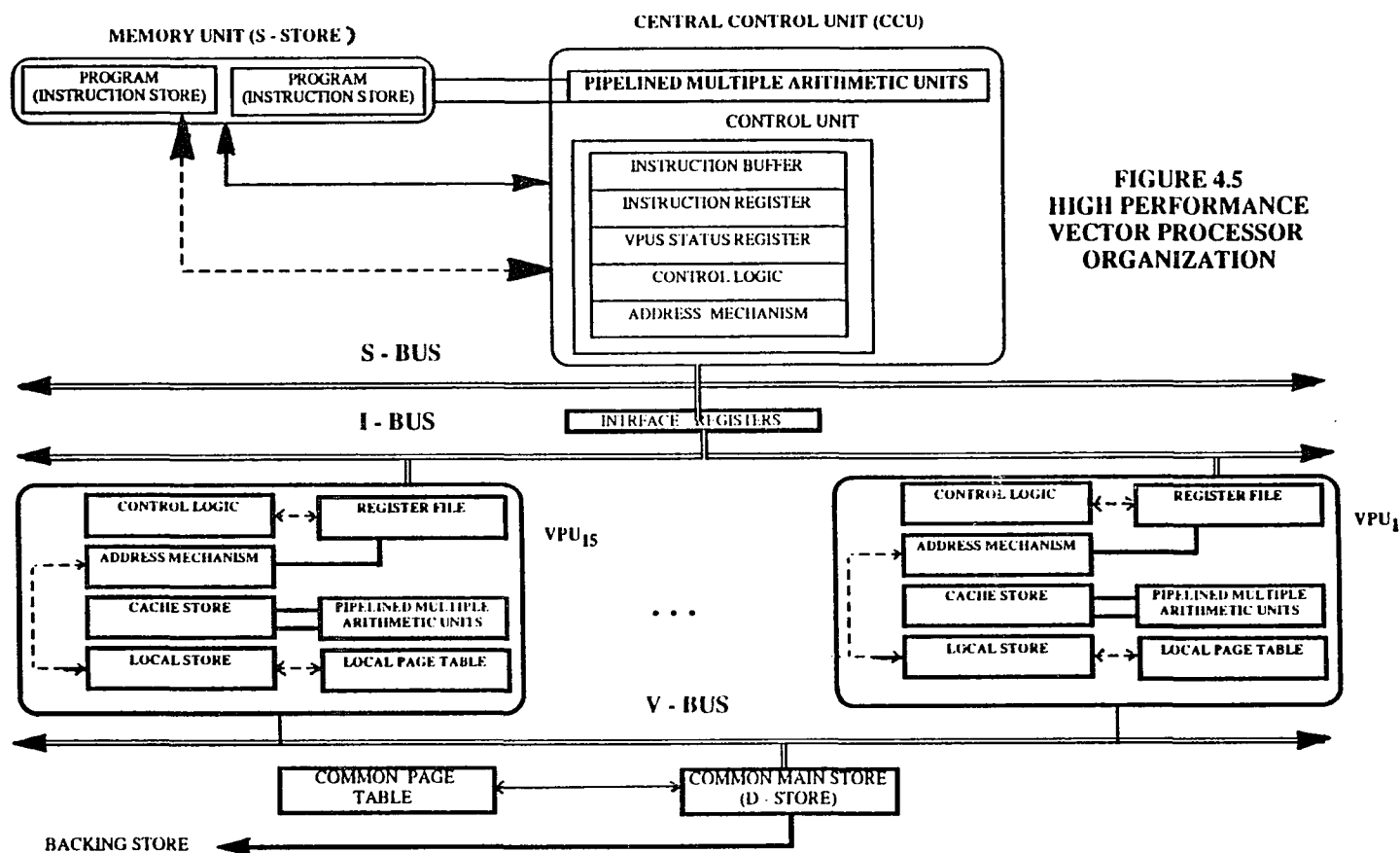


FIGURE 4.5
HIGH PERFORMANCE
VECTOR PROCESSOR
ORGANIZATION

Since this vector processor is intended to solve problems involving large linear equations and linear programming, and to achieve high operational speed in doing so, its performance should at least match that of the CRAY-1, CRAY-X-MP and the CYPER-205 in solving certain problems. More important, since the CRAY-X-MP has a clock period of 9 ns at present, the **NetVecX/HP** clock period should be at least less than the 9 ns of the CRAY-X-MP in order to achieve the desired operational speed. The following general specifications are guidelines for this design:-

Clock period ≤ 10 ns

No. of VPUs is 16

Memory bandwidth ≥ 800 M (32-bit)/sec.

Virtual store capacity $\geq 2^{32}$ (32-bit) words

D-Store capacity ≥ 16 M (32-bit) words

S-Store capacity ≥ 4 M (32-bit) words

Cache store capacity ≥ 2 K (32-bit) words

These figures are a product of an extensive experimentation with a discrete simulation model of the general architecture.

As shown in Figure 4.5, each vector processing unit contains a descriptor based address generation mechanism, a local store with its local page table, a cache memory and a multiple arithmetic unit (MAU). A typical MAU may consist of several operating pipelines such as an adder, a multiplier, a divider, shifter etc. which may operate simultaneously for

higher improvement in performance. The address origins of vector operands are supplied by the scalar unit (the Central Control Unit) to each VPU for processing whilst the scalar unit itself processes scalar operations. The maximum operating rate of the **NetVecX/HP** is approximately estimated as 0.8 gigaflops and about 1.6 gigaflops when the adder and multiplier work in parallel for certain problems.

In accessing vectors, the information passed to the interface registers describes the characteristics of a vector in terms of its origin address, vector length, increments, subscripts etc., see **Section 4.3**, which the VPUs access for vector processing.

In order to reduce the accessing pressure on the D-Store and the local stores, a cache store is put between the Multiple Arithmetic Units(MAUs) and the local store in each VPU to reduce the memory contention problems and therefore lead to an increase in performance of the overall system. In this mode, all arithmetic operations are between the cache store and the MAU and local store in each VPU. Thus source vector operands are sent to the MAU from the cache stores and result vectors loaded into cache stores from the MAU. The cache stores may contain variable length vectors depending on the vector operation. There are 32 current address registers and therefore up to 32 current vectors in each vector cache.

Since a vector operation may involve three vectors e.g.

$$V1 = V2 * V3 \quad \text{or}$$

$$V1 = V2 - (S * V3)$$

The cache store should contain at least three vectors (4 chosen for future extension and as multiple of 2). Thus the number of resident current vectors is between 4 and 32. To ensure that at least 4 vectors can occupy a cache store, for a vector longer than one-fourth of the total capacity of the cache store, i.e. 512-bit words, the cache therefore acts as a buffer. On loading the first 512 words of a long vector into the cache vector, the first element is removed from the cache store after being processed, and the next element fetched from the local store and placed at the end of the 512 word queue in the cache store. This operation is repeated until all elements of the long vector are processed. In this way long vectors can be processed via the cache store even though its capacity may be limited to 2K 32-bit words. However, in such circumstances, the burden on the local store is very heavy because it has to work at its maximum bandwidth, nevertheless the overall system performance is improved.

4.2.3.1 Data Streaming Through The NetVecX/HP

Once the NetVecX/HP, connected as a node in the computer network and loaded with a compiled code for an application program, it proceeds autonomously to execute the program. In

the course of execution, the vector control unit of the scalar processor initiates vector actions in each VPU by writing into the set of interface registers to be subsequently accessed by the selected VPU. This then proceeds autonomously leaving the scalar unit free to initiate further actions in lock-step fashion depending on the control sequence of the executing program.

The D-store, local stores and the cache stores constitute a one level memory hierarchy. In processing a vector operation in a VPU, its address origin is associated with the 32 current origin addresses of the current vectors in the cache store. An equivalence indicates that the required vectors is in the cache store and may be accessed from it. A non-equivalence implies that the vector does not reside in the cache store and therefore the cache store must be updated. A currently resident vector is therefore rolled out to make room for the new vector. It may also happen that several old vectors may be rolled out if there is limited space for the incoming vector. The new origin address is then substituted for the old one(s). A check is then made to determine whether the pages (or blocks) containing the elements of the vector are in the local store of the VPU. This involves the calculation of address of the elements which are associated with the page table of the local store. If they exist, the elements of the vector are copied into the cache store from the local store; otherwise

the page table in the D-store is referenced to find out if the required pages are in the D-store. On a match these pages are then transferred into the local store in which they are required. The new vector may be used for arithmetic operations as soon as its first element arrives at the cache store from the local or d-store. It does not have to wait until the whole vector has been loaded into the cache store.

4.3 NetVecX MC68000 Microprocessor-Based Implementation

This section describes the main architectural feature of a prototype NetVecX vector processor to be implemented as an anticipated future extension to this thesis. The hardware components are connected as in Figure 4.2 in which the 16-bit MC68000 microprocessor is the major component. A brief description of the main components are presented, followed by a discussion of some of its architectural features.

4.3.1 Principal Hardware Components

The hardware components described here consist of the following:-

CENTRAL CONTROL UNIT (CCU)

The CCU is an MC68000 16-bit microprocessor-based [3, 49,76] computer which executes the code of the program held in the S-store. It is linked through the S-bus, the S-store, to the interface registers, through which information is passed to the vector processing units (VPU0 - VPU15), and to the D-

store through the bus map which links the D-bus and V-bus.

VECTOR PROCESSING UNITS (VPU0 - VPU15)

Each of the identical VPU's contains a 16 bit MC68000 microprocessor CPU, a PROM, a local store and a single chip floating-point execution unit. It also contains a set of slave interface registers in its address space which hold the information about the type of vector instruction being processed. The CPU executes code in the PROM appropriate to the type of vector order and the type of vectors identified in the interface registers at the time the VPU is activated. Each VPU maintains a BUSY/FREE signal which is transferred back through the interface registers so as to appear in the address space of the CCU.

The programming model of the CCU is based on the MC68000 architecture which provides the relevant scalar instruction set for the implementation of the control of the overall system. In this respect the MC68000 scalar instructions have been extended to include vector instructions for the execution of the vector operations.

15

0

Operation Word (first word specifies operation and modes)
Immediate Operand (if any, one or two words)
Source Effective Address Extension (if any, one or two words)
Destination Effective Address Extension (if any, one or two words)

Figure 4.6 MC68000 Scalar Instruction Format

The MC68000 scalar instructions are from one to five words (16-bits) in length as shown in **Figure 4.6**. The length of the operation and the type of the operation to be performed is specified by the first word of the instruction called the operation word. The remaining words further specify the operands of which they are either immediate operands or extensions to the effective address mode specified in the operand word. For a vector instruction, the operand word identifies the instruction vector then followed by the relevant information for vector processing.

INTERFACE REGISTERS

These interface registers are specifically designed and dedicated to vector processing. They are described in more detail in **Section 4.3.2**. Their individual functions have been permanently assigned so that it is advantageous to hardwire them to the CCU or other control logic units for fast operation without having to worry about access conflicts among them. These registers appear in the address space of the CCU and are used by the CCU to pass information to the VPUs. One of these registers is the number of operations register which when written causes the contents of all the interface registers to be copied from a master set accessed by the S-bus into a slave set accessed by the I-bus, and the VPU selected by the vector unit number in register to be activated. Through

the I-bus, the interface registers appear in the address space of each of the VPUs, which use the information contained within them to execute vector operations.

BUSES AND BUS MAP

The hardware components are interconnected by a number of address and data buses. The S-bus is a 16 data bit scalar (or Multibus) used by the CCU to access instructions and scalar values from the S-store. Since the CCU may use up to about 85 percent of the time on this bus a separate V-bus is used to link the VPUs to the main data store (D-store). The bus map is used to allow bi-directional access between the two buses because it allows the CCU access to the D-store and the VPUs access to the S-store. There is no direct communication between adjacent VPUs. The interface register bus (I-bus) allows each VPU to access the slave set of interface registers and carries decoded selection signals to and from the vector unit number in register to the individual units. It also returns the BUSY/FREE signal from each unit to the interface registers. Interrupt signals are also transferred between the CCU and the individual units.

MEMORY ORGANIZATION (S-STORE AND D-STORE)

The memory organization in this system is significant because it is geared to complex memory intensive applications. In meeting the needs of such applications, two stores, the S-

store, and D-store, are provided. The CCU may access instructions and scalar values from the S-store and the D-store holds data for vector operations which are executed in the VPU. The CCU is capable of directly addressing more than 16 megabytes of storage. Large programs are conveniently accounted for by the use of an external Memory Management Unit (MMU) which manages the address space of the user. The MMU provides translation and protection for the megabyte addressing range and thus provides the separation and security of user programs.

4.3.2 Interface Registers Definition

The Interface Registers which the CCU uses to initiate a vector operation in a VPU, fall into two categories, those concerned with defining an operation and those concerned with defining the unit which carries it out, see **Figure 4.7**. The registers have been defined according to the general form of the vector operations which are predominant in the algorithms underlining the application problems considered in **Chapter 3**. This is as follows;-

$$V_0 = V_1 \quad F_0 (V_2 \quad F_1 \quad S)$$

where

V_0 = destination or result vector

V_1, V_2 = source vectors

S = a scalar value

F_0 is a major function

F₁ is a sub-function

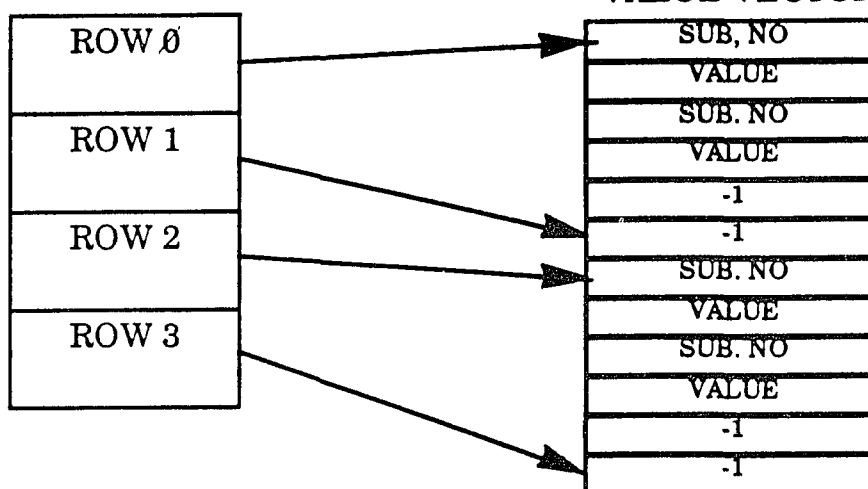
and the operation is repeated n times. Each vector is defined by its ORIGIN byte address (32-bits) and STRIDE (16-bits). The ORIGIN byte address points to the location of the first element of the required vector operand from which successive elements are accessed relative to it using the STRIDE offset from subsequent locations. The STRIDE offset introduces the flexibility required in vector processing by helping to select particular elements as needed. In addition, elements in rows, columns and diagonals of multi-dimensional dense arrays are accessed with relative ease. Subscript is introduced principally for sparse vector processing which occurs largely in linear programming. It represents either the row or column number of a non-zero element of a sparse vector which has been stored using ILLIFE vector storage techniques. **Figure 4.8** illustrates these techniques. The vector mode and vector functions are fully described in the next section. A VPU is initiated when the **Vector Unit Number In Register (VUNIR)** is loaded by the CCU. The VPU may be selected in one of two ways according to the setting of the most significant bit in the VUNIR. **Figure 4.9** show the required bit interpretation.

N	16	Registers for defining the vector operation to be undertaken
Function	8	
V ₀ Origin (lsb)		
V ₀ Origin (msb)	32	
V ₀ Stride	16	
V ₀ Bound	16	
V ₀ Mode	8	
V ₁ Origin(lsb)		
V ₁ Origin (msb)	32	
V ₁ Stride	16	
V ₁ Subscript	16	
V ₁ Mode	8	
V ₂ Origin (lsb)		
V ₂ Origin (msb)	32	
V ₂ Stride	16	
V ₂ Subscript	16	
V ₂ Mode	8	Registers for defining the vector execution unit which will carry out the defined vector operation
S (lsb)		
S (msb)	32	
S Mode	8	
Vector Unit No. IN		
Vector Unit No. OUT		
Vector Unit Free	16	
Vector Unit Interrupt Enable	16	
Vector Unit Interrupt	16	

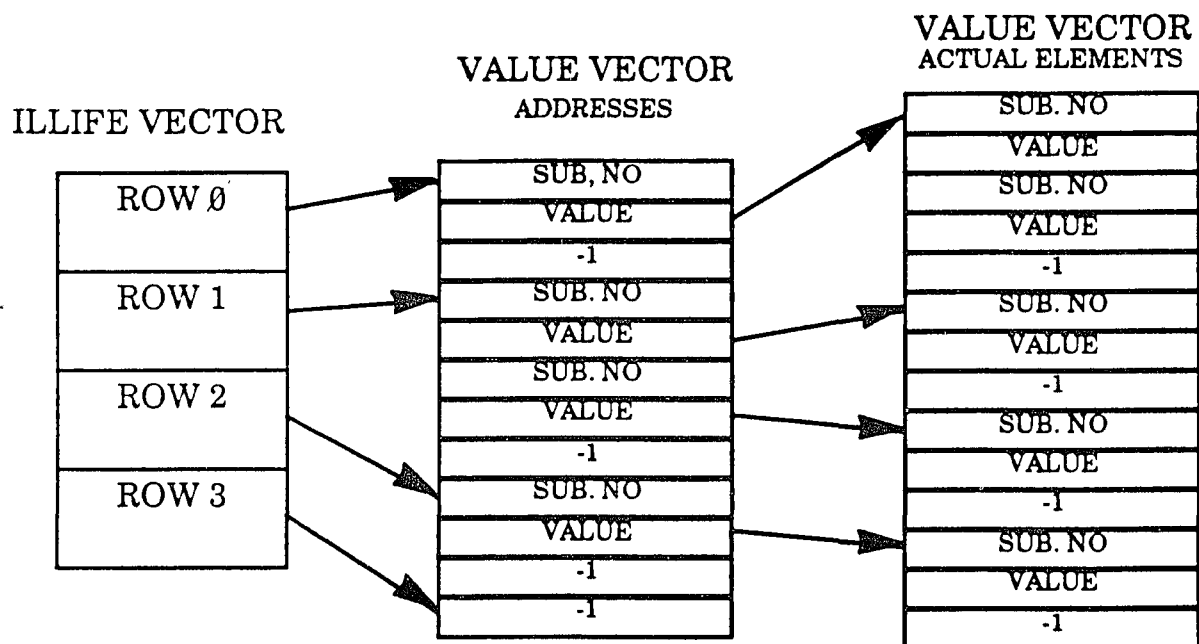
Figure 4.7 Interface Registers Layout

ILLIFE VECTOR

VALUE VECTOR

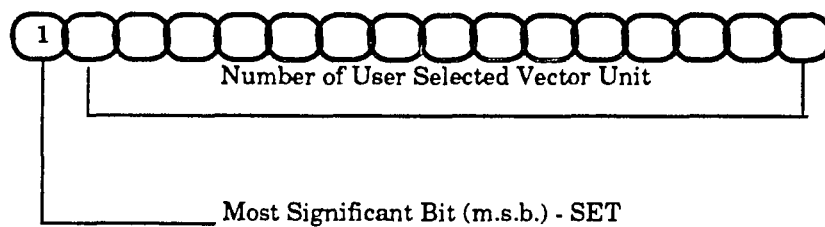


(a) ORDINARY SPARSE VECTOR STORAGE

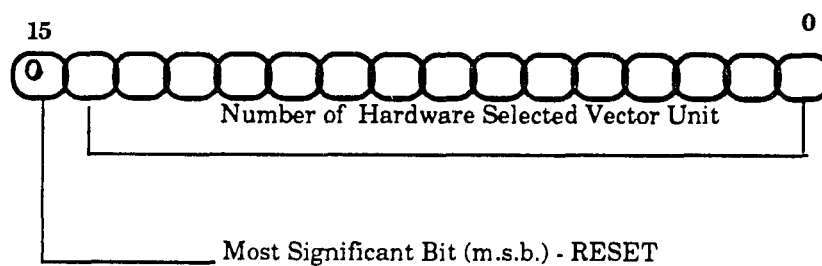


(b) SUPER SPARSE VECTOR STORAGE

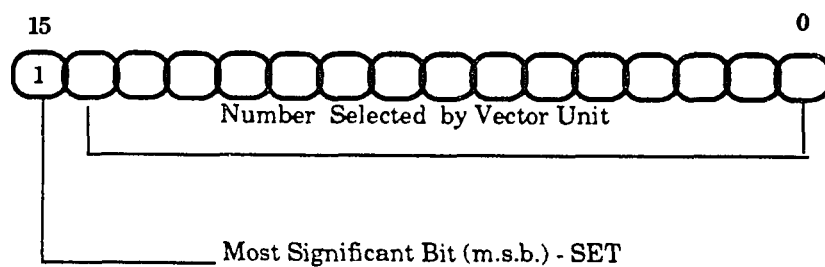
Figure 4.8 ILLIFE Vector Storage Technique



(a) Vector Unit Number In Register (VUNIR) - SET



(b) Vector Unit Number In Register (VUNIR) - RESET



(c) Vector Unit Number Out Register (VUNOR) - SET

Figure 4.9 Free Vector Unit Selection

Vector Unit Free Register (VUF)

BUSY / FREE status bits for each VPU

FREE / BUSY Bit = 0 Implies activated VPU is BUSY

= 1 Implies the Specified VPU is FREE

Figure 4.10 BUSY /FREE bit Setting for VPUs

The **Vector Unit Number Free Register (VUNFR)** contains one bit for each VPU indicating its **BUSY/FREE** status. When a VPU is activated its free bit is set to 0; when it has completed its operations the free bit is reset to 1. **Figure 4.10** shows an illustration of this selection.

