1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian-Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

A DISTRIBUTED CONTROL DATA FLOW MACHINE.

by

DOUGLAS R. SMITH

A Thesis submitted to the
Faculty of Graduate Studies and Research
In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University

January 1986

© copyright

1986, Douglas R. Smith

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN - 0-15-29865-0

The undersigned hereby recommends to the Faculty of
Graduate Studies and Research acceptance of the thesis,

A DISTRIBUTED CONTROL DATA FLOW MACHINE

submitted by DOUGLAS R. SMITH, M.ENG., in partial
fulfillment of the requirements for the degree of Doctor
of Philosophy.

Thesis Supervisor

External Examiner

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

January, 1986

KEYWORDS

Data Flow Architectures

Computer Simulation

Multiprocessor Architecture

Parallel Processing

Concurrent Computation

ABSTRACT

Data flow architectures have not found wide application as DSP processors because the efficient execution of DSP algorithms requires the use of a multiple operand and multiple destination format instruction and both primitive and function level operations.

All of the three contemporary data flow implementation strategies suffer cost-performance degradation when extended to accommodate these requirements. The MIT cell-block architecture requires the addition of costly hardware and complex control logic. The Manchester ring can not support this instruction format without multiple searches into the associative memory with an accompanying decrease in performance. The Aarnio DSP architecture cannot support primitive level instructions due to the extensive overhead in communicating the operands.

A novel data flow architecture, which has a variable size instruction template and can concurrently execute both primitive and function level operators, is proposed. The architecture is modular so that a minimum cost configuration can be tailored to achieve a specific performance specification. A simulation of the architecture demonstrated a twofold performance improvement over that obtained with a two operand two destination format instruction for a class of programs found in digital signal processing and control applications.

ACKNOWLEDGEMENTS

I would like to thank DR. B. A. Bowen for suggesting this area of research. His guidance and encouragement over the years has been greatly appreciated.

None of this work would have been possible without the co-operation and support of my superiors and the Department of National Defence. In particular, I would like to thank DR. J. D. Wilson and DR. P. E. Allard, past and present heads of the Department of Electrical Engineering at the Royal Military College for their assistance in balancing work loads and providing an invigorating academic work environment. I would also like to thank Mrs. Betty Allen and her staff at the RMC computing centre for their un-failing support and advice.

Most important is my gratitude to my family who have been deprived and neglected on many occasions during the past years. Their support and assistance ultimately made this work possible.

CONTENTS

1.	INTRODUCTION	
1.1	Overview	1
1.2	Statement of the Problem	3
1.3	Objectives of this Research	6
1.4	Thesis Outline	7
2.	BACKGROUND	
2.1	Introduction	9
2.2	Data Flow Architectures	11
	2.2.1 Data Flow Program Graphs	
	2.2.2 Data Flow Computer Organization	
2.3	Maximizing Concurrent Execution of Program Graphs	27
	2.3.1 Introduction	
	2.3.2 Instruction Processing in a Static Machine	
	2.3.3 Instruction Processing in a Dynamic Machine	
	2.3.4 Static and Dynamic Performance Comparisons	
2.4	Description of a Practical Data Flow Processor	47

2.5	Performance Constraints	51
2.5.1	Introduction	
2.5.2	A Note on CAM Technology	
2.5.3	Imposed Sequentiality	
2.5.4	Analysis of Typical Program Constructs	
2.5.4.1	Introduction	
2.5.4.2	Linear Operators	
2.5.4.3	Matrix Operations	
2.5.4.4	The Kalman Filter	
2.5.5	An Ideal Static Simulator	
2.5.5.1	Overview	
2.5.5.2	Mechanism	
2.5.5.3	Simulation Results	
2.6	Chapter Summary	81
3.	A PRACTICAL MEDIUM SCALE PROCESSOR	
3.1	Introduction	82
3.2	Description of the Architecture	83
3.2.1	Overview	
3.2.2	The Slotted Ring	
3.2.3	Processing Elements	
3.2.4	Token Storage Units	
3.2.5	Discussion	
3.3	Chapter Summary	112

4.	SIMULATED PERFORMANCE OF THE SLOTTED RING MACHINE (SRM)	
4.1	Introduction	114
4.2	Simulation Model	115
	4.2.1 Overview	
	4.2.2 SIAM Model	
	4.2.3 Control of the Simulation	
4.3	Modelling	119
	4.3.1 Model Instrumentation	
	4.3.2 Description of the Simulated Topologies	
4.4	Summary of Simulation Results	125
4.5	Chapter Summary	134
5..	COMPARISONS AND DISCUSSION	
5.1	Introduction	135
5.2	Program Comparisons	136
5.3	Performance Comparisons	144
	5.3.1 Linear	
	5.3.2 Matrix Multiplication	
5.4	Chapter Summary	150