Each VPU and the CCU are associated with a status register which contains the information described in **Figure 4.11**. The information indicates the status of a VPU at the point of an interrupt requiring servicing by the CCU.

The **Vector Unit Interrupt Enable Register (VUIER)** is addressed and read by the CCU. The CCU interrogates all the bits in order to ascertain which VPUs require servicing. Once a particular bit in this register has been enabled, its corresponding VPU is then selected for servicing by the CCU and subsequently disabling the enable bit in the VUIER register. See **Figure 4.12**. The destination (V_0) Origin may also be read by the CCU. Each VPU maintains its own B_0 Origin register pointing to the element currently being updated, and the value obtained by the CCU is selected from among these according to the current setting of the VUNIR register. The use of this facility is of utmost importance in sparse vector operations, for example, to indicate the position of the last element created in a sparse result vector.

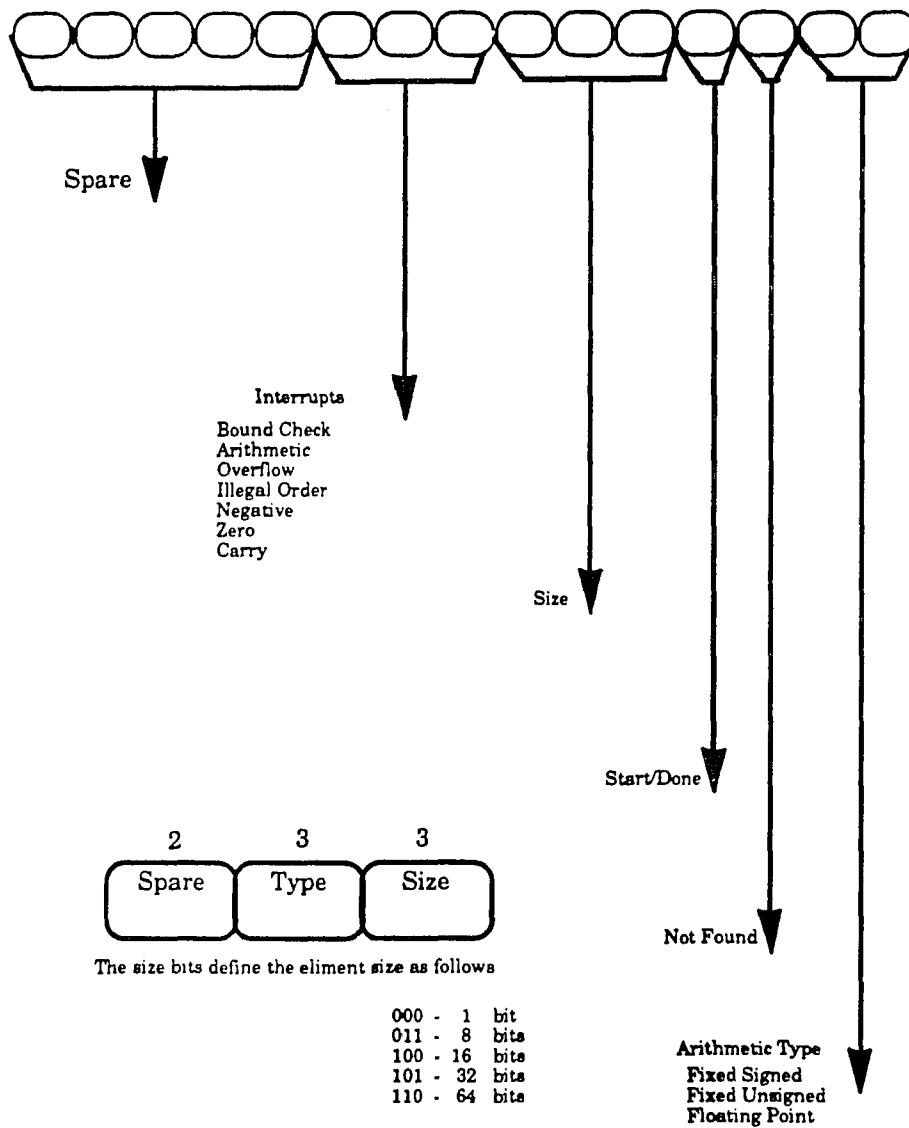


Figure 4.11 Vector Unit Status Register

Interpretation

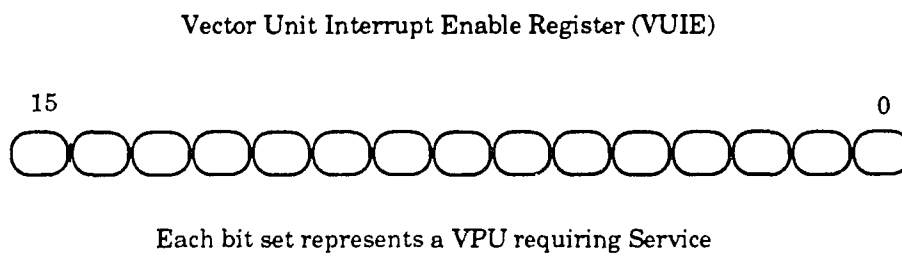


Figure 4.12 Vector Unit No. Requiring Service

4.3.3 Vector Types and Structures

Each vector operand specification is associated with vector Mode Bits which is to be interpreted as indicated in **Figure 4.11**. The TYPE defines the mode of vector accessing as shown in **Figure 4.13**.

When the indirect bit is set the operands are accessed according to the mechanisms defined by the two other bits which are treated as address, requiring a further access for the operand itself. The detailed interpretation of the mode bits is described in **Appendix 1**.

4.3.4 Vector Instructions

The vector instructions have been carefully chosen to reflect the needs of the applications considered in **Chapter 3**. Most of these instructions are by no means conventional, and how to use them effectively in programs becomes a burden. This burden has been taken from the user by the provision of simple but effective mechanisms in the language features discussed in **Chapter 5**, which aid the vectorization of algorithms to be processed on the **NetVecX** architecture. The 8-bit vector function field is split into three subfields as shown in **Figure 4.14**. where GROUP identifies one of the major classes of function as indicated in the figure. Within these groups, the functions are allocated as shown in **Figure 4.15**.

Examples of the sub-functions within each group are used in **Table 3.1**. Normally any vector may be dense or sparse even though some functions are specifically intended for use with one or the other type. Most functions are described in terms of dense vector operations which normally terminate after N cycles, whilst the corresponding sparse vector operations normally terminate when the end marker is reached. After the completion of the vector operation, the V_0 origin field, which is accessible to the CCU, address the byte immediately beyond the location of the last byte written. In any operation in which the number of cycles is not specified absolutely by the N field, for example, in sparse operation, an interrupt is if an attempt is made to exceed the number of cycles specified by the V_0 bound field.

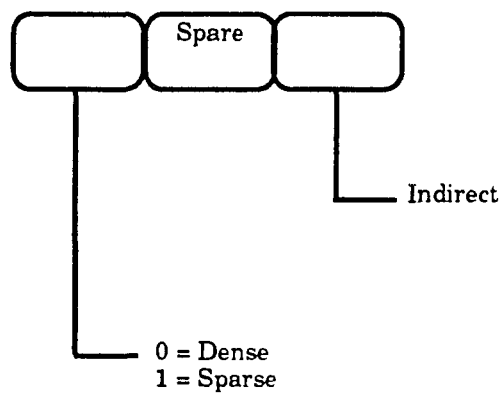
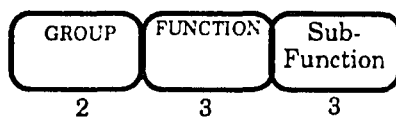


Figure 4.13 Vector Types



Where GROUP identifies one of the major classes of functions as follows:-

- 00 Data Movement
- 01 Data Identification
- 10 Arithmetic
- 11 Accumulative Arithmetic

Figure 4.14 Vector Instructions

00	01	10	11
000 MOVE	VALUE SEARCH	ADD	ADD
001 FILL	INDEX SEARCH	SUBTRACT	
010 COMPRESS	RANK	MULTIPLY	MULTIPLY
011	COMPARE	DIVIDE	
100	COUNT	LOGICAL	LOGICAL
101		SHIFT	
110			
111			

Figure 4.15 ALLOCATED SUB-FUNCTIONS WITHIN GROUP

4.4 Performance Analysis

In considering the performance of the **NetVecX** processor in terms of its throughput, the analysis is viewed in the context of the hardware characteristics and the problem being solved. The performance characterization of the **NetVecX** partly depends on the number of vector processing units (VPUs) in the system. This is illustrated by two cases, depending on whether the number n of concurrent vectors is less than or greater than the number of VPUs (N) in the **NetVecX** system. If $n \leq N$, the number of needed VPUs are initiated only once. (This, naturally, depends on the type of operation being performed). Thus, the time for a parallel operation is independent of n and is equal to t_{par} . Therefore, from the problem's point of view, the **NetVecX** acts as though it has an infinite number of VPUs. By applying Hockney's and Jesshope's generic formula [43], the correct limit is obtained if one assumes:-

$$\begin{aligned} n_{\frac{1}{2}} &= \text{infinity} \quad \text{and} \\ r_{\text{inf}} / n_{\frac{1}{2}} &= (t_{\text{par}})^{-1} \quad \text{for } n \leq N \end{aligned} \quad (4.1)$$

$n_{\frac{1}{2}}$ is the half-performance length, the vector length required to achieve half maximum performance;

r_{inf} is the maximum or asymptotic performance, the maximum rate of computation in

millions of floating-point operations per second (MFLOPS).

On the other hand, if $n > N$, released VPUs will have to be re-initiated several times and the best characterization will be generic approximation that represents the average behavior of the **NetVecX** system. From the generic formula the correct limit, in this case, is given by:

$$\begin{aligned} n_{\frac{1}{2}} &= N / 2 \quad \text{and} \\ r_{\text{inf}} &= N / t_{\text{ll}} \quad \text{for } n \geq N \end{aligned} \quad (4.2)$$

With the pipelined multiple arithmetic units of the VPUs in the **NetVecX/HP** configuration, a more complicated situation arises. The actual execution time in this case is given as:

$$t = z [(s + l - 1) + \text{Sealing}(n/N)] \quad (4.3)$$

where N now represents the number of identical arithmetic pipelines, $s * z$ and l are the start-up time and number of segments in each pipe respectively. $\text{Sealing}(n/N)$ represents vector length, L , that must be processed in each VPU. The generic approximation in this case is given by

$$\begin{aligned} n_{1/2} &= N (s + l - 1/2) \quad \text{and} \\ r &= N / z \end{aligned} \quad (4.4)$$

Thus the performance of a system of N pipelines is shown to be the same or better than that of an equivalent N stage pipelined system.

The performance analysis provides another derivation of the overall throughput which can be defined as the number of operations to be performed in a given unit of time required to execute a single operation in a VPU. Considering the following parameters of the system as a whole :-

N = no. of VPUs.

L = Vector Length

T_p = average processing time per VPU

t = average data transfer time (input/output)

Hence the total time of operation for a vector operation is given by:

$$T = (L + 1)d + T_p + (|L/N| - 1) * (T_p - (N - 1) * d) \quad (4.5)$$

Where the maximum or the asymptotic processing time is simplified to :

$$((T_p + d) * L) / N \quad \text{for very large values of } L,$$

Showing an effective processing time of :

$$(T_p + d) / N \quad \text{per operation.}$$

Hence, the **NetVecX** system exhibits a parallelism of order "N" in that it processes the operations at a rate "N" times that of a single VPU.

The number of VPUs needed to achieve a given throughput may not exceed a theoretical maximum value. It places an upper bound on the number of VPUs which can theoretically be

successfully operated in parallel. Two factors are considered :-

- (1) problem dependent factor,
- (2) hardware dependent factor.

In considering the first factor, it is recognized that the time taken to complete a vector operation is clearly problem dependent and may vary for successive operations. In similar circumstances, the set-up time for an operation may also vary depending on the vector orders and does not exceed a fixed maximum time which is related to the VPU set-up time. In order to achieve a high rate of VPU initiation, the CCU must be capable of initiating all available VPUs in a time less than or equal to the time taken to complete a vector operation in only one VPU. With this assertion, the number of VPUs required to operate successfully in parallel is derived as :

$$\text{No. of VPUs} \leq (\text{Average vector operation time}) / (\text{VPU Set-up time}) \quad (4.6)$$

In (4.2), since several VPUs access the common main store, this store maintains its store data rate determined by its cycle time and the degree of memory interleaving. Similarly, the VPUs maintain their data rates which may vary according to the number of vectors involved in a given vector operation, and possibly the type of arithmetic operation also involved. This, therefore, introduces the maximum data rate, the VPU data rate, so that :-

$$\text{No. of VPUs} \leq (\text{Store Data Rate}) / (\text{VPU Data Rate}) \quad (4.7)$$

Conditions (4.6) And (4.7) should ideally match, but since they are largely program and hardware dependent, it is natural to view the first condition in an alternative way which gives the minimum vector length for which a system with a given number of vector units operates efficiently. Thus, given a vector L and each element operated in a time called Element Operation Time (EOP), then for efficient parallel operation :-

$$L \geq (\text{No. of VPUs} * \text{VPU SetUp Time}) / (\text{Element Operation Time}) \quad (4.8)$$

Element Operation Time (EOT) is approximately the time between the production of successive vector results and is therefore determined by the pipelined clocking rates of the VPUs. The unit set-up time mucks up any vector set-up time arising from the arithmetic unit pipelines of the VPUs. Condition(4.8) would however not apply if the units are operated in lock-step fashion.

In order to sustain a high store Data Rate, the common main store has a very high bandwidth and its design therefore presents some technical difficulties. In this situation, the presence of the local and cache stores actually reduces the common store data rate without degrading the overall performance of the system.

At this point, it is instructive to derive relationships

which govern the performance of the whole processor as result of the use of cache stores in each VPU.

Hence, let:

R_0 be the maximum operating rate for each MAU

R_{par} the maximum data rate between each MAU and its cache store (C_s)

m the maximum no. Of pipelines permitted to operate in parallel in each MAU

C_p the clock period.

Therefore,

$$R_0 = m / C_p \quad (4.9)$$

$$R_{v1} = 3 * R_0 \quad (4.10)$$

where the coefficient 3 represents two source vectors and one result vector in general, but when a vector operation is between a vector and a scalar, or where some pipelines are chained together, R_{v1} is less than $3R_0$. Each C_s also has a second data stream (R_{v2}) to its corresponding local store (LS) :-

$$\begin{aligned} R_{v2} &= R_{v1} / K_v + R_{v1} ((1/3 - 1) / 3 K_v) \\ &= R_{v1} (1/3 + 2/3 K_v) \end{aligned} \quad (4.11)$$

Where K_v is the average utilization coefficient for each vector in the cache stores. Equation 4.11 is based on the

theory that a write operation in C_s must be followed by a write operation in LS. $R_{v2} = R_{v1}/K_v$ if vectors are written into LS when they are rolled out of C_s . **Equation 4.11** clearly shows that each vector must sustain a very high data rate, which is several times that of the maximum processing rates of each MAU. Each local store should have a bandwidth as high as the cache stores but is generally expected to sustain a much lower data rate, since the current vectors in the cache store may be used many more times before being rolled out. If the local stores are large enough, the average data rate for the common main store is very slow.

Chapter 5

NetVecX Programming and Specification Language (NPSL)

The design of the new vector architecture has been influenced by the need for a vector language with mechanisms for parallel computation that have a compatible representation between the architecture and the language making the hardware idiosyncracies transparent to the user. Since most of the parallel languages surveyed in **Chapter 2** and [7, 23, 24, 25, 26, 36, 46, 64, 65, 79, 87, 92, 100, 106, 107] do not contain mechanisms which can fully exploit the parallel features of the **NetVecX** architecture, it became essential to develop a special purpose parallel programming language to enhance vector processing on the architecture we are proposing. In the next section we identify the main features of this new language which are particularly relevant to vector processing. In the rest of this chapter we introduce a base language suitable for implementing NSPL and discuss some of its built-in functions and main procedures.

5.1 General Characteristics of Vector Languages

The design of the **NetVecX** architecture necessitated the development of a programming language for the machine. The formulation of the relevant language abstractions for the machine was abreast of the design of the **NetVecX**

architecture. These features resulted from the study carried out in **Chapter 2** for which the guiding principles were influenced by the following factors :-

- (1) for practical and pragmatic reasons we had to depend on an already existing language and use it as a base language for developing our vector language to reduce cost and implementation difficulties;
- (2) the actual implementation is beyond the scope of this work; it is projected to be undertaken as a future extension of the current work;
- (3) access to a run-time diagnostics system for debugging and testing;
- (4) the architectural features of the machine should be transparent to the user and allow its fully exploitation; and
- (5) it should be possible to directly express parallelism and concurrency of a problem, and the extent of parallelism and its control should be problem defined.

The vectorizing compilers and machine dependent languages discussed in **Chapter 2** have several drawbacks, including the need for the user to restructure his programs by taking into account the hardware characteristics of the processors. Hence, they are not suitable for implementation

on the **NetVecX**.

Although the parallel language, ACTUS [79] enables parallelism to be explicitly expressed in the language without recourse to the machine's architectural features, it nevertheless, has several drawbacks which make it unsuitable for implementation on most supercomputers.

On such machines, parallelism can only be expressed in one direction in a multi-dimensional array and therefore, it is more biased towards array processors, such as the ILLIAC IV, than to the rectangularly or orthogonally distributed array processors such as the ICL-DAP and the multi-pipeline/sequential processors in the **NetVecX** category. The components of the dimension to be processed in parallel can only be allocated to the processors one at a time. Thus in operations where neighboring components are employed in updating each component, the non-local accesses or routings cannot be minimized by using an alternative mapping scheme. The parallel "CASE" can only be implemented on some of the supercomputers which permit each processor to execute chunks of code, and not on lock-step array processors and vector processors such as the CRAY-1 and CRAY X-MP series. Fourthly, the automatic buffering facility is inadequate and inefficient for memory management. In algorithms, such as the numerical solutions of the Navier-Stokes equations a

three-dimensional mesh is processed along all planes in sequence, with the processing of each plane performed concurrently using components from neighboring planes. Thus this facility is far from adequate for implementation on our system.

Implementing ACTUS on a machine such as the **NetVecX**, would lack the essential constructs that can be used to express certain forms of concurrency. For instance, the concurrent execution of several vector instructions, and possibly with a sequence of scalar instructions cannot be adequately expressed. More importantly, the language is not provided with any vector instruction set. It was therefore decided that a language must be developed to exploit fully the **NetVecX** architecture.

The section below describes some of the conclusions of the survey in **Chapter 2** which led to the provision of the following features to exploit vector processing on the **NetVecX** architecture. These are :-

- (1) General data types,
- (2) Vector operations on these data types,
- (3) Operations on conformable vectors, and arrays, i.e., vectors or arrays with the same number of elements within each subscript range,

- (4) Explicit requests for access to rows, columns, diagonals, and sub-vectors,
- (5) Adequate expression of vector conditional evaluations in the language,
- (6) Expression of parallelism in the language,
- (7) Suitable control structures for ease of programming,
- (8) Vector calculations on arbitrary length of vectors,
- (9) Provision of interlocking and synchronization mechanisms.

The aim of the study in Chapter 2 has been to identify the frequency of occurrence and use of certain programming features so that their vector language syntax and rules for manipulation could be formed. These findings were largely consistent with the language criteria adopted for the design of ACTUS [79]. The vector oriented features have been put together to form the basis for the development of the vector language, NPSL, a super-set of the parallel programming language Parallel Pascal (PP) [10,81,87,101].

5.1.1 NPSL Extensions to Parallel Pascal

NPSL is an extension to Parallel Pascal(PP). The choice of Parallel Pascal as the base language has been influenced by the fact that a proven Parallel Pascal compiler has been successfully implemented on a number of machines from mainframes to microprocessors. Thus, minimum effort is required in extending the compiler and to generate code for the **NetVecX** and similar architectures. Parallel Pascal, moreover, is a structured programming language and therefore brings the benefits of structured programming by its implementation on the **NetVecX**.

The areas for extension include the introduction of facilities for the specification of vector operands and hence vector operations, for expressing concurrent vector and scalar operations for accessing the rich and powerful **NetVecX** vector instruction set through a host of built-in functions and procedures, and for declaring, reading and writing sparse data structures. There are also vector quantifiers for scaling vector conditional tests into a single Boolean value in order to determine execution paths. Since the **NetVecX** has a single main memory, there are no provisions for declaring parallel data structures.

5.2 Parallel Pascal Overview

The full definition of Parallel Pascal(PP) and its syntactic rules are presented in [10,87]. The features of Parallel Pascal that lend themselves to vectorization are discussed in the following subsections.

The language permits direct access to parallel Library procedures and functions and has no input and output procedures other than the ones in the System I/O library, which perform I/O operations on basic scalar and aggregate types. There is no notion of files, I/O operations are performed on streams and there are seven streams each for input and output. The files and peripheral devices are first attached to the streams and operations can be performed on a stream by selecting it as the currently detected input or output stream.

As noted above, this approach has been adopted for pragmatic reasons since Parallel Pascal is portable [81,101] which makes the compiler relatively easy to modify. The Vector Language (NPSL) becomes a super-set of Parallel Pascal as a result of the addition of the vector oriented features. The syntax of NPSL is similar to that of Parallel Pascal(PP), **Appendix 3**, and therefore its implementation can be done using a copy of the existing Parallel Pascal Compiler. In this mode, the compile and run time diagnostic and tracing facilities of Parallel Pascal(PP) Compiler can also be

extended to check the new language features[81,101].

It is natural at this point to discuss the most important vector oriented features of the language. This comprises new word delimiters, data declaration statements, assignment statements, arithmetic operations, control and conditional statements which are useful for the applications mentioned in Chapter 3

5.2.1 NPSL Main Vector Characteristics

Appendix 3 and [10,81,82,87,101] reveal the full syntax and capabilities of Parallel Pascal, Parallel Pascal and their extension NPSL. The new facilities include multi-dimensional sparse and dense array declarations and how to access them. Others are techniques for manipulating vectors and sub-vectors of multidimensional arrays as well as the specification of the extent of parallelism [82]. These facilities allow algorithms to be reformulated and new ones developed to express the natural parallelism of problems in the language and hence exploit the vector processing capabilities of the new architecture.

5.2.1.1 NPSL Data and Operands Specifications

In order to specify vector statements, the basic array referencing notation has been extended, so that complete vector and sub-vector operands can be specified. The basic arithmetic, logical and conditional operators are permitted

on the vector operands, therefore the syntactic difference between scalar and vector instructions are the types of the operands of the instructions. The vector operands can be specified in any one dimension, and the dimension must be explicitly stated. Thus matrix operands and other higher dimension operands are not allowed. The vector operands of a vector instruction must be type compatible to match the extent of parallelism.

5.2.1.2. New Reserved Word Delimiters

These have been introduced to facilitate vector and sparse array processing. Some are necessary to extend the scalar control features of Parallel Pascal making it more flexible to handle vector control statements. The most significant are those that introduce parallelism into the language and new dimensions to the conditional statements. These are SPARSE, PAR, SIM, INITIALIZE, ANY, AND ALL. Their meanings are made clear in the appropriate declarative and computational statements which follow in the subsequent sections.

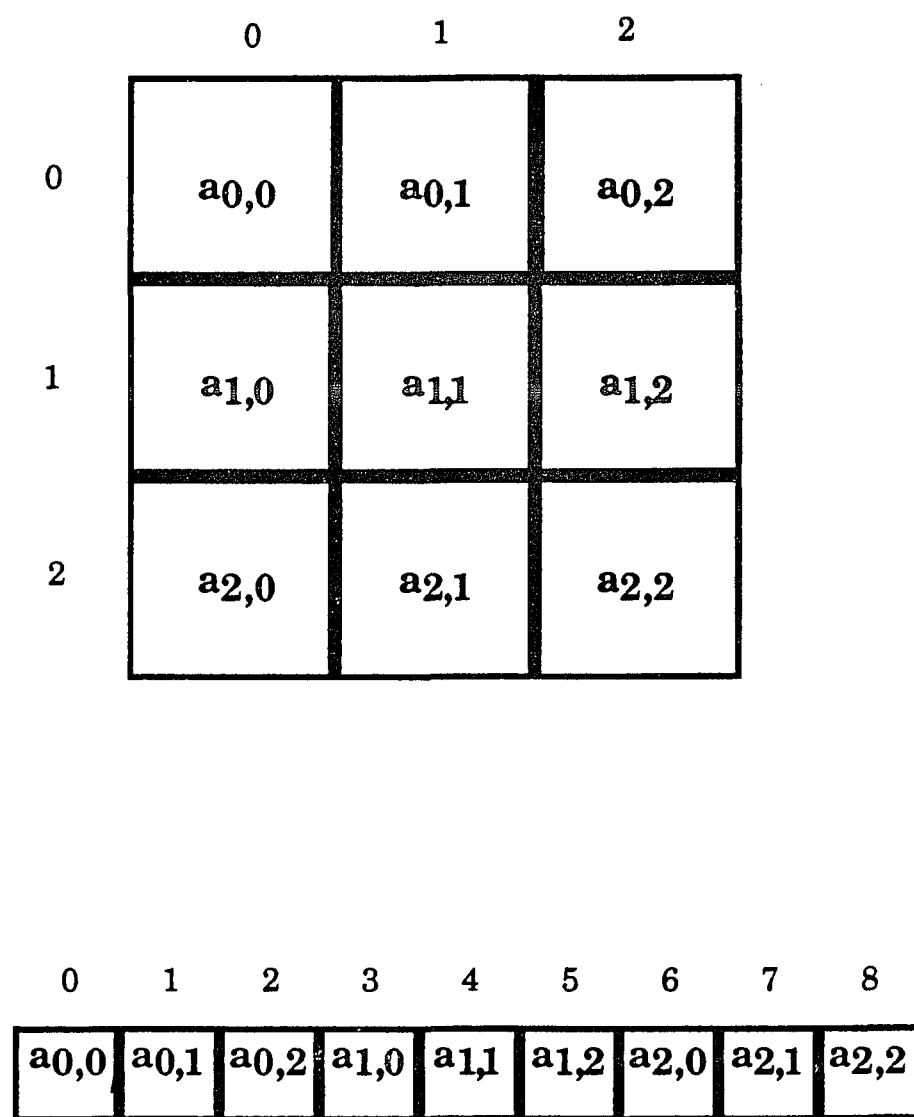


Figure 5.1 A Vector of Mapped 2-Dimensional Array :
Row Storage

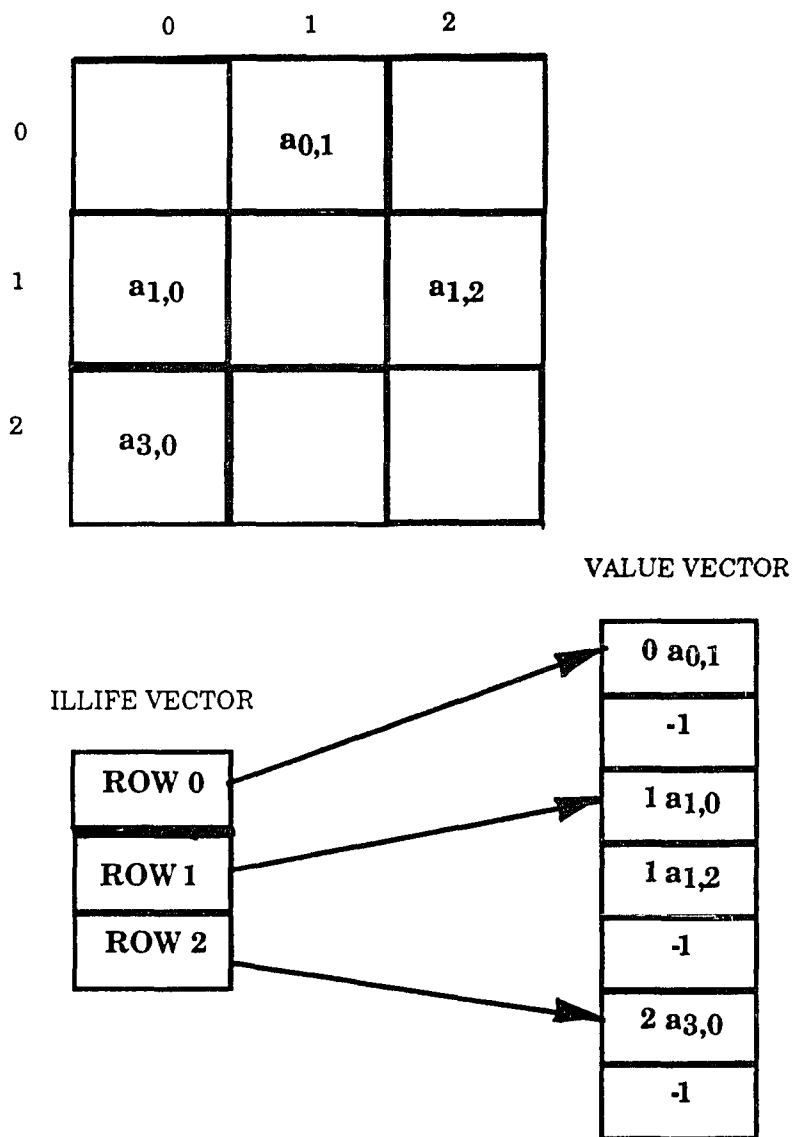


**Figure 5.2 ILLIFE and VALUE Vectors of 2-Dimensional
Sparse Array: Row Storage**

5.2.1.3 Declaratives and Data Structures

The variable and constant declarations optionally include the basic internal formats (INTEGER, REAL, BOOLEAN) and their word length (1, 8, 32, 64 bit hardware operations are provided). Multi-dimensional arrays are declared according to these characteristics as well as the size determined by the compiler and underlying target language. Multi-dimensional arrays are of two kinds, sparse and dense arrays, and hence, the particular type of mapping of the data structure in memory is different for each kind. Dense multi-dimensional arrays are stored as a vector either by row or column. **Figure 5.1** shows a 2-dimensional mapped array for simplicity. It uses a polynomial mapping function following the convention employed in the target language. Sparse multi-dimensional arrays are stored as "value vectors", **Figure 5.2**. Only the non-zero values are stored together with their subscript values either row-number or column-number, depending on the mode of storage. In addition to the value vector, an ILLIFE vector of the array is also created whose elements point to either rows or columns.

5.2.1.4 Arithmetic Operations and Assignment Statements

Arithmetic Operation on entire vectors and vector assignment statements are permitted in the language without necessarily resorting to subscripting. Using conformable vectors, the expression

```
R := P + Q ;
```

adds corresponding elements of P and Q (in standard vector addition) to form the result vector R. Similarly, other standard vector scalar operations are provided. The operators involved take their definitions from the dimension information about the operands. The assignment statement could mean scalar, vector or logical OR depending on the declarations of P, Q, and R. Other forms of representation of vector, scalar and rows or columns of multi-dimensional arrays are presented in **Appendix 3**.

5.2.1.4.1 Accessing Sub-Vectors

NPSL allows operations on sub-vectors and accessing a vector and sub-vectors of multi-dimensional arrays in any given dimension. For example,

```
Z[*] := X[:,2] + Y[10,*] ;
```

adds the second column of X to the tenth row of Y and puts the result in Z. The asterisk is an option which specifies all elements in the conformable vector or all elements in a row or column of a multi-dimensional array. Other operations such as the inner product, maximum element of a vector and vector functions and procedures supported by the hardware are also provided. Accessing an element of a vector or an array by the use of subscripts and other control features for selecting diagonals and other parts of arrays are

incorporated in the languages to facilitate vector processing on the **NetVecX** system.

5.2.1.5 NPSL Control Statements

The control statements embody the imperative statements in Parallel Pascal, the means by which computations are specified. In each MODULE, BLOCK or PROCEDURE in Parallel Pascal are a sequence of imperative statements following its declarative statements [10]. The PARALLEL statement introduces parallelism into the language and is discussed in this section. It basically informs the compiler about the group of statements enclosed in the DO block which may be processed simultaneously in lock-step fashion or synchronously. The compiler then sets up interlocks in the program thereby maintaining program logic by correctly preserving the direct and indirect precedence among the statements. The examples below illustrate the implications of parallel statements in NPSL.

5.2.1.5.1 Abstraction of Parallelism

In processing vectors on the **NetVecX**, execution of identical or different vector instructions are performed in lock-step with scalar instructions. In the process, the scalar unit is forced by the compiler to await the completion of an initiated vector instruction in the user code sequence, thus, a "WAIT" instruction is effectively planted at the end

of each vector instruction. The "PAR" and "SIM" constructs can be used to delay the planting of the "WAIT" instruction or alter the point where it is planted. They can both be nested to any compiler-defined depth with all the existing conventional constructs.

The "SIM" construct is similar to the "FOR" construct where a "WAIT" instruction is planted at the end of each vector instruction in the construct, hence each instruction is completed before another one is executed. The "PAR" construct on the other hand, suppresses the planting of the WAIT instruction at the end of the last statement in the construct until the termination of the loop. Thus if the last statement of the construct is vector instruction, the scalar unit can increment the loop index and enter the loop again without WAITing for the termination of the last vector instruction. However, if the last statement is a scalar instruction, the "PAR" construct becomes a "FOR" construct, since there is only one scalar and it must finish the execution of the scalar before entering the loop again. Therefore, if there is only one vector instruction in the "PAR" construct, several strands of the instruction can be concurrently executed provided vector units are available, and synchronization of the execution is performed when the loop terminates, at which time the execution of all the instructions must have been initiated. Some examples

follow:-

```
(i) PAR (P < J)
      DO
          BEGIN
              S1: S[P] := INPROD(A[*], B[P,*]);
              S2: D[* ,P] := C[P] * B[/,/];
          END;
      S3: B[ , ] := X[*] + A[*];
```

This sets up an interlock on the PAR DO block so that (J) strands of statement S1 and S2 are processed simultaneously to completion before S3 may be started; i.e. first,

```
S[P] := INPROD(A[*],B[P,*]);
S[P+1] := INPROD(A[*],B[P+1,*]); first completed
.
.
.
S[P+k] := INPROD(A[*],B[P+k,*]);
```

then

```
D[* ,P] := C[P] * B[/,/];
D[* ,P+1] := C[P+1] * B[/,/]; completed next
.
.
.
```

```
D[*,P+k] := C[P+K] * B[/,/];
```

before S3 starts

The "SIM DO" construction is used for specifying that several vector instructions, and possibly, sequences of scalar instructions are to be executed. It suppresses the normal planting of the "WAIT" instruction at the end of each statement until the end of the construct, although the scalar instructions are sequentially executed. The following NPSL codes demonstrate the use of this construct. In (ii) below, the "SIM DO" construct forces the simultaneous execution of the three statements in it, and the "WAIT" instruction which ought to be planted at the end of it are suppressed. The **NetVecX** system maintains a stack mechanism in the interface unit for knowing the vector instructions in a scope or construct that are still being operated upon.

(ii) SIM DO

```
BEGIN
```

```
    S1 : B[2,*]:= A[*] + B[1,*];
```

```
    S2 : Z[*] := X[*] / Y[*];
```

```
    S3 : Q[*] := P[*] * C[1,*] + S;
```

```
END
```

```
S4 : Q[*] := MAX.ABS. VALUE(Z[*]);
```

Statements S1 - S3 are independent statements which may be processed simultaneously and all must be completed before statement S4

(iii) SIM DO

BEGIN

PAR P = k:j-1, j+1:n DO

BEGIN

S1: B[*,P] := A[*] * B[P,*];

S2: Z[P] := INPROD(Y[*],X[*]);

END

S3: S := MIN.INDEX(Q[*]);

S4: TEMP := SCL * N;

END

S5: SS := MAX.ABS VALUE(B[*]);

This example shows that (k+n) strands of statements S1 and S2 will be executed in parallel with statements S3 and S4. The PAR DO block is considered as a single statement in the nested block.

The above controls are designed to allow the software to make use of the parallel structures of the architecture.

It should be noted that two main types of PARALLEL statements have been defined : Indexed interlocked and Non-

Indexed interlocked. The former implies seriality on the precedence of the statements within the block but does not lead to standard loops and is not similar to the FORTRAN DO or the ALGOL FOR statements. The bonds on the indices inform the compiler about the number of simultaneous activities to be initiated per statement; all interlocked to preserve the precedence of the statement.

The latter type implies sequential control of the statements within the block but does not impose an interlock on it. The user observes that everything within the scope is executed simultaneously.

5.2.1.6 Conditional Branching

Conditional branching enables the user to decide between two or more execution paths as determined by a test expression in the IF construct. Thus:-

```

IF   $\left[ \begin{array}{c} \text{ANY} \\ \text{ALL} \end{array} \right]$  C1 THEN S1
                                     ELSE S2

```

tests all elements of vectors in C1, a conditional expression.

Vectors involved in a test must have the same lengths

and types. A test involving a vector and a scalar is evaluated for each element of the vector and if found to be true the appropriate statements of the construct are obeyed. The vector quantifiers ANY and ALL may be used in a test by applying them across a sub-vector or the whole vector, in the sense that if at least one element obeys the test or if all the elements obey the test respectively, the appropriate path is taken. The logical operators, AND and OR may also be used in a test alongside the quantifiers. For instance,

```
IF ANY (A[*] > B[*])  
    THEN S1  
    ELSE S2
```

tests each $A[I] > B[I]$ and performs S1 if any is successful otherwise S2 is executed.