6.	SUMMARY AND RECOMMENDATIONS FOR FURTHER WORK	
6.1	Summary of the Work Presented	152
6.2	Recommended Areas of Investigation for Further Work	156
6.2.1	Conditional and Loop Structure Support	
6.2.2	Data Structures Mechanism	
6.2.3	Error control and Interrupt Environment	
6.2.4	Implementation Issues	
6.2.5	Extensions to the Basic	
6.2.6	Program File Organization Architecture	
	LITERATURE CITATIONS	165
	Appendix A. SUMMARY OF SIMULATION RESULTS .	177
	Appendix B. SIMULATOR SOFTWARE	194
	Appendix C. BIBLIOGRAPHY	206

LIST OF FIGURES

<u>Figure</u>	<u>Name</u>	<u>Page</u>
2.1	Data Flow Operator Node	13
2.2	Generalized Data Flow Operator	14
2.3	Node Firing Rules	18
2.4	Data Flow Program Graph	19
2.5	Instruction Form of a Data Flow Program	22
2.6	Activity Template Form of a Data Flow Program	23
2.7	Block Diagram of a Data Flow Computer	26
2.8	Node firing rules for Switch and Merge Operators	28
2.9	Fundamental Program Loop Constructs	29
2.10	Program Graph for an Iterative Solution	30
2.11	Recursive Computation in a Program Graph	33
2.12	The U-Interpreter	39
2.13	The Manchester Data Flow Computer	46
2.14	Binary Tree for a Series Summation	56
2.15	A Signal Processing Application for a Data Flow Processor	60
2.16	Kalman Filter Data Flow Graph	65
3.1	Slotted Ring Architecture	86

3.2	Ring Slot Mechanism	88
3.3	Form of an Instruction Template	89
3.4	Block Diagram of a Processing Element	99
3.5	Block Diagram of the Arithmetic Processing Unit	100
3.6	Block Diagram of an Interface Processing Unit	101
3.7	Block Diagram of the Template Storage Unit	109
4.1	Slotted Ring Machine Configuration	124

TABLES

<u>Table</u>	<u>Name</u>	<u>Page</u>
2.3.1	Static Computer Operation	42
2.3.2	Dynamics of the U-Interpreter System	43
2.3.3	Execution Frequency of Operations for a Dynamic Data Flow Machine	44
2.3.4	Comparative Static-Dynamic Performance	45
2.5.1	Linear Operator Programs Names	69
2.5.2	Linear Operator Program Formats	70
2.5.3	Simulated Results - Linear Operator (Ideal)	72
2.5.4	Matrix Multiplication Program Names	75
2.5.5	Simulated Results - 8x8 Matrix Multiply	76
2.5.6	Simulated Results - 5x5 Matrix Multiply	77
2.5.7	Simulated Results - Square Matrix Execution Times	78
2.5.8	Simulated Results - Kalman Filter	80
3.2.1	Header Slot Coding	90
3.2.2	Operand Slot Coding	94
3.2.3	Destination Slot Coding	95
3.2.4	Flow Diagram for the Interface Unit Operation	104

4.4.1	Simulated Results - Linear Operator (SRM),	126
4.4.2	Matrix Multiplication Program Formats	128
4.4.3	Square Matrix Multiply - Execution Times	130
4.4.4	Square Matrix Multiply - Result Wait Times	131
4.4.5	Square Matrix Multiply - Processor Utilization	132
4.4.6	Square Matrix Multiply - Ring Utilization	133
5.2.1	Program Characteristics Linear Operator Version 1	139
5.2.2	Program Characteristics Linear Operator Version 2	140
5.2.3	Program Characteristics 8x8 Matrix Multiply	141
5.2.4	Program Characteristics 5x5 Matrix Multiply	142
5.2.5	Program Characteristics Kalman Filter	143
5.3.1	Potential Performance Improvement Ratio Ideal and SRM Machines	146
5.3.2	Figure of Merit Ideal and SRM Machines	147

LIST OF ABBREVIATIONS

CAM	Content addressable Memory
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
FIFO	First In First Out
K	the value 1024
IC	Integrated circuit
MIPS	Millions of Instructions Per Second
MOS	Metal Oxide Semiconductor
RAM	Random Access Memory
TTL	Transistor Transistor Logic
usec	micro-second
nsec	nano-second

CHAPTER

INTRODUCTION

1.1 Overview

The demand for digital computers to have more computational power and the advent of powerful VLSI processors, has added emphasis to a continuing interest in the design of multiprocessor and special purpose parallel processing systems. The search for very high speed computing machinery must, according to Dennis, [Dennis80a] meet three challenges:

- a. achieve high performance at an acceptable cost;
- b. exploit the potential of LSI technology; and

c. produce a machine that is possible to program in a manner such that the performance potential may be exploited.

Several computers have been synthesized from a collection of smaller processors, for example ILLIAC IV, CM*, and C.mmp [Barnes68, Fuller77, Wulf78]. However, these multiprocessors have not yet achieved the ease of programming and level of performance sought. VLSI advances now challenge the computer designer to create architectures incorporating thousands of asynchronously co-operating processors. In these new architectures, programmers should not be concerned with how a program is partitioned into concurrently executable pieces nor how these pieces co-ordinate. With potentially thousands of active processors, programmers should not require a knowledge of the numbers or physical arrangement of the processors to map their algorithms efficiently onto the hardware. Instead, it should be possible to automatically decompose a program into small pieces that can be executed asynchronously with minimum interference from one another.