These test quantifiers are used to reduce a vector test to a single value, and are therefore governed by the scalar test. They are employed in a program sequence for determining the execution path to be taken as explained above. Since a vector test returns a single value, it can be regarded as an accumulative vector instruction, which the **NetVecX** is capable of handling effectively.

5.3 Built-In Vector functions and Procedures

NPSL allows user-access to the **NetVecX** instructions through built-in functions and procedures. The traditional basic arithmetic and logical operators are the only ones allowed in the language in addition to the traditional mathematical functions, such as ABS, COS, SIN and SORT etc. Thus, programming in the non-conventional language is relatively simple except for the vector operands and the constructs for expressing concurrency.

A built-in function returns a scalar result while a procedure does not return any result, but can be used to update vector and scalar operand parameters. The operations performed by some of the functions and procedures depend on certain conditional tests, therefore there are some pre-declared conditional test specifiers or comparators which are reserved word delimiters. These comparators are "EQ", "NE", "GT", "LT", and "LE" and are only permitted as parameters of built-in functions and procedures. A list of the built-in functions and procedures is presented in **Appendix 2**.

CHAPTER 6

NetVecX Modeling And Performance Evaluation

This chapter discusses the modeling aspects of **NetVecX** and introduces an experimental facility for its performance evaluation. This facility has been implemented through the construction of a deterministic simulation model of the **NetVecX** architecture. The technique is necessary because it is particularly important to understand how the system design parameters and job load variation interact with system performance. The simulation introduced in this chapter differs significantly from most other simulations in that it uses a detailed, deterministic system model rather than the less accurate probabilistic approach. Variation of the system parameters is made possible by incorporating these parameters as variables in the implemented simulator.

The chapter is organized as follows. **Section 6.1** describes the basic concepts of the modelling philosophy, while **section 6.2** discusses the technique for the generation of a control stream needed to exercise the simulator. **Section 6.3** gives a description of the model used while **Section 6.4** discusses the actual control streams used in the simulation experiment. The implementation of the simulator is discussed in **Section 6.5**, while **Section 6.6** discusses the results of

some experiments conducted using the simulator. **Section 6.7** discusses the limitation of the model followed by a discussion of the parallelism abstraction in two programs that are used in the experiments in **Section 6.8**.

6.1 NetVecX Modeling

As stated above, the model of the **NetVecX** computer system is oriented towards simulation of the **NetVecX** system, with a view towards evaluating the performance of various system configurations. This approach entails three most important aspects:

- (1) the concept of a control stream,
- (2) the best technique for selecting the relevant combination of resources to construct the model,
- (3) that the simulation is totally deterministic.

These aspects are discussed in the following subsections.

6.1.1 Control Stream Simulation

Since the model is based on construction of a simulator to model those aspects of the system needed to evaluate its performance, the simulation should provide timing and resource usage statistics for typical system operation. As the system need not be actually constructed for this information to be gathered, recognition of this fact led to a significant reduction in model complexity by the introduction of the concept of a control stream.

In the real system, both the instruction and its data traverse their independent flow paths simultaneously. But the model of the system consists of resources which correspond to the resources of an actual system. In the model, a unit of traffic is generated corresponding to the starting of an instruction along its flow path in the actual system. The two taken together form the control flow of that instruction and are reflected in the flow path of the corresponding unit of traffic in the model. Thus the instruction and data streams in the real system are replaced by a control stream in the model.

The traffic for the simulation model is derived from a program execution trace and hand compilation as discussed in **Section 6.2** below. During the simulation, however, no attention is paid to the actual data used or produced by the program. The data and the results of the executed programs were used to check the validity of the parallel algorithms and program correctness. Thus the model is concerned only with the data flow path and not with the actual data itself. Since only timing statistics are important, the processing of a unit of traffic by resource in the model consists solely of occupancy of the resource by the traffic unit for the characteristic period of time for that resource in the real systems.

6.1.2 Simulation Level and Model Resources

The appropriate level at which the model simulates the system is carefully chosen so that it is not so low as to include system detail that is irrelevant to the performance evaluation nor is it so high as to obscure any factor interactions that possibly play some part in determining system performance. Thus, the most appropriate level of the simulation is the functional level, using the system clock period as the chosen basic time quantum of the simulator.

The model associates a logical resource with combination of various steps in the execution sequence of an instruction. This separation of the association of the execution sequence and assignment of associated resources is made only as fine as necessary to describe the degree of concurrency possible in the system. Thus two distinct consecutive steps in the execution sequence which can never be executed simultaneously for two different instructions are combined and executed in a single resource in the model. The processing time of each resource is fixed by the combination of execution steps that it represents. Hence, each of these resources can process one unit of traffic at a time. A consequence of this technique is that the model has no more resources than required to reflect system timing and dependency accurately and also significantly reduces the model complexity.

The simulator being entirely deterministic is in sharp contrast with probabilistic simulation in which certain aspects are treated as randomly varying quantities. In our

simulator, the flow of a unit of control stream and the time it spends in each resource along its flow path are completely determined by the actual process in a real system.

6.2 Control Stream Generation

In order to drive the simulator of the model, the control stream to be processed by the simulator has to be generated. The simulator does not perform the computation accomplished by the real system. In this case the instructions are sufficiently described to the simulator to determine its dynamic flow. The required information consists principally of:

- (1) the static control flow path of the instruction which comprises the resources needed to process the instruction in the required order;
- (2) the instruction dependency of the instruction stream.

The simulator requires this information in order to set up interlocks for the correct sequencing of the instruction stream. There are two sides to this dependency. These are the data dependency caused by two instructions, say, which reference the same operand location, and the procedural dependency caused by instructions which follow conditional and unconditional branch instructions. All of these constitute the control stream which represents a single execution of a single program. It, therefore, implies that all activities

following branches are known before hand and fed to the simulator. The execution in the simulator is merely delayed until such a time as the branch would have to be completed in a real system.

6.2.1 NetVecX Control Stream Generation

The generation of control stream for the **NetVecX** simulator is carried out in two stages. The first stage involves the design of a preprocessor which is used to execute the application programs needed to exercise the simulator in a program execution trace. This technique enables the procedural dependency information of a typical workload to be gathered as the program is executed successfully. This stage is vital because it enables the programs coded in NPSL to be executed in order to check the validity of the parallel algorithms used and the effectiveness of the parallel features of NPSL used to exploit the **NetVecX** architecture.

In the second stage each instruction of the NPSL application programs is hand compiled into the target machine instruction, in this case the MC68000 microprocessor based configuration described in **chapter 4**, formed the base architecture of the model. Each instruction in the generated code is then mapped into a control stream instruction, specified by the set of parameters needed to describe it to the simulator. The static control flow path of each instruction is entirely determined by the operands that it

used and the operation that it performs on them. The flow-path information is gathered during the execution of the application program by the preprocessor. The dependency interlocks built into the simulator use this and the procedural dependency information to prevent the improper out of sequence usage of operands.

6.3 Modeling of the Central Control Unit Memory System

The purpose of the simulation is to evaluate a number of systems configurations based on the **NetVecX** architectural specifications. The model is therefore constructed to handle any number of such configurations. The following subsections describe the major functional resources modelled, in which the execution times were given in CPU clock periods, each period depends on the spectrum of performance factors supplied to the simulator.

6.3.1 The Central Control Unit (CCU)

Figure 6.1 shows the resources and control flow paths possible in this unit. Instructions are prefetched into the **S-Buffer** from the **S-Store** until they are extracted and decoded. Having extracted and decoded these instructions in a FIFO order, the **Central Control Unit** extracts all the relevant information for processing a vector instruction. It then loads the master Interface Registers via the **S-Bus**. This information is held until a Vector Processing Unit (**VPU**) becomes available for initiation. In the event of gaining access to a free **VPU**, the **CCU** sets its busy flag and transfers

the contents of the master Interface Registers to the corresponding slave set in the selected VPU. Control is also transferred to the VPU which runs autonomously. Meanwhile, the Central Control Unit is then free to initiate further VPUs or execute scalar instructions by itself. Various resource usage statistics are gathered in the CCU. The Central Arithmetic unit performs the floating-point and fixed-point arithmetic operations depending on the data type and corresponding execution time fed to the simulator. The instruction decoding rate is limited by the availability of space in the S-Buffer and the clearing of conditional branch interlocks and the sequence of instructions. For scalar arithmetic operations, decoded arithmetic instructions are sent to the Central Arithmetic Unit. The operand addresses are generated by the Generate Operand Address resource which corresponds to the computation of operand address in the real system. The data and the procedural dependencies are handled by this unit for the correct sequencing of the control stream instruction. These were used in calculating the normalized performance figures in the discussion of the results.

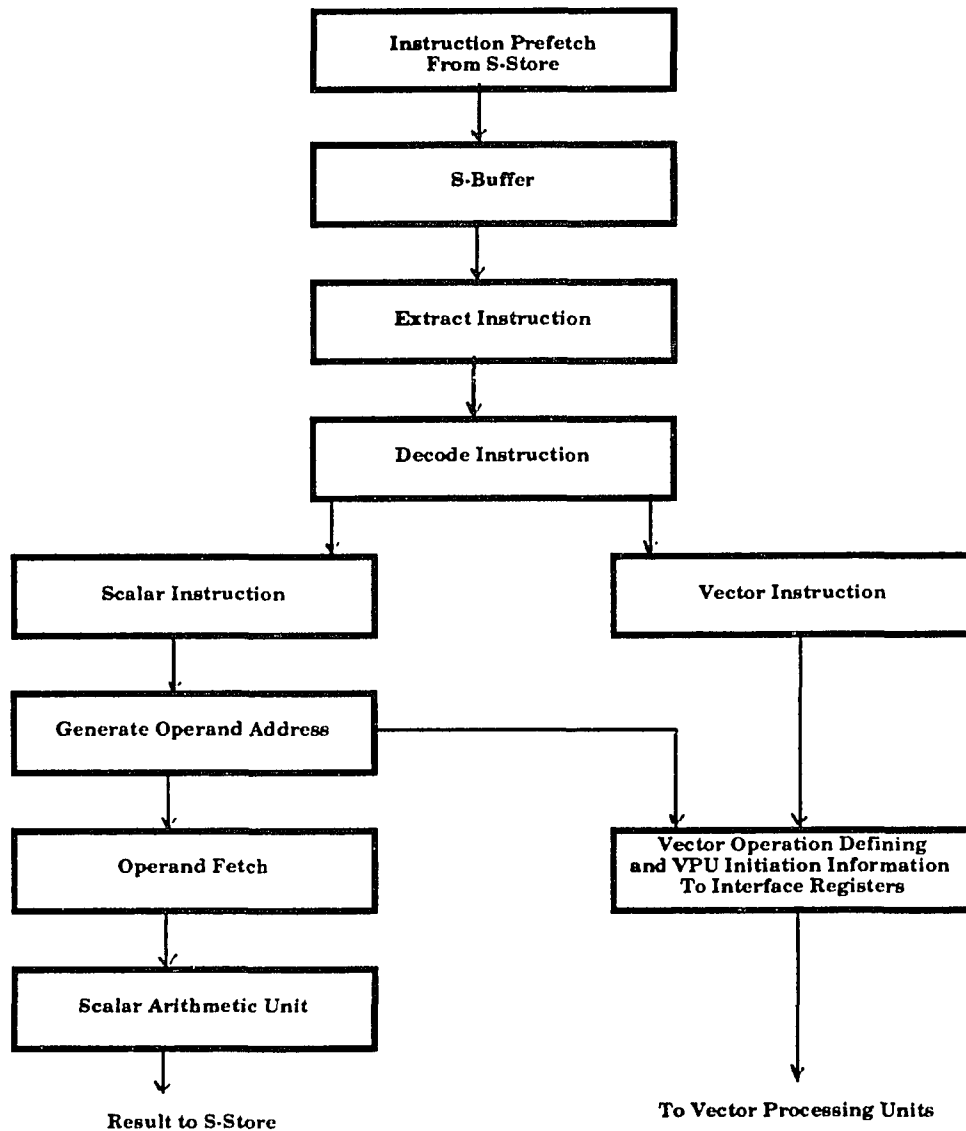


Figure 6.1 Possible Control Flow Paths In the Central Control Unit

6.3.2 The Memory System

In order to increase the performance of the system, the memory system has been split into two sub-systems **S-Store** and **D-Store** for instructions and data respectively. **Figures 6.2 and 6.3** show these resources and the possible control flow paths in these memory sub-systems. In the **S-Store**, for instance, the bus map selects a reference to it and then transfers it to the **S-Store**. If the reference is a fetch request and the **S-Store** is busy, the reference is put into the FIFO queue buffer to wait its turn. When a reference is ready to be initiated, i.e., when the store is free, the reference is transferred from the FIFO queue to the **S-Store**. It occupies the **S-Store** for a period equal to the memory cycle time. At this point, if it is a store reference, then the execution of the instruction is terminated and other instruction begins. If the instruction is a fetch it returns to the instruction unit section of the CCU or to the Central Arithmetic Unit as the contents of the location addressed by that reference. The **D-Store** operates in a similar fashion except in this case the operands are sent to the Vector Processing Units and the result elements sent to the **D-Store** either directly or via the buffer store.

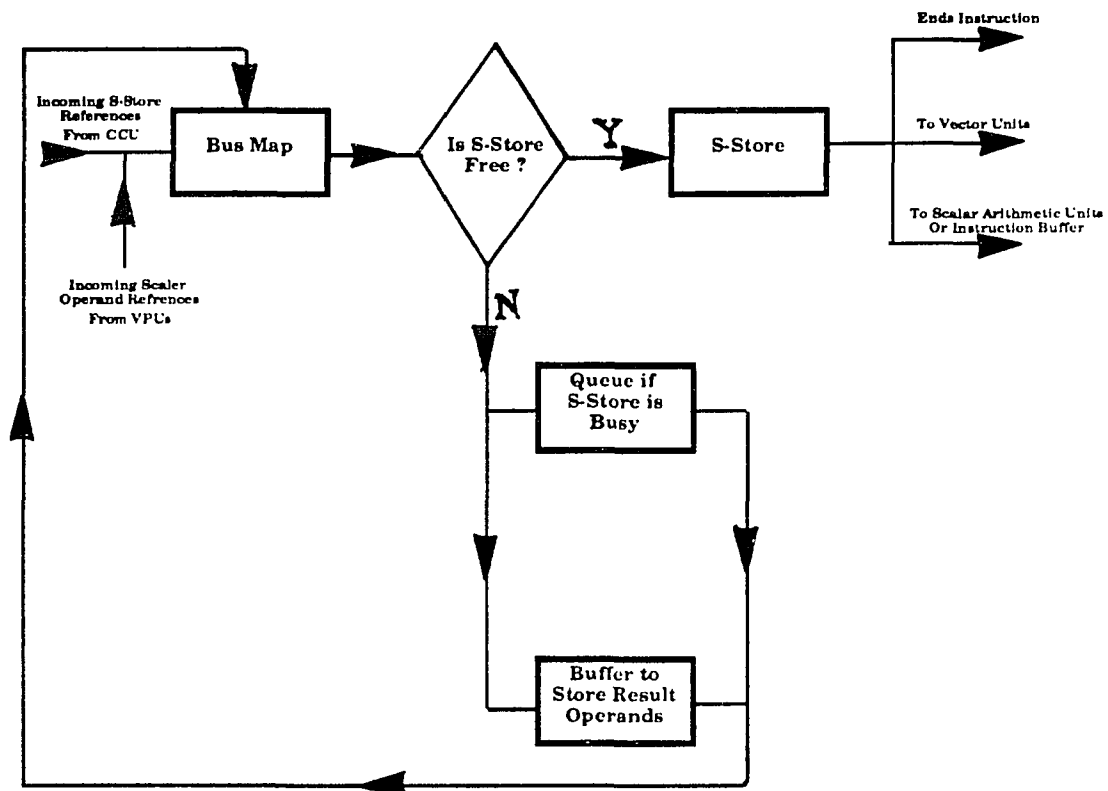


Figure 6.2 Possible Control Flow Paths in The S-store Subsystem

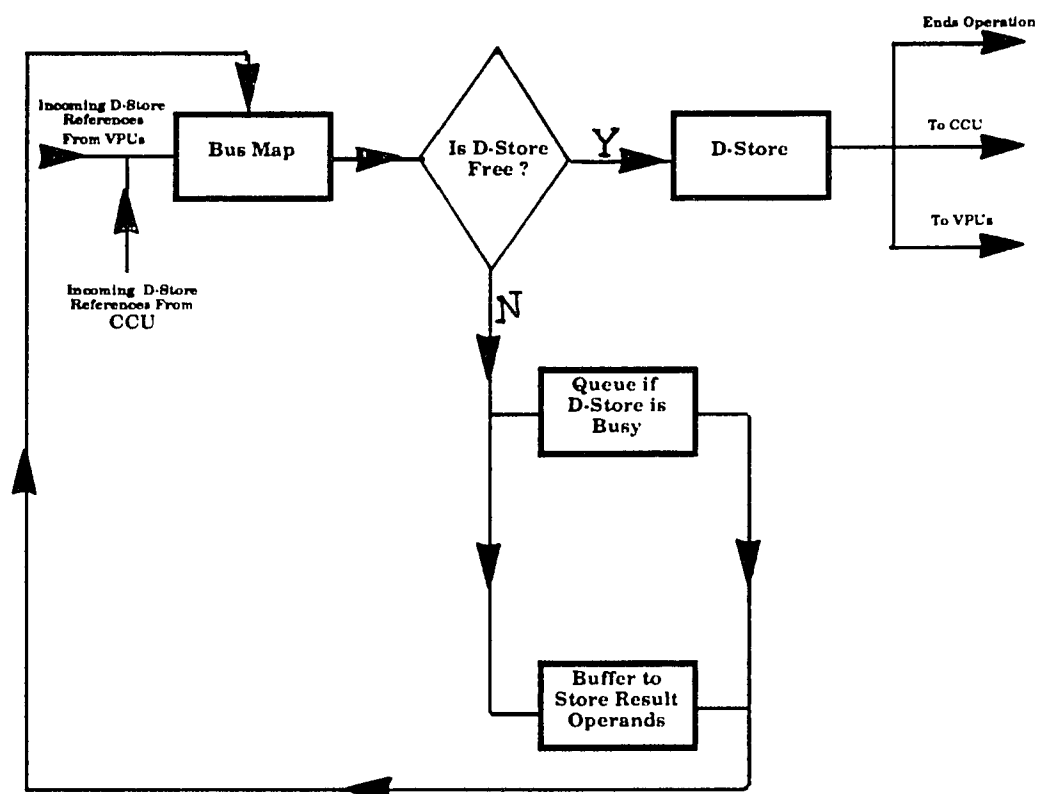


Figure 6.3 Possible Control Flow Paths in The D-store Subsystem

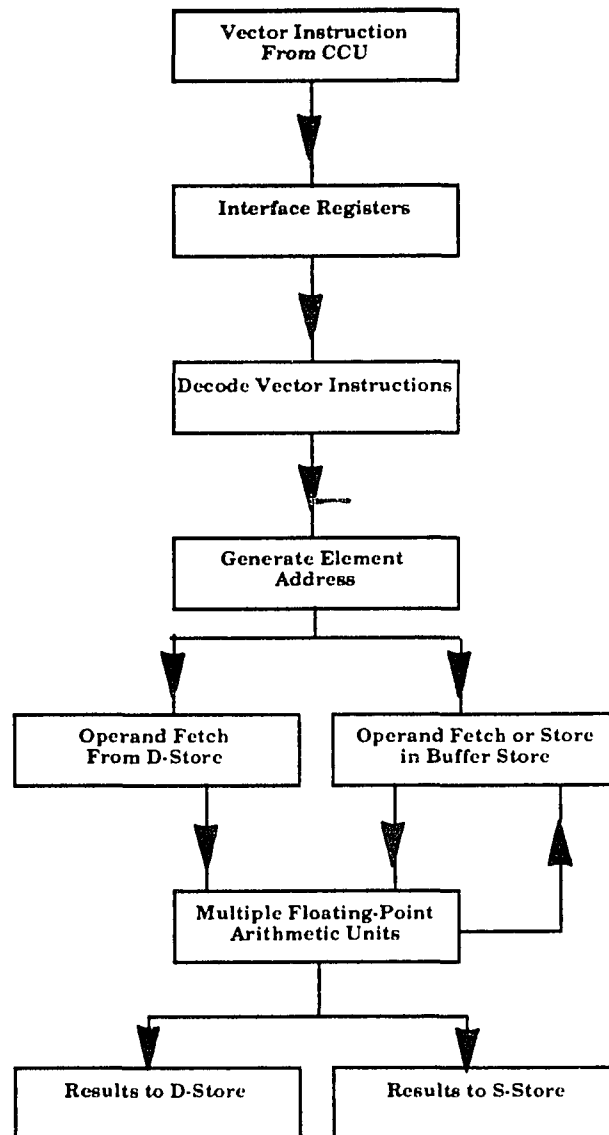


Figure 6.4 Possible Flow Paths in the Vector Processing Units (VPUs)

6.3.3 The Vector Processing Units (VPUs)

Figure 6.4 shows the resources and the control flow paths possible in these units. The VPUs are dedicated solely to vector processing in two system configurations. The first employs no buffering of vector elements while the second configuration uses buffering to ease the pressure on the **D-Store**. After initiating a VPU it proceeds autonomously to execute its vector operations. The information in the slave interface registers is decoded to select the appropriate vector function followed by subsequent address generation for the required vector operand elements. Fetched operands are then transferred to the floating-point unit for execution. The result elements are then sent to the **D-Store** via the **V-Bus** when operating without using the buffers. This process is repeated until all the elements in the vector have been processed. When the buffer flag is set, vector elements are buffered according to the number of source operands involved. In this case result elements are also buffered before being sent to the **D-Store** after all the buffer elements have been processed. The processing rate is limited by the **V-Bus** bandwidth and the memory contention which increases with the number of vector units operating simultaneously.

6.4 Control Streams Used in the Experiments

The control streams are generated as described in **Section 6.2.1**. Two programs written in NPSL are hand compiled and

then transformed by a program written in TURBO PASCAL into control streams used in the experiments. One of the NPSL programs is a matrix multiplication by the inner product method which is highly parallel and easily adapted to the **NetVecX** architecture. The computation is performed for a 16 X 16 floating-point matrices. The sequential version of this algorithm is also used in the experiment for determining the speed-up ratio of this algorithm on the architecture. The second NPSL program is the Gaussian Elimination algorithm of a matrix of size 16 X 16 floating-point numbers. Being a matrix manipulation program it has a large number of loops of varying sizes and instruction mix consisting of floating-point operations, fixed-point operations, and branches. It represents a scaled down version of a typical workload which exercises most of the architectural features of the **NetVecX**.

6.5 Simulator Implementation

The simulator of the model described in **Section 6.3** is implemented in TURBO PASCAL. To facilitate experimentation, a number model features are parameterized to specify alternative sets of parameter values. In this way, a number of various system configurations based on **NetVecX** architecture are realized. Most significantly, a spectrum of computer processor speeds are selected to span the range from low performance machines to high performance vector computers. In addition, the corresponding arithmetic speeds were also

parametrized to reflect the overall performance characterization of these computers as used in the simulator. **Table 6.1** shows a sampling of clock cycle and corresponding arithmetic operation times.

Other parameters which are varied include, the memory cycle time as well as the buffer size used in each of the modelled Vector Processing Units. The system is also configured to run sequentially and the number of Vector Processing Units in the system is also varied to study its effect on performance. Instruction processing in the **Central Control Unit** is pipelined to a limited degree, while the floating-point arithmetic units are not pipelined, since the length of the pipeline depends on the operation being carried out and the technology used. Therefore, arithmetic operations merely occupy the resource for the characteristic period of time required for their execution. A number of these simulator features are used in the experiments described in the next section.

An implementation of the simulator using the discrete-event simulation language "smpl" [69,70,71] is currently being developed. "Smpl" is a functional extension of a general purpose programming language called the host language. This extension takes the form of a set of library functions (the smpl simulation subsystem) which, together with the host

language itself, compose an event-oriented simulation language. A "smpl" simulation model is implemented as a host language program: simulation operations are performed via calls on the functions of the simulation subsystem [68,70,75]. This approach to discrete event simulation is a venerable one. It provides a simulation capability suitable for small-to-medium scale models which can be implemented in most languages on most machines.

The "smpl" simulation subsystem is part of a simulation environment called SMPL. SMPL provides additional tools, including debugging, data collection and plotting, and simulation output analysis functions, together with an interactive interface to simulation model execution, called "mtr". Versions of "smpl" have been implemented in languages including BASIC, C, Fortran, Pascal, and PL/1 on machines ranging from a microprocessor with 32K bytes of memory through an Amdahl 580 mainframe up to a CRAY supercomputer.

Table 6.1 Spectrum of CPUs and Their Floating-Point Arithmetic Operation Time

Computer	CPU CLOCK Period(ns)	ADD/SUBTRACT		MULTIPLY		DIVIDE	
		Time(ns)	Clock Periods	Time(ns)	Clock Periods	Time(ns)	Clock Periods
CDC-6600	120	600	5	1000	9	3000	25
CDC-7600	27.50	110	4	137.5	5	500	20
CYBER-205	20	100	5	100	5	1080	54
STAR-100	40	440	11	520	13	1240	31
TI-ASC	80	400	5	320	4	1280	16
CRAY-1	12.5	751	6	87.5	7	262	21

TABLE 6.2 CHARACTERISTICS OF SOME SUPERCOMPUTERS

	PERFORMANCE				TECHNOLOGY	
	FIRST INSTALLATIO N	MAX (MFLOPS)	CYCLE-TIME NSEC	MAX MAIN MEMORY SIZE (MBYTES)	LOGIC (GATE DELAY)	MAIN MEMORY
FUJITSU VP-200	1983	500	15 VECTOR 7.5 SCALAR	256	ECL (350 PSEC)	64 K MOS STATIC (55 NSEC)
HITACHI S-810/20	1983	630	15	256	ECL (350 PSEC)	10 K MOS STATIC (40 NSEC)
CRAY-1M	1983	250	12.5	32	ECL (0.7 NSEC)	16 K MOS (70 NSEC)
CRAY-X-MP	1983	630	9.5	32	ECL (0.5 - 1 NSEC)	4 K ECL (25 NSEC)
CYBER 205	1982	400	20	64	ECL (700 PSEC)	? MOS (45 NSEC)
CRAY-2	1984	1000	4	1024	ECL ?	? MOS ?

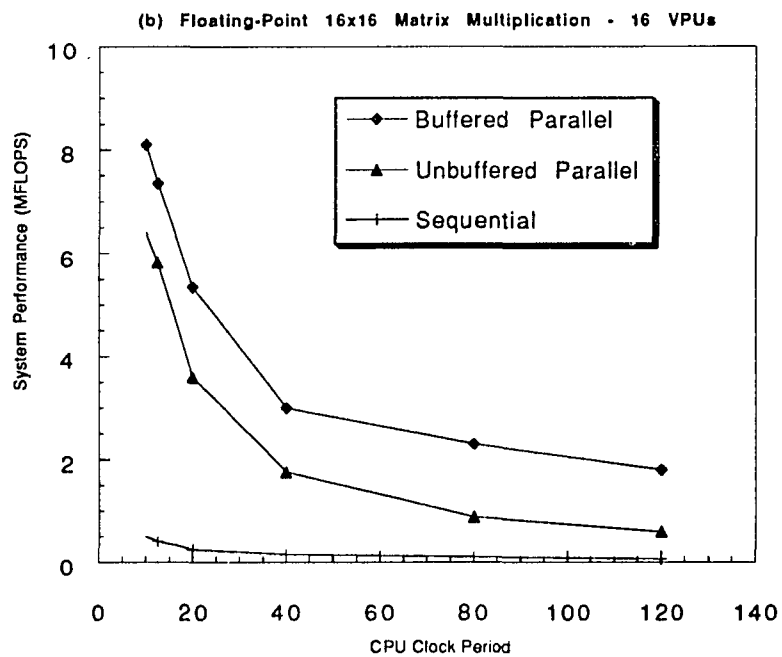
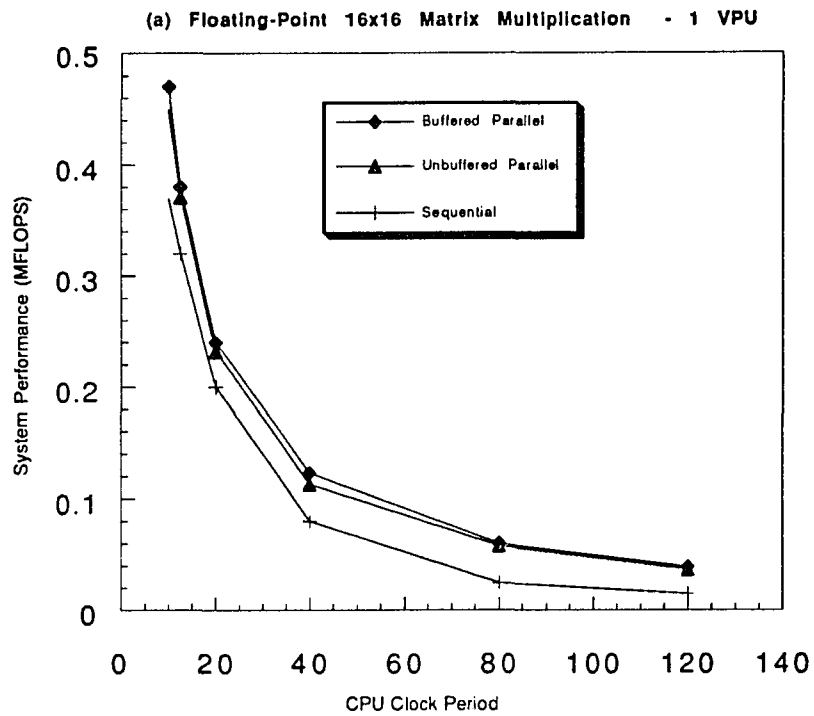


Figure 6.5 System Performance Vs CPU Clock Period

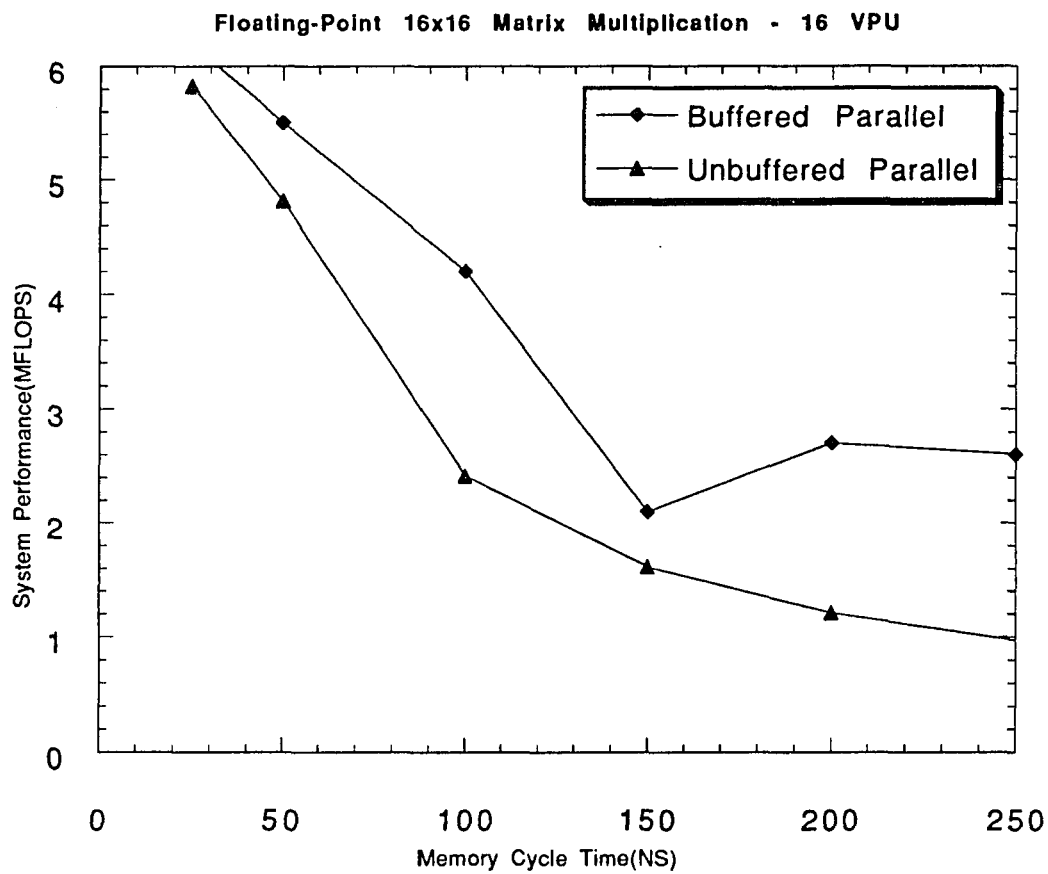


Figure 6.6 System Performance Vs. Memory Cycle Time

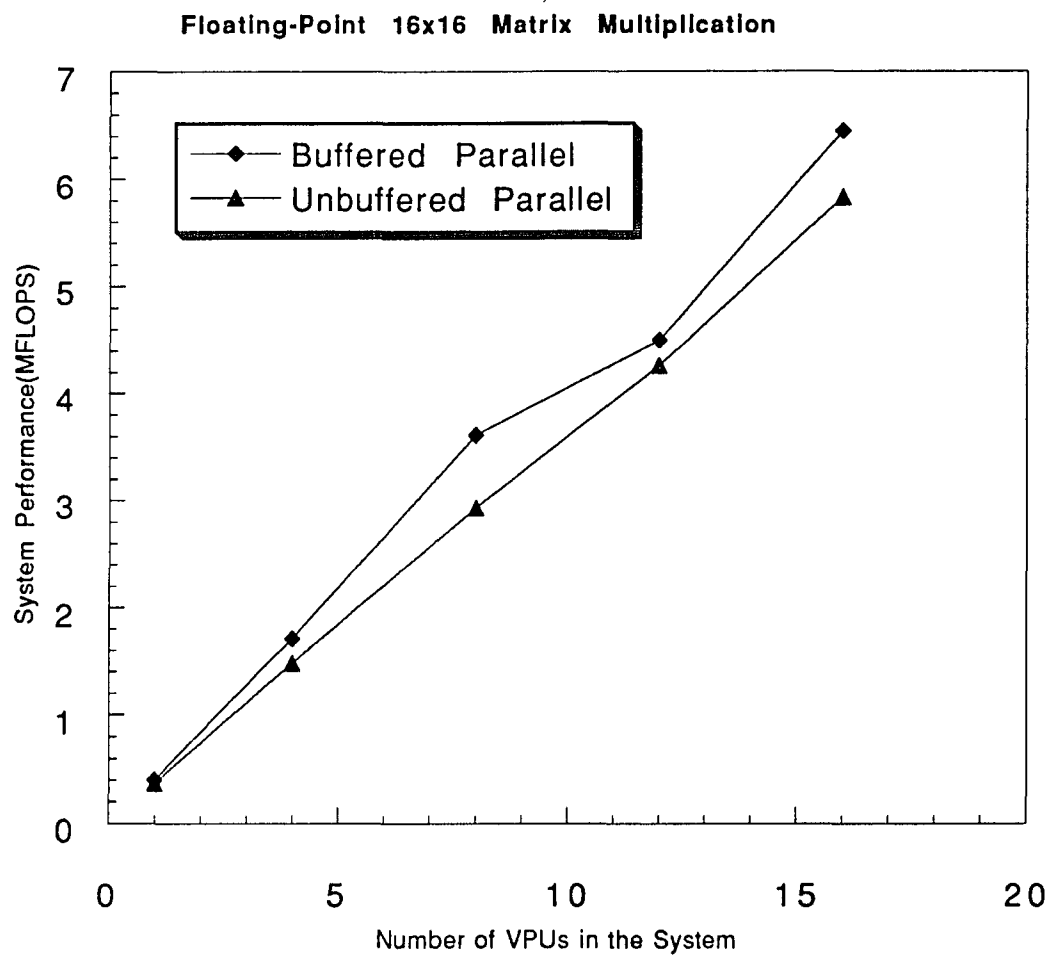


Figure 6.7 System Performance Vs. Number of VPUs

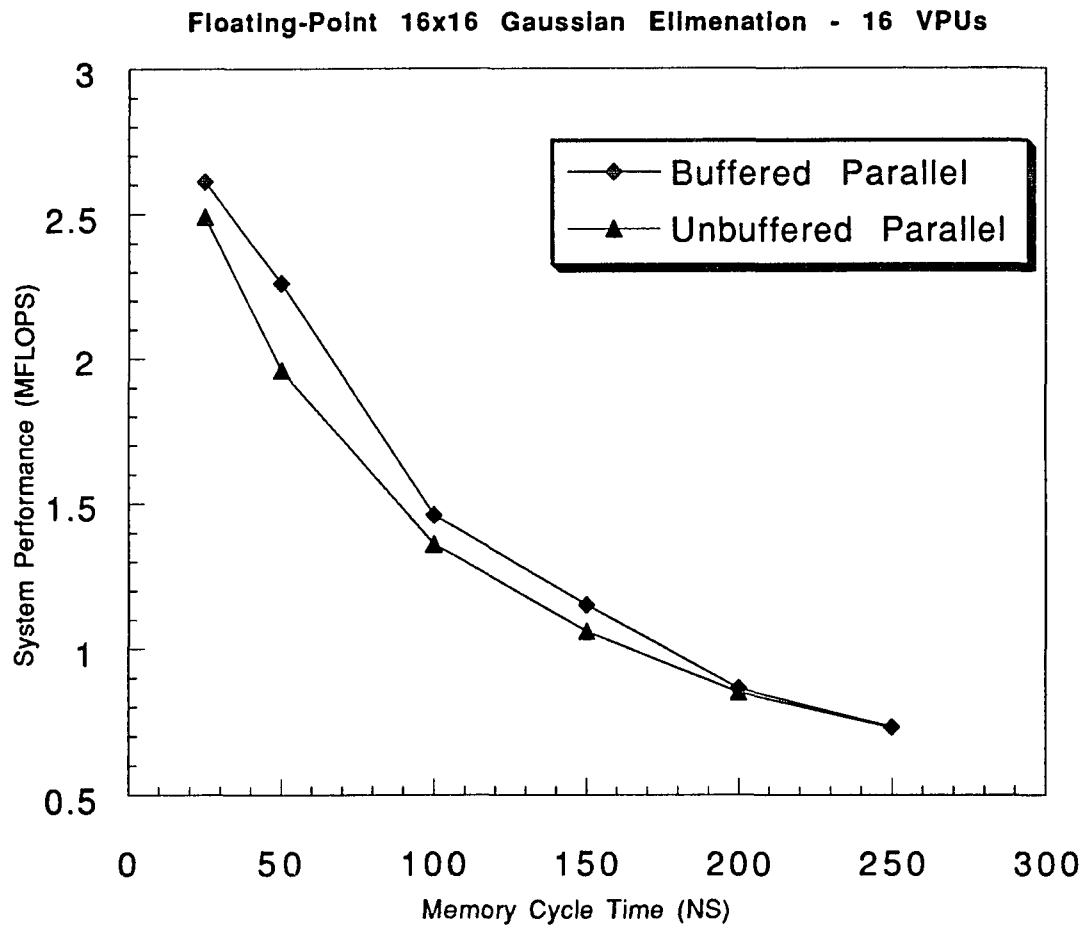


Figure 6.8 System Performance Vs. Memory Cycle Time

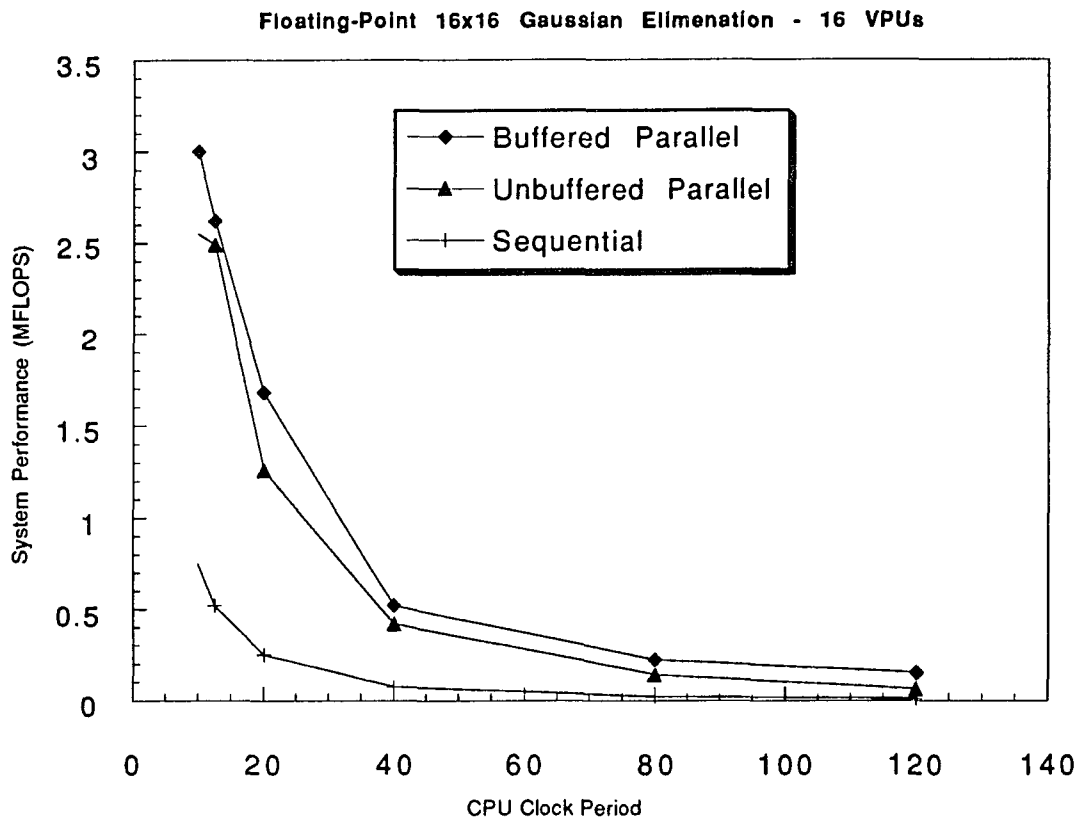


Figure 6.9 System Performance Vs CPU Clock Period

6.6 Discussion of Experiments Performed and Results

This section presents a discussion of the experiments conducted using the simulator. The graphs of **Figures 6.5 through 6.11** and **Tables 6.1 through 6.4** show the performance and behaviour of each experiment on each one of the two programs. The system performance is measured by the average floating-point operations throughput of the system. This is defined as the average number of floating-point operations completed per CPU clock cycle over the run-time of the program. This is computed by dividing the total number of floating-point operations executed in the program over the number of clock cycles that the simulator took to execute the program. Thus the performance represents an estimate of the MFLOPS rate of the system normalized to a CPU time of one second. This measure is more meaningful than the classical "Millions of Instructions Per Second" (MIPS), especially when comparing relative speeds of scalar and vector machines.

The number of floating-point operations required to multiply two n -dimensional square matrices is:

$$n^2 * (2n - 1),$$

since the n^2 elements of the result matrix is formed by running n products. The corresponding figure for the Gaussian Elimination is given as:

$$n^3/3 + n^2 - n/3.$$

These numbers are used in calculating the normalized performance figures in the discussion of the results.

6.6.1 Sequential Model

An experimental system in which only the **Central Control Unit (CCU)** is used to conduct an experiment involved a strictly serial mode of processing. The system uses the same resources as a **VPU** but no concurrency is allowed between different instructions. As illustrated in the graphs of **Figures 6.5 and 6.9**, the respective performance of the sequential **Floating-point Matrix Multiplication (FMM)** and the sequential **Floating-point Gaussian Elimination (FGE)** algorithms are low as expected. This model is used because it is required in conjunction with the parallel versions of the algorithms to measure the amount of parallelism in the **NetVecX** architecture by calculating the speed-up ratios.

6.6.2 Bufferless-System Performance

A number of systems are configured by varying the number of **VPUs** in the model. The first of such systems employs only one Vector Processing Unit to study the effect of the **FMM** on the overall performance. Thus the **CCU** only initiates one **VPU** repeatedly to execute vector instructions after loading the interface register with all the relevant information about the vector instruction. As expected, the performance of the system increased by about 7% over the sequential model and

a speed-up ratio of about 1.7. The graph of **Figure 6.5 (a)** illustrates the effect on the CPU speed on system performance when configured with one VPU. With no buffering of the vector elements in the VPU, no attempt is therefore made to increase the bandwidth and therefore does not significantly change the performance over the sequential system. The expected improvement is partly due to the elimination of the scalar loop mode feature which dominated the sequential algorithm. Hence several scalar instructions have been eliminated, since a single vector instruction is equivalent to a loop of several scalar instructions. **Appendices 1 and 2** shed more light on this assertion.

A number of systems were configured based on the use of the maximum number of VPUs (16) intended for this system. The CCU served as the main communication channel between it and the VPUs via the interface registers. The Bus Map establishes the two-way communication between the VPUs and the scalar store on the one hand and the **Central Control Unit** and the VPUs on the other hand. The CCU decodes the instructions and generates all the relevant information about the vector instruction. This information is then transferred to the Interface Registers which define the vector operation. The CCU subsequently selects an available VPU and initiates it by writing into the vector unit defining register, thus transferring control to the selected VPU. This unit then

proceeds autonomously until it completes its vector operation. Meanwhile, the CCU becomes free to start other available VPUs provided the next instruction is a vector and ready for execution. Memory references generated by the VPUs are sent to the D-Store via the V-Bus so that the CCU can access the S-Store for instructions with minimum of memory access conflicts. When the FGE algorithm is run, there is a concealed dependency of the CCU on the D-Store with respect to operand accessing.

The performance on the FMM and the FGE algorithms are illustrated graphically in Figures 6.5 through 6.9 when executed using all the 16 VPUs. As expected, the performance of the FMM Figure 6.6 is superior to the FGE Figure 6.8, since the FMM is more amenable to vector processing than the FGE. The performance of the FMM over the serial version rises by about 89 % while the parallelism measured in the parallel FMM over the serial version is given by the speed-up ratio of 8.4. This is at least half the expected theoretical value of 16 for an "Idealized Parallel Machine" with 16 ideal processors. This increase in performance is very promising considering the fact that the arithmetic units are not pipelined. A number of factors that contribute to the degradation of the system are discussed below and to shed some light on the speed-up values obtained from the simulation.

The performance of the parallel version of the **FGE** algorithm is also higher than the serial version by about 60% and a corresponding speed-up of 5.6. This level of performance is not so surprising because the **FGE** algorithm is a mixture of scalar and vector instructions giving a more realistic appraisal of the system. **Figure 6.9** shows this trend in relation to its sequential version which suggests that the amount of parallelism contained in the parallel **FGE** algorithm is worthwhile.

Table 6.3 Memory Contention in a Typical System Configuration

VPU _s										
MEMORY CYCLE TIME(ns)	Matrix Multiplication					Gaussian Elimination				
	1	4	8	12	16	1	4	8	12	16
250	0.0	89.36	88.34	98.35	98.61	0.0	86.89	88.10	89.50	98.31
200	0.0	88.00	87.40	98.96	98.44	0.0	81.41	85.96	88.63	97.44
150	0.00	89.10	86.93	96.67	99.00	0.0	78.13	79.93	89.45	47.10
100	0.0	84.99	83.87	97.69	98.00	0.0	84.61	83.87	88.51	49.88
50	0.0	10.01	24.66	29.41	60.62	0.0	75.11	66.24	85.66	87.41
25	0.0	0.02	49.89	47.35	48.61	0.0	66.26	69.89	70.12	70.21

(a) Unbuffered System

VPU _s										
MEMORY CYCLE TIME(ns)	Matrix Multiplication					Gaussian Elimination				
	1	4	8	12	16	1	4	8	12	16
250	0.0	10.75	11.87	10.66	18.43	0.0	68.12	54.41	61.45	68.12
200	0.0	4.71	7.32	7.11	8.13	0.0	38.37	37.31	57.00	66.44
150	0.00	3.81	6.23	6.70	7.51	0.0	35.11	36.41	55.11	64.83
100	0.0	3.41	5.80	6.13	6.88	0.0	34.10	31.11	48.55	53.11
50	0.0	2.81	4.57	5.10	6.11	0.0	40.63	29.21	21.31	44.75
25	0.0	0.01	1.18	0.98	1.21	0.0	38.51	25.21	39.67	41.47

(a) Buffered System

6.6.3 Buffered-System Performance

This system configuration attempts to increase the bandwidth of the arithmetic units in the Vector Processing Units by introducing a moderate amount of "concealed pipelining". A buffer size of 1K 64-bit words, operating at speeds matching that of the VPUs, is introduced into each VPU in order to ease the pressure on the D-Store as the number of VPUs increases with different configured systems. A buffer hit ratio of 100 per cent is assumed. Table 6.3 lists the percentage of memory contention encountered with and without buffers during a typical run of the two algorithms. Using a system configured with 16 VPUs, the relative performance increases of the unbuffered parallel FMM and FGE over their serial versions are 4.4 and 3.8 respectively, as depicted in Figures 6.5 and 6.9 respectively. These low relative increases are explained by the fact that at high CCU speeds both the unbuffered and buffered systems exhibit relatively low memory access conflicts increasing as the number of VPUs increase as shown in Table 6.3. The measured parallelism using buffered systems for the two algorithms are given by the speed-up ratios at 13.5 and 6.3 respectively. This implies that the V-Bus becomes less busy and queues formed by VPUs waiting for operands from the D-Store are drastically reduced. It therefore leads to a relative improvement in system performance by increasing the bandwidth of the V-Bus. It further suggests the introduction of cache and local

stores into each VPU to buffer up operands for pipeline processing. This presupposes that the arithmetic units must be pipelined; and with advanced technology high performances comparable to at least the CRAY-1 and the CYBER-205 may be achievable on the NetVecX architecture.

Factors that affect the speed-up ratio are now discussed. When assessing the effectiveness of a parallel computer to utilize the algorithmic parallelism in a problem, it is appropriate to define a measure of the improved performance. One such measure for specific problems is the speed-up ratio used above and defined as:-

$$\text{Speed-up Ratio} = \frac{\text{Run time on a serial computer}}{\text{Run time on a parallel computer}}$$

The speed-up ratio for a problem requiring "P" processors on an "idealized" parallel computer would be "P"; unfortunately, such an increase is seldom obtained. A number of factors contribute to a degradation of the speed-up ratio may be classified into two main groups: machine-dependent factors and algorithm-dependent factors. The machine-dependent factors that reduce the speed-up ratio include the following:-

- (1) the number of each type of resource available at any instant;

- (2) the time delays in transferring data;
- (3) the memory access conflicts;
- (4) the parallelism within the computer architecture;
- (5) the instruction mix and the time to perform the required operations;
- (6) the degree of instruction parallelism that can be exploited by the processors.

Even with the elimination of machine-dependent factors from studies of speed-up ratios, certain algorithm-dependent factors still degrade the performance of a computer system. For example, theoretical limits to the degree of parallelism that can be exploited are found to exist for some problems [1].

To achieve a performance near the theoretical maximum value, the arithmetic units must be pipelined with a very fast clock. The clock period of a pipeline is influenced by two factors, namely the technology employed and "rate of servicing" of the store requests. The technology employed in a pipeline should present little difficulty since the CRAY-1 built with advanced integrated circuits has a pipeline clock period of 12.5 nsecs, however the CRAY-X-MP has a pipeline clock period of 9 nsecs.

6.6.4 Effect of the Memory Subsystems on Performance

To evaluate the effect of the memory sub-systems on performance, several systems are constructed by varying the memory cycle time. Table 6.3 lists the percentage memory contention in a number of typical system runs using the **FMM** and **FGE** algorithm. Figures 6.6 and 6.8 describe this phenomenon graphically and show how the relative performance of both programs decreases approximately exponentially with increases in cycle time over a fairly wide range of cycle times. This is true not only for the **FMM** but also for the **FGE**. The unbuffered execution of both programs performs very poorly. A number of **NetVecX** architectural features such as extensive buffering and arithmetic unit pipelining are necessitated by the disparity in the **VPUs** and memory bandwidth. The experiments described in this subsection show that the memory cycle time is still one of the dominant factors in determining performance despite the above mentioned architectural features.

6.6.5 Number of VPUs Effect on Performance

It is clear from the previous sections that the **NetVecX** system provides a highly successful new way of employing a large number of **VPUs** on one problem. It is also suggested to investigate the relative performance by varying the number of **VPUs** in several different configurations. This is illustrated graphically in Figure 6.7 for the **FMM** algorithm. It is clear

from the figure that system performance increases approximately linearly with the number of **VPUs** in the system. With a buffered system, there is minimal change in performance with fewer **VPUs** in the system.

This suggests that the external functional behavior of the system does not depend on the number of **VPUs** in the system. The only effect of changing the number of **VPUs** is to change the computational rate of the system. Thus **VPUs** can be added or removed from the system without changing any applications or systems programs. This allows a straight forward trade-off between hardware and performance.

6.6.6 Matrix Size and Vector Start-up Time Aspects

A number of systems are configured to study the effect of matrix size and vector start-up time on performance. The result of this study on the **FMM** algorithm are illustrated graphically in **Figure 6.10** in which a remarkable feature emerges. The performance of the system increases linearly when the matrix size is less than the maximum number of **VPUs** in the system. As the matrix size becomes greater than the number of **VPUs** in the system, the performance remains fairly constant with minute increments. This may be explained by the fact that from the point of view of the problem, the matrix acts as though it has an infinite number of **VPUs** in the system, where the waiting time for the release of a

vector unit relatively short with shorter vector lengths per **VPUs**. On the other hand, released vector units will have to be re-initiated several times incurring the penalty of more vector start-up times which degrades the system performance. The relative wait time for the release of the vector units is problem dependent and varies with the vector length. This, therefore, suggests a number of architectural improvements in the **Central Control Unit**, arithmetic pipelines, and cache stores, to minimize overhead. It is very significant that the performance does not exhibit the "drop pattern" observed on the CRAY-1 when executing matrix multiplication [19]. This result implies that with fully pipelined **VPUs**, the performance of the **NetVecX** will be significantly improved in matrix manipulation problems.

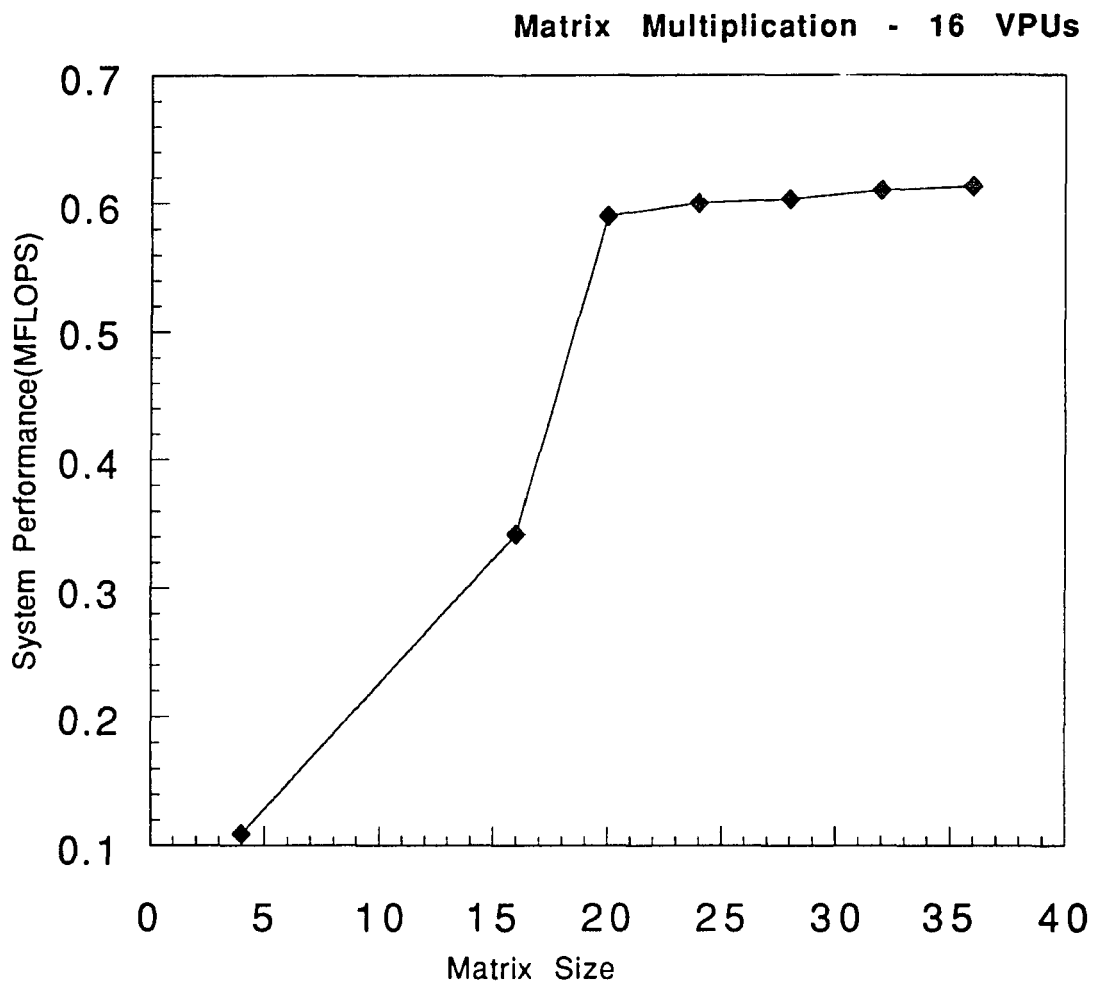


Figure 6.10 System Performance Vs Matrix Size

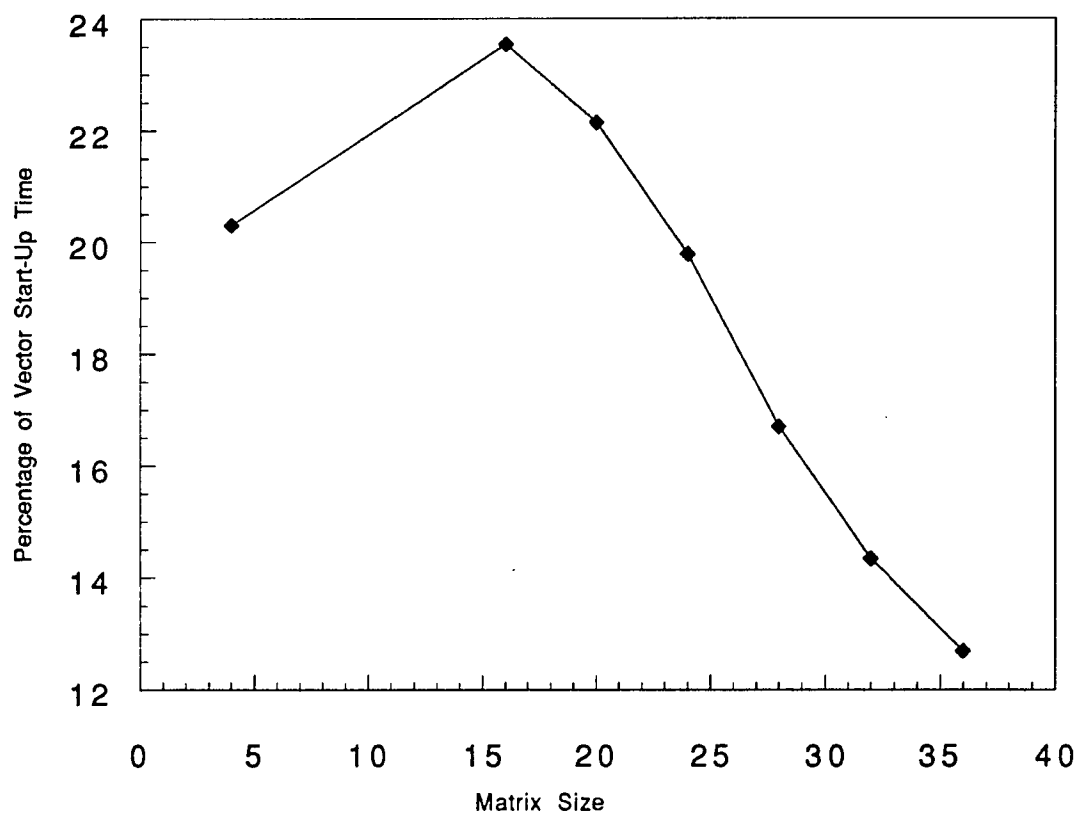


Figure 6.11 Effect of Vector Start-Up Time on Performance

The vector start-up times adversely affect the performance of the configured system. **Figure 6.11** clearly suggests that the performance is severely degraded with shorter matrix sizes. This implies that a new technique should be found to improve performance with respect to shorter vectors. The vector start-up time in this case include the following:

- (1) time for defining a vector operation;
- (2) time for loading the Interface Registers;
- (3) time for selecting a free vector unit;
- (4) time for signalling the selected vector unit.

These time overheads have been incurred because of the basic **NetVecX** architecture. **Figures 6.5 and 6.9** clearly suggests that with low performance **Central Control Unit**, the vector start-up times dominate as a result of the longer instruction execution times. With advanced technology and improved **CCU** speeds this trend can be more than reversed.

Percentage Utilization of Unbuffered System

Resource	Matrix Multiplication	Gaussian Elimination
Central Control Unit	18.20	12.78
S-Store	21.87	10.24
D-Store	60.14	28.44
Bus Map	2.36	23.81
S-Bus	20.36	38.49
V-Bus	59.44	58.11
I-Bus	8.83	45.36
Interface Registers	16.43	11.23
VPU ₀	7.30	56.07
VPU ₁	7.30	34.43
VPU ₂	7.30	33.88
VPU ₃	7.30	33.05
VPU ₄	7.30	31.95
VPU ₅	7.30	30.57
VPU ₆	7.30	28.92
VPU ₇	7.30	26.99
VPU ₈	7.30	24.31
VPU ₉	7.30	22.38
VPU ₁₀	7.30	19.27
VPU ₁₁	7.30	15.39
VPU ₁₂	7.30	12.39
VPU ₁₃	7.30	8.53
VPU ₁₄	7.30	4.41
VPU ₁₅	7.30	3.97

6.6.7 Basic System Profile

Statistics gathered during the simulation experiments show that the resources utilization in the various configured systems are clearly unbalanced in some of the systems. This is especially true in the case of the **FGE** which uses the maximum 16 **VPUs**. The relative percentage system utilization figures of typical system profiles are shown in **Table 6.4**. It confirms the assertion that the memory unit (**D-Store**) is a major bottleneck in the system, since it is a resource shared by all the **VPUs** and the **Central Control Unit**. The high relative utilization of VPU_0 during execution of the **FGE** is caused by the frequent allocation of it to execute single vector instructions in lock step fashion, a decision made to simplify the simulation model. The unbalanced utilization of the remaining **VPUs** indicate the effect of decreasing parallel computations as the upper triangular matrix of the **FGE** is formed. This is expected since fewer arithmetic operations are carried out compared to the **FMM** algorithm.

6.7 Limitations of the Model

In order to facilitate implementation of the simulation model it is expedient to simplify the notional execution of the two programs in the simulator as follows:

- (1) no **S-Bus**, **V-Bus**, and **I-Bus** conflicts;
- (2) the **Central Control Unit** and **VPU**

- registers are always ready for use;
- (3) no pipelining of the arithmetic units;
 - (4) vector and some scalar calculations are based on 64-bit floating-point arithmetic;
 - (5) calculations on both programs are based on 16x16 matrices;
 - (6) Program execution occurs in lock step;
 - (7) Input-output assumed.

Further conclusions about the **NetVecX** architecture in this research suffer from the limitation that only two test programs were used. While the Gaussian Elimination is a fairly typical linear systems program, the matrix multiplication is not so typical. It is, nevertheless, an interesting extreme point in the program space. The conclusions are drawn with the above limitations in mind.

6.8 Parallelism in the Programs

The two programs used are fairly typical of the class of problems to be solved by the **NetVecX** system. The structure of both programs contains serial sections which are followed by highly parallel sections. The significance of this structure is related to the execution speed of the two sections. In the serial section of the two programs, the execution speed is approximately that of the Central Control

Unit, and thus the performance is adversely affected. The increase in performance obtained by the **NetVecX** is mainly from the parallel sections of the two programs, with the relative increase frequently limited by the number of **VPUs** available. In both these parallel sections, copies of vector instructions are executed in a number of **VPUs**. If a large number of different vector instructions are being executed, the overhead of loading vector instruction information into the **VPUs** makes parallel processing relatively expensive. From these two observations, it is clear that the **FMM** algorithm and the **FGE** example are well suited to the **NetVecX** architecture.

CHAPTER 7

CONCLUSIONS

The main objective of this study is to understand parallel and vector processing architectures and identify its bottlenecks and limiting factors. A new vector architecture is proposed and a simple performance model is used to examine the impact of certain architectural features on system performance. Also, a language compatible with the proposed architecture is introduced. In this final chapter we discuss the value of this research and assess possible future work in the areas identified.

This project studies high performance machine architectures for an **Open Network Environment** and proposes a class of vector architectures called **NetVecX**. A related aspect of the research project involves a review and comparison of currently implemented vector and parallel architectures with a view on identifying their limitations for exploiting new concepts of parallelism. These inadequacies partly provided the basis for considering alternative architectures in the high performance range for solving large scale scientific problems. The result of this study highlighted the lack of standardized modeling techniques for these classes of architectures.

The **NetVecX** architecture provides a new technique for employing a large number of dedicated vector processing units on a single problem to attain high performance. Simulation studies reveal that this organization has great potential for achieving high performance. This is illustrated by the significant increase in relative performance of the buffered system compared with unbuffered and sequential systems. The buffers used between the arithmetic units and the **D-Store** in each vector processing unit reduce the access rate to the **S-Store** and **D-Store** and therefore lead to improved system performance. The **NetVecX** organization is capable of obtaining high performance in two major ways. Firstly, in a high performance version of this architecture, the cache store between the local store and the multiple arithmetic units reduces the access rate to the local store and the common main store and hence increases the performance of the vector processor. Secondly, mapping a problem onto the hardware involves vectors in a single direction as opposed to two directions in an array processor.

In designing these high performance vector computers, it is very important to consider the effective exploitation of their parallel processing capabilities, bearing in mind the high cost of currently available vector computers. The proposed **NetVecX** architecture uses, in addition to the above factors, a number of methods to extract high processor

utilization, the most favored ones are:-

- (a) High level language constructs and instructions which allow users to exploit the parallel capabilities of the **NetVecX** without having to bother themselves with the machine idiosyncracies.
- (b) High speed scalar processor, which is of paramount importance because many applications involve some scalar processing operations which need to be processed sequentially in the shortest possible time, irrespective of the number of parallel units in the **NetVecX** system. With a large number of parallel processing units, the time taken to execute sequential or scalar operations becomes extremely significant with respect to the overall execution time.

The **NetVecX** should therefore be capable of executing each single operation in the shortest possible time to make the machine more viable.

The experiments conducted in this project reveal the following areas that require special attention; the memory subsystems, the **Central Control Unit (CCU)**, Unit set-up time, unpipelined arithmetic units, and the buffer stores.

The simulation results show that high performance can be achieved with fast memories. Also, a judicious separation of instructions from data greatly improves the **NetVecX** system performance. Since the **Central Control Unit(CCU)** combines the two functions of executing scalar instructions and controlling **Vector Processing Units(VPUs)**, it introduces a delay by holding the initiation of a vector unit until the complete execution of a scalar instruction. Significant performance improvements can be gained by separating the scalar processing function from the vector control function in a high performance derivative of **NetVecX** architecture.

Another source of time overhead is incurred during a vector unit set-up. As pointed out in **Chapter 6**, the unit set-up time is vector-operation dependent and becomes dominant for short vectors. Optimization of the techniques for initiating a vector processing unit would further improve the overall system performance.

To improve the execution speed of arithmetic operations, arithmetic units should be pipelined; and so doing requires considerable parallelism in the bandwidth of the central memory. As reviewed in **Section 3.1**, typical high performance vector computers have memory bandwidth of the order of hundreds or more memory banks to work in parallel. Since a much higher performance is envisaged for the **NetVecX** than

other vector computers, its memory bandwidth requirements needs to be in the thousands of millions of words per second; a concomitant requirement entails the use of hundreds of memory banks to work in parallel, which would always involve implementation difficulties from the engineering point of view. Thus the memory organization for the **NetVecX** would have to be carefully designed from that of currently available vector computers. This therefore would lead to the provision of vector cache mechanism to buffer frequently used vectors so that the access rate to the main store can be reduced. This implies that the **NetVecX** can attain a performance higher than that of currently implemented vector computers while using similar memory technology.

Even though a machine-independent programming language has yet to be implemented, preliminary work presented in this thesis indicates that such a language may be easier to implement for programming the **NetVecX**. For example, explicit control of the architectural features are transparent to the user and parallel constructs can be introduced to facilitate programming at a higher level.

The **NetVecX** simulator is not without its implementation difficulties. In particular the allocation and release of a vector processing unit, synchronizing the vector processing units and resolving memory clashes. These are areas that

require further research. This architecture, however, demonstrates a key role of vector processing and a step forward in the search for higher levels of computer performance.

A method for modelling systems to evaluate their performance by deterministic simulation is implemented. The experiments conducted show that a number of issues of performance evaluation for design optimization can be examined by this technique. There is, however, a wide range of choices for the level of detail at which the model simulates the system. Certainly, the level chosen for this study is higher than the logic gate level, where a lot of detail is irrelevant to performance evaluation. However, the complexity of the simulator at even this level suggests that higher levels may have to be examined. Further research should investigate the possibility of combining models of different levels of accuracy and complexity to yield useful and accurate performance information which would then be used to derive analytic functions to describe parts of the system and their interactions with specific large scale and scientific computer applications.

Appendix 1

Vector Mode Bits

The hardware assumes that sparse vectors are stored in the form of consecutive elements each consisting of a **Sub.No**, **VALUE** pair, where **Sub.No** (the Subscript Number of a non-zero element) is a 16-bit number and **VALUE** is of size defined by the **SIZE** bits. **Sub.No** is interpreted by the vector accessing hardware as a 2's complement number, with a negative value acting as an end marker. Thus sparse vectors can be up to 2^{15} elements long.

When adding two sparse vectors a result is generated for each **Sub.No** occurring in either input stream and **Sub.No**, **VALUE** pairs are written into successive locations in the destination vector. Adding a sparse vector to a dense vector produces another dense vector. The mode bits for the result vector must be set appropriately. When multiplying two sparse vectors together a result is generated only when a given **Sub.No** occurs in both input streams and **Sub.No**, **VALUE** pairs are written into successive locations in the destination vector. Multiplying a dense vector by a sparse vector produces another sparse vector. The mode bits for the result vector must be set appropriately.

0 0 0 Successive operands are read from or written to location at: **ORIGIN + n * STRIDE** where **STRIDE** is scaled according to element size

0 0 1 Successive operands are read from or written to location whose addresses are found at: **ORIGIN + n * STRIDE** where **STRIDE** is scaled as for a 32-bit quantity

1 0 0 Elements are treated as elements of a sparse vector each consisting of **Sub.No**, **VALUE**; successive elements are read from locations at: **ORIGIN + n** where **n** is scaled according to element **SIZE + 2** bytes operands are used from the first value with (**Sub.No** $\geq$ **SUBSCRIPT**) until a negative **Sub.No** is found. **Sub.No** is interpreted as a 2's complement number and each sparse vector must end with a negative **Sub.No** for which no corresponding **VALUE** is required. Thus when the result vector is a sparse vector the hardware plants a negative **Sub.No** in the 16-bit word following the last element written.

1 0 1 Each element is treated as element of a

supersparse array each consisting of **Sub.No ADDRESS**; successive elements are read from locations at: **ORIGIN + n** where **n** is scaled as for a 48-bit quantity. Accesses proceed as for mode **1 0 0** except that an extra access is required to obtain the element pointed to by the **ADDRESS**.

1 1 0 Each element is treated as an element of a multi-dimensional sparse array accessed in a direction other than the storing direction. The **SUBSCRIPT** register contains the required **Sub.No**. The addresses of the starts of each of **N** columns are read from: **ORIGIN + n * STRIDE** where **STRIDE** is scaled as for a 32-bit quantity for each column successive elements of the column are then accessed from: **COLUMN.START + n** where **n** is scaled according to element **SIZE + 2** bytes until an element with **Sub.No** equals required **SUBSCRIPT** number is found and used. If an element with **Sub.No** greater than required **SUBSCRIPT** number is found, or a negative **Sub.No** is found, the next **COLUMN. START** is accessed (until **n = N**) and that column is searched.

This mode cannot be used in conjunction with a result vector.

1 1 1 Elements are treated as elements of a multi-dimensional supersparse array accessed in a direction other than the storing direction. Accesses proceed as for mode **1 1 0** except that **n** is scaled as for a 48-bit quantity and an extra access is required to obtain the value pointed to by the **ADDRESS**.

This mode cannot be used in conjunction with a result vector.

Appendix 2

STANDARD OPERATORS, FUNCTIONS and PROCEDURES

(A) OPERATORS

OPERATOR	MEANING
< -	Operands v, n; W is formed from V by n left rotations, i.e. logical circular left shift.
- >	Operands v, n; W is formed from V by n right rotations, i.e. logical circular right shift.
*	Multiply
+	Add
-	Subtract
/	Divide
:=	Store
=	Equal to
<>	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
&	Vector logical "AND"
.	Vector logical "OR"
-=	Vector logical "NEQ"
<<-	Vector logical left shift a number of places as specified by the scalar operand.
->>	Vector logical right shift a number of places as specified by the scalar operand.

<\- <Vector arithmetic left shift a number
 of places as specified by the scalar
 operand.

</- Vector arithmetic right shift a number
 of places as specified by the scalar
 operand.

(B) BUILT-IN FUNCTIONS

FUNCTIONS	MEANING
SUM.ELEMENTS (U)	$S = u_0 + u_1 + \dots + u_{n-1}$
MULT.ELEMENTS (U)	$S = u_0 * u_1 * \dots * u_{n-1}$
AND.ELEMENTS (B)	$1 = b_0 \& b_1 \& \dots \& b_{n-1}$
OR.ELEMENTS (B)	$1 = b_0 b_1 \dots b_{n-1}$
NEQ.ELEMENTS (U)	$1 = u_0 \neq u_1 \neq \dots \neq u_{n-1}$
NORM(U)	$S = \text{SQRT}(u_0^2 + u_1^2 + \dots + u_{n-1}^2)$
MAX.ABS.VALUE (U)	$S = \text{Arithmetically MAX ABS}(u_i)$
MAX.VALUE (U)	Arithmetically largest value of the elements in U.
MAX.VALUE.INDEX (U)	$n = \text{Index of largest positive value of the elements in U.}$
INSQR(U)	The first n values defined by U are each squared and added together and the result copied to the location specified by the user.
INPROD(U,V)	The first n values defined by U are multiplied element by element by the first n values defined by V and the sum of the result copied to the location specified by the user.
MAX.ABS.INDEX (U)	$n = \text{Index of largest element in U.}$
MIN.ABS.VALUE (U)	$S = \text{Arithmetically MIN.ABS}(U_i)$

MIN.ABS.INDEX(U)	n = Index of smallest element in U.
MIN.VALUE(U)	S = Arithmetically largest negative element in U.
MIN.INDEX(U)	n = Index of largest negative element in U.
NEGATE(U)	Changes the sign of the vector U.
SELECT.INDEX.EQ(U,V)	Finds the index of the first occurrence of a pair of value of U and V satisfying the specified condition.
SELECT.INDEX.OP(U,V)	Operator OP, where OP is EQ, NE, GT, GE, LT, LE.
RANK.OP(U,S)	Finds the index of the first occurrence of S in U which satisfies the required condition
specified by OP where	OP is EQ, NE, GT, GE, LT, LE.
COUNT.ELEMENTS.OP(U,S)	At the end of each operation, returns the count of the number of elements within the first n operand values defined by U which satisfy the specified condition indicated by OP, where OP is EQ, NE, GT, GE, LT, LE.

(C) BUILT-IN PROCEDURES

These procedures are similar to the built-in functions but they do not return a result.

PROCEDURE	MEANING
SWAP.ALL(U,V)	u_i and v_i are exchanged.
SWAP.PART(U,V,n)	First n values of U and V are exchanged.

COMPRESS.OP(R,U,S)

Copies selected elements of (normally) dense U to normally sparse result vector, R. If U is sparse the operation terminates after n operations or when the last element of U is encountered. S is the scalar value each element is compared with and OP is one of the relational operators EQ, NE GT, GE, LT, LE.

EXPAND(R,U,S)

Elements of (normally) sparse U are copied to locations in the result vector, R, with the value, S, filling non-existing U values; the number of elements of R is the new bound. If U is not sparse copy operation takes place from U to R all the same.

SMOVE(R,U,S,n)

First n values defined by U are copied to the first n locations of the result vector R; the remaining elements of R filled with the value in S.

VMOVE(R,U,V,n)

First n values of vector U are copied to the first n locations defined by the result vector R; the remaining elements of R are copied from successive locations defined by V.

Appendix 3

NetVecX Vector Extensions

1.1 Introduction

The NetVecX Programming and Specification Language (NPSL) is a super set of Parallel Pascal (PP) in the sense that some of its features have been modified and vector processing capabilities have been added to it. Users are advised to consult [81,87,101] for explanation of the full facilities and capabilities of Parallel Pascal (PP).

The new facilities required include multi-dimensional sparse and dense arrays and manipulations of whole vectors of multi-dimensional arrays. Specification of the extend of parallelism has also been added.

Areas Considered for Extension

The areas considered for extension are:-

- (a) Introduction of new reserved word delimiters.
- (b) Declarative statements.
- (c) Array referencing operations.
- (d) Computational statements.

1.2 New reserved Word Delimiters

New reserved word delimiters are required in order to indicate the extent of parallelism and to differentiate between sparse and dense multi-dimensional arrays. These are:-

- (a) SPARSE
- (b) PAR
- (c) SIM
- (d) INITIALIZE
- (e) ANY
- (f) All

Their meanings will be made clear in the context in which they are used in the following sections.

2.0 DECLARATIVES

The declarative statements are mainly concerned with multi-dimensional arrays of which there are two kinds, i.e.,

- (a) SPARSE arrays and
- (b) DENSE arrays

They serve similar functions as the general declarative statements of the base language (PP). The extended syntax for the variable declaration is as follows:-

<VAR.DEC> ::= <NAME.LIST> <TYPE>

A variable declaration specifies an arbitrary list of names of variables of a given type. They may be scalar, vector and multi-dimensional sparse or dense array variables depending on whether or not the appropriate dimensions are given in:-

<TYPE> ::= <SCALARTYPE> | <ARRAYTYPE>

<ARRAYTYPE> ::= [<SPARSE> | <NIL>] <SCALARTYPE> <[> <CONST.LIST> <]>

<CONST.LIST> ::= <CONST> [, <CONST.LIST> | <NIL>]

Scalar types are permitted in multi-dimensional arrays and vectors. These are real and integer; the other interpretations are as defined in the basic manual.

3.0 INITIALIZATION

INITIALIZE is used to initialize a sparse multi-dimensional array statically. The data can be embedded in the source program text or read from the currently selected input stream. It contains only the non-zero elements of the array with information as to the position of each element specified thereof. The syntax is as follows:-

<INT.ST> ::= INITIALIZE <ARRAYNAME> [#data# | <NIL>]

The second option specifies that the data is to be read from the currently selected input stream. The multi-dimensional array is first transformed into a two-dimensional array and the data for each row having a non-zero element is listed in the form :-

Row.no : Col.no/Value

4.0 ARRAY REFERENCING OPERATIONS

The extensions have the capabilities of

- (a) Accessing an element of a multi-dimensional array.
- (b) Accessing a vector of multi-dimensional array in any dimension.
- (c) Accessing sub-vectors of a multi-dimensional array in any dimension given the lower and upper subscripts.

4.1 Operand Forms

The syntax for operand definition has been extended to include multi-dimensional array referencing operations. The basic operand defining structure has been retained except for the modification of the <SUBSCRIPT> option which is shown below. The user is advised to consult [81,87,101] for full operand capability.

```

<SUBSCRIPT> ::= <[><SUB><]> | <NIL>
<SUB>      ::= <SUB.COMP>[,<SUB.COMP> | <NIL>]
<SUB.COMP> ::= <COMPUTE> | <DIAGSYMB> | *|[(<COMPUTE> <END.SUB>)] | <NIL>
<DIAGSYMB> ::= [\, \ | /, /]
<END.SUB>  ::= <COMPUTE> | <NIL>

```

NOTE: The "*" symbol implies accessing a vector in the specified dimension. The second option of <SUB.COMP> may only appear once in a multi-dimensional array operand, since it is only a vector that can be processed at a time. <DIAGSYMB> is only allowed to specify the last two dimensions in a multi-dimensional array when accessing the diagonals.

EXAMPLES

A[0]	Denotes one element, element 0 of vector A.
A[I]	Denotes the Ith element of a vector A.
A[*] or A	Denotes all elements of the vector A.
A[* (I)]	Denotes the sub-vector of A starting from the Ith to the last element of the vector.
A[* (I:L)]	Denotes the sub-vector of A from element I to element L of the vector inclusively.
B[I, *(J:L), K]	Denotes the subvector of B in plane I starting from row J to row L of column K inclusively.
B[*, J, K]	Denotes a vector of a column K of row J of all planes in B.
C[\, \]	Denotes the main diagonal elements of the array C.
C[/, /]	Denotes the minor diagonal elements of array C.

5.0 IMPERATIVE STATEMENTS

In the Imperative statements, the modified options the computational statements which now incorporates the concept of the extend of parallelism denoted by the <PARALLEL.ST> and whole vector processing. The syntax is given below:-

```

<COMP.ST>      ::= <COMPUTATION> | <IF.ST> | <WHILE.ST> | <FOR.ST> |
                  <PARALLEL.ST>

<PARALLEL.ST>  ::= [ <SIM.ST> | <PAR.ST> ] DO <STATEMENTS>

<SEQPAR.ST>    ::= SIM | PAR[ <COMPUTATION> , | <'<'> | <COMPUTATION>

```

<PAR.ST> ::= SIM|PAR

The <PARALLEL.ST> informs the compiler about the group of statements enclosed inside the DO block which may be processed simultaneously. The examples explain how these are used.

6.0 CONDITIONS

<CONDITION> ::= {ANY|ALL|<NIL>}<TEST>[<LOGOP><CONDITION>|<NIL>]

<TEST> ::= <[><CONDITION><]>|<COMPUTATION><COMPARISON>

<LOGOP> ::= OR|AND

The simplest form of condition is a TEST in the form of a COMPARISON applied to the result of a COMPUTATION. The syntax for COMPARISON is :-

<COMPARISON> ::= <COMPARATOR><COMPUTATION>[<COMPARISON>|<NIL>]

<COMPARATOR> ::= <> | < | > | | <= | >= | =

If vectors are involved in a test then their lengths and types must be the same. Except in situations where a test involves a vector and a scalar value. In this case, the test is evaluated for each element of the vector and the scalar and if found to be true the appropriate statements of the construct are applied as required.

Vector quantifiers may be used in a test. These are ANY and ALL, which may be used to apply a test across a subvector or the whole vector, in the sense that if at least one element obeys the test or if all the elements obey the test respectively. Since these vector quantifiers reduce to a single value for the test, they are governed by the rules for a scalar test.

BIBLIOGRAPHY

- [1] Albrecht A and Melhorn K 1987 Parallel Algorithms and Architectures Lecture Notes in Computer Science 269(Berline:Springer).
- [2] Babb II R G Programming Parallel Processors Reading MA: Addison-Wesley, 1988.
- [3] Bacon J The Motorola 68000 Englewood Cliffs, NJ: Prentice-Hall, 1986.
- [4] Barlow R H, Evans D J, Newman I A and Woodward M C 1981 The NEPTUNE Parallel Processing System Internal Report Department of Computer Studies, Loughbrough University, UK.
- [5] Barlow R H, Evans D J and Shanehchi J Performance Analysis of Algorithms on Synchronous Parallel Processors Comput. Phys. Commun. 26, 1982, pp 233-6.
- [6] Barnes G H, Brown R M, Kato M, Kuck D J, Slotinick D L and Stokes R A 1968 The ILLIAC IV Computer IEEE Trans. Comput. C-17 746-57.
- [7] Basili V R and Knight J C A Language for Vector Machines SIGPLAN Notices 10(3) 1975 pp 39-43.
- [8] Baskett F and Keller T W (1977) An Evaluation of the CRAY-1 Computer In Kuck, Lawrie, and Sameh (1977) 71-84.
- [9] Baudet G M (1978) Asynchronous Iterative Methods for Multiprocessors J ACM 25 pp 226-24.
- [10] Ben-Ari M (1982) Principles of Concurrent Programming Englewood Cliffs, NJ: Prentice Hall.
- [11] Bernstein A J (1966) Analysis of Programs for Parallel Processing IEEE Trans. on Comput. Oct pp 757-763.
- [12] Burroughs (1979) Final Report, NASF Feasibility Study NASA Contractor Report, NAS2-98897.
- [13] CDC (1979) Final Report, NASF Feasibility Study NASA Contractor Report NAS2-98896.
- [14] Chen S S, Dongarra J J, and Hsiung C C 1984 Multiprocessing Linear Algebra Algorithms on the CRAY X-MP-2: Experiences With Small Granularity J Parallel Distrib. Comput. 1 22-31.

- [15] Chin-Wen Ho and Lee R C T (1990) A Parallel Algorithm for Solving Sparse Triangular Systems IEEE Trans. Comput. vol 39, no 6, pp484-52.
- [16] Cosnard M, Quinton P, Robert Y, and Tchente M (1986) Parallel Algorithms and Architectures Amestrdam: North-Holland.
- [17] Cray 1980 CRAY-1 computer systems library reference manual Publication SR-0014(Mendota Heights, Minn.: Cray Research Inc.).
- [18] CRAY X-MP Series of Computer Systems Minneapolis, Minn.: Cray Research Inc., 1985.
- [19] CRAY-1 Computer System Hardware Reference Manual Minneapolis, Minn: Cray Research, Inc., 1977.
- [20] De Vel O Y and Krishnamurthy E V (1987) An Iterative Pipelined Array Architecture for Generalized Matrix Inversion Information Processing Lett., 26, 263-7.
- [21] Dennis J B (1980) Dataflow Supercomputers Computer 13(11), 48-56.
- [22] Desrochers G R (1987) Principles of Parallel and Multiprocessing New York, NY: McGraw-Hill.
- [23] Di Nucci D N (1987) In Programming Parallel Processors (Babb II R G ed.). Reading, MA: Addison-Wesley.
- [24] Dijkstra E W (1968) Cooperating Sequential Processes In F Genuiys(Ed), Programming Languages, Academic Press, New York, 43-112.
- [25] Dogarra J J (1987) Experimental Parallel Computing Architectures Amestrdam: North-Holland.
- [26] Eager D L, Zahorjan J, and Lazowska (1989) Speedup Versus Efficiency in Parallel Systems IEEE Trans. on Computers.
- [27] Elrod T H (1970) The CDC 7600 and Scope 76 Datamation April pp 80-85.
- [28] Enslow P H (1977) Multiprocessor Organization ACM Comp Surv 9(1), 103-129.
- [29] Evans D J 1980 On the Solution of Certain Toeplitz Tridiagonal Linear Systems SIAM J. Numer. Anal. 17 657-80.

- [30] Flynn M J 1972 Some Computer Organization and Their Effectiveness IEEE Trans. Comput. C-21 948-60.
- [31] Flynn M J (1980) Directions and Issues in Architecture and Languages Computer, 13(10), 5-22.
- [32] Forsythe G E and Wasow W R 1960 Finite-difference Methods for Partial Differential Equations(London: John Wiley).
- [33] Foster M J and Kung H T 1980 The Design of Special-Purpose VLSI Chips IEEE Computer 13(1) 26-40.
- [34] Fuss D and Hollenberg J 1984 A Comparison Between Japanese and American Supercomputers Supercomputer 3 8-9.
- [35] Geers J 1989 A New Generation of Benchmarking Mips, vol. 1, no 1, 92-100.
- [36] Gehani N and McGettrick A D (1988) Concurrent Programming Wokingham: Addison-Wesley.
- [37] Graham W R 1970 The Parallel and Pipeline Computers Datamation 16(4) 68-71.
- [38] Gurd J R 1987 Data Flow Architectures Major Advances in Parallel Processing ed C R Jesshope 113-132 (Aldershot: Technical).
- [39] Handler Wolfgang (1982) Innovative Computer Architecture- How to Increase Parallelism but not Complexity of Parallel Computer Systems An Advanced Course Edited by D J Evans, Cambridge University Press.
- [40] Heller D 1978 A survey of Parallel Algorithms in Numerical Linear Algebra SIAM Rev. 20 740-77.
- [41] Higbie L C (1978) Applications of Vector Processing Computer Design 17(2).
- [42] Hinnant D F Benchmarking: Art, Science, or Black Magic? IEEE Micro Magazine, vol 8 pp 64-75 October 1988.
- [43] Hockney R W and Jesshope C R 2nd ed Parallel Computers, Philadelphia, Pa.: Adam Hilger, 1988.
- [44] Hockney R W 1977 Supercomputer architecture - Infotech State of the Art Conference: Future Systems Chairman F Summer (Maidenhead: Infotech) 277-305.

- [45] Hwang K and Briggs F A (1984) Computer Architecture and Parallel Processing, New York, NY: McGraw-Hill
- [46] Iverson K E (1962) A Programming Language New York, NY: John Wiley.
- [47] Jensen C (1978) Taking Another Approach to Supercomputing Datamation 24(2) 159-75.
- [48] Johnson P M 1978 An Introduction to Vector Processing Computer Design 17(2) 89-97.
- [49] Johnson T L 1986 A Comparison of MC68000 Family Processors Byte vol 11 pp 205-218.
- [50] Keyes R N (1975) Physical Limits in Digital Electronics Proc. IEEE vol 63 no 5 pp 740-767.
- [60] Kuck D J, Lawrie D H, and Sameh A (Eds) (1977) High Speed Computer and Algorithm Organization New York, NY: Academic Press.
- [61] Kuck D J 1968 ILLIAC IV Software and Application Programming IEEE Trans. Comput. C-17 758-70.
- [62] Kuck D J 1977 A Survey of Parallel Machine Organization and Programming Comput. Surv. 9 29-59.
- [63] Kung H T, Sproull R, and Steele G. (1981) VLSI Systems and Computations Rockville, Md: Computer Science Press.
- [64] Lawrie D H et al 1975 GLYPNIR - A Programming Language for ILLIAC IV Commun. Assoc. Comput. Mach. 17 157-64.
- [65] Li K-C and Schwetman H 1985 Vector C: A vector Processing Language J. Parallel and Distribut Comput., 2-2, 132-169.
- [66] Lincoln R A Safari Through the Control Data Star-100 with Gun and Camera AFIPS National Computer Conference 1978.
- [67] Lipovski G J and Mirowski M (1987) Parallel Computing John Wiley and Sons, New York, p 2.
- [68] MacDougall M H (1975) System-Level Simulation. Digital System Design Automation: Language, Simulation, and Data Base Rockville, Md.: Computer Science Press.
- [69] MacDougall M H (1979) The Simulation Language SIML/I Proc AFIPS National Computer Conf., 48 (June) 39-44.

- [70] MacDougall M H (1984) Instruction-Level Program and Process Modeling Computer 17(7) 12-24.
- [71] MacDougall M H (1987) Simulating Computer Systems Cambridge, Mass: The MIT Press.
- [72] Maruyama K (1973) The Parallel Evaluation of Matrix Expressions IBM T J Watson Research Center, Yorktown Heights, New York.
- [73] Mead C A and Conway L (1980) Introduction to VLSI Systems Reading, Mass.: Addison-Wesley
- [74] Miranker W L 1971 A Survey of Parallelism in Numerical Analysis SIAM Rev. 13 524-47.
- [75] Morris D and Ibbett R (1979) The MU5 Computer Systems London:Macmillan.
- [76] Motorola (1980) Motorola Semiconductors - Advance Information Motorola Inc. ADI-814-R1.
- [77] Padeges A, Moore B B, Smith R M, and Bachholz W (1988) The IBM System/370 Vector Architecture: Design Considerations IEEE Trans. Comput., vol 37, no 5, pp 509-520.
- [78] Pease M C 1968 An Adaption of the Fast Fourier Transform for Parallel Processing J Assoc. Comput. Mach. 15 252-64.
- [79] Perrot R H(1987) Parallel Programming Wokingham: Addison-Wesley.
- [80] Perrot R H and Zorea-Aliabadi A (1986) Supercomputer Languages ACM Comp Surv 18(1),5-22.
- [81] Perrot R H, Lyttle R W and Dhillon P S (1987) The Design and Implementation of a Pascal-Based Language for Array Processor Architecture J Parallel and Distributed Computing 4 266-87.
- [82] Perrott R H 1979 A language for Array and Vector Processors ACM Trans. Programming Languages Systems, 2, 177-95.
- [83] Poole W G Jr and Voight R G 1974 Numerical Algorithms for Parallel and Vector Computers: An Annotated Bibliography Comput. Rev. 15 379-88.
- [84] Ramamoorthy C V and Li H F (1977) Pipeline Architecture ACM Comp. Surv. 9(1), 61-102.

- [85] Ramamoorthy C V and Gonzalez M J A survey of the Techniques for Recognizing Parallel Processable Streams in Computer Programs AFIPS - FJOC (1969) pp 757-763.
- [86] Reddaway S F 1973 DAP - a Distributed Array Processor, 1st Annual Symp. on Comput. Architecture (IEEE/ACM) Florida.
- [87] Reeves A. P. 1984 Parallel Pascal for Parallel Computers J. Parallel and Distribut. Comput. 1-1, 64-80.
- [88] Rumbaugh J (1977) A dataflow Multiprocessor IEEE Trans Comput C-26(2), 138-146.
- [89] Russell R M (1978) The CRAY-1 Computer System Comm ACM 21(1) 63-72.
- [90] Sameh A H 1977 Numerical Parallel Algorithms - A survey In Kuck, Lawrie, and Sameh(1977) 207-228.
- [91] Schwartz J T 1980 Ultracomputers ACM Trans. Prog. Lang. Syst. (2) 484-521.
- [92] Stevens K G Jr 1975 CFD - A FORTRAN-like Language for the ILLIAC IV SIGPLAN Not. 10(3) 72-80.
- [93] Stone H S (Ed) (1980) Introduction to Computer Architecture Chicago, Ill : SRA.
- [94] Sumner F H (Ed) 1982 State of the Art Report: Supercomputer System Technology Series 10 no 6 (Maidenhead: Pergamon Infotech Ltd).
- [95] Thornton J E 1970 Design of a Computer - The Control Data 6600 (Glenview, Ill: Scott, Foresman and Company).
- [96] Toong H D and Amar G An Architectural Comparison of Contemporary 16-bit Microprocessing IEEE Micro May 1981 pp 26-37.
- [97] Treleaven P C, Brownbridge D R and Hopkins R P 1982 Data-Driven and Demand-Driven Computer Architecture Comput. Surv. 14 93-143.
- [98] Voigh R G 1977 The Influence of Vector Computer Architecture on Numerical Algorithms In Kuck, Lawrie, and Sameh(1977) 77-86.
- [99] Watson W J 1972 The TI-ASC A Highly Modular and Flexible Super Computer Architecture AFIPS FJOC vol 41 Pt 1.

- [100]Wedel D (Miss) FORTRAN for the Texas Instruments
ASC System SIGPLAN Notices 10(3) 1975 p 119.
- [101]Welsh J and Bustard D W (1979) Pascal Plus - Another
Language for Modular Multiprogramming Software Practice
and Experience, 9, 947-57.
- [102]Wethrell C (1980) Design Cosiderations for Array
Processing Languages Software Practice and Experience
vol 101 pp 265-271.
- [103]Whiddett D (1987) Concurrent Programming for Software
Engineers New York, NY : John Wiley.
- [104]Wulf W A and Bell C G 1972 c.mmp, A Multi-
Miniprocessor AFIPS vol 41 Pt 11 FJOC pp. 765-777.
- [105]Wulf W A and Bell C G Future Computer Requirements for
Computational Aerodynamics NASA CP-2032 Oct. 1977.
- [106]Zosel M E A Modest Proposal for Vector Extensions to
Algol SIGPLAN Notices 10(3) 1975 p 62.
- [107]Zwakenberg R G Vector Extensions to LRLTRAN SIGPLAN
Notices 10(3) 1975 pp 77-86.

Autobiography

The author graduated in June 1965 from Cairo University, Egypt with a B.Sc. degree in Special Physics, with Honors and Distinction. He subsequently joined Cairo Military Academy and graduated in July, 1966, with a Diploma in Military Science. From August, 1966, until July 1972, he worked in various positions with the Egyptian Military Forces. He was a Lecturer in Physics and Electrical Engineering at Cairo Military Academy from 1966 to 1969. In September 1970, During his career with the Egyptian Army, he was posted on a Military education mission to the Soviet Union where he completed a course on Strategic Chemical-War Defence at Moscow Military Academy of Chemical War.

In 1973 the author returned to Cairo University to Lecture in Physics and to supervise the Faculty of Science Library while pursuing a Masters degree in Theoretical Solid State Physics. In November 1973, he was awarded the MS degree in Solid State Physics.

In 1974 the author was awarded a Ford Foundation Fellowship to study Library and Information Science in the United States of America. In December, 1976, he was awarded the MLS degree from Pittsburgh University, School of Library

and Information Science.

During his study at the University of Pittsburgh, the author was fascinated by the rapid development in Computer Science and Technology and decided to pursue a course of study in this new and exciting field. In 1979 he began the study of Computer Science at the School of Engineering, City College of the City University of New York, and subsequently was awarded the MS degree in Computer science in February 1980. In 1981 He joined the Computer Science Department of the City University of New York and has subsequently been conducting the research that has lead to this dissertation. Since 1984, the author has been on the Faculty of the Computer Science Department of the State University of New York College at Old Westbury.

The End

Said M. Mahmoud