Several researchers [Arvind77, Bachus78, Glushokov74] have concluded that this can only be achieved, with significant changes to the widely accepted "von Neumann model".

1.2 Statement of the Problem

Data flow architectures which have a two operand, two destination instruction format do not extract the maximum horizontal parallelism available in the program graph. This can be attributed to the requirement to copy data values out of program nodes when they are required at more than two destinations, and to include a sequentially executed program subgraph under circumstances when certain primitive operations are to be performed on a series of operands. When these instruction format imposed constraints are relaxed, the maximum horizontal parallelism, and hence the minimum execution time can be achieved.

Data flow processors depend on an intrinsic mechanism of some sort to recognize instructions which are complete and ready to ~~execute~~. Several implementation strategies are used to effect this "ready" instruction detection mechanism.

Circular pipelined architectures such as the Manchester data flow computer [Watson82] use a content addressable memory for this function. Boyer shows that this component is the critical element in the operation of this computer and that the capacity of the machine to consume instructions depends directly on the speed at which completed instructions can be recognized. [Boyer84]. The binary match, no-match nature of the CAM is used to determine if instructions can be fired. This mechanism can not be extended to accommodate multi-operand instructions without requiring multiple searches into the CAM and sacrificing its performance potential.

Dennis suggests a hardware intensive register based construct for executing data flow programs [Dennis75]. In this architecture, an instruction forms in a cell which contains a register for each expected

operand. Supervisory logic can easily detect the presence of two operands and form the instruction template. The number of registers in each cell could be increased to accommodate multi-operand instruction formats. The supervisory logic required would become more involved, and it would require the appropriate conditioning to detect the required number of operands. Further, it is not clear if a scheme could be devised to dynamically allocate the required number of registers to an instruction template and thus maintain a high hardware register utilization. Alternatively, an instruction format with a fixed number of operands would have to be adopted.

A combination of hardware and software is used in some architectures to implement the instruction ready detection mechanism [Aarnio80, Caluwaerta83, Cornish79, Kronloef81, Takahashi83]. In general, these schemes involve a software driven search and comparison of instruction associated status and format words. This process is normally performed by an intelligent memory controller used to store and update the instruction templates. This scheme is easily extended to

accommodate multi-operand instruction formats by modifying the algorithm used. The format control word would be initialized at compile time to reflect the desired number of operands. The status word could be updated and compared by the memory controller on the arrival of each operand. The implementation must make efficient use of memory and achieve the highest possible speed. With respect to the hardware schemes described above, this approach is slow and can only be effective when the granularity of the machine basic operation is elevated to the function level [Aarnio80].

1.3 Objectives of this Research.

This thesis proposes a data flow computer architecture which can accommodate multiple operand and multiple destination format instruction templates without the performance degradation that would accompany an enhanced memory controller or extended function CAM implementation..

The architecture incorporates a mechanism to accommodate variable size instruction templates. As any number of input operands and destinations can be included, the computer can accommodate a concurrent mix of primitive and functional level operators.

The system architecture allows the number of processor elements and the topology to be tailored to a given problem. Thus a practical cost-effective data flow computer can be implemented for a specific performance specification. The functional components are modularized to take maximum advantage of current and future VLSI technology.

1.4 Thesis Outline

Chapter two examines the performance limitations of two operand type data flow processors. This chapter establishes performance objectives for the proposed architecture by examining selected program constructs.

Chapter three proposes an innovative data flow architecture which distributes the instruction ready detection mechanism and accommodates a variable length, multi-operand multi-destination instruction format. Chapter four introduces a simulation model and tabulates the results obtained. In chapter five the results are compared against the objectives established in chapter two.

Finally, chapter six summarizes the work, discusses improvements and extensions to the modelled architecture and suggests areas for further investigation.

Chapter Two

BACKGROUND

2.1 Introduction

Studies of concurrency attainable within computer systems and of the representation of parallelism in programming languages have yielded a form of program representation and a supporting architecture known as a data driven or a data flow system. Data flow architectures offer a possible solution to the problem of efficiently exploiting concurrency of computation.

The concept of data driven computation is old, perhaps traced originally to the work of Seeber and Lindquist [Seeber63]. More recent interests have evolved from the data flow program graphs developed by Rodriquez [Rodriquez69] and Adams [Adams68].

The implementation strategies for data flow processors are as varied as the number of researchers involved. Many data flow models have been proposed [Dennis75, Arvind77, Arvind79, Arvind80a, Aarnio80, Rumbaugh75, Watson79] and some implementations described [Watson82, Arvind83, Takahashi83]. However, all the noted data flow models subscribe to program execution governed by the basic data flow principles.

The following sections introduce the reader to the data flow program graph and the notion of concurrent activity in the execution of the graph. Next some approaches to extracting the maximum concurrent activity from the graph and the limitations resulting from instruction format imposed restrictions are detailed. Several typical program constructs are analyzed and the maximum horizontal parallelism available in these algorithms extracted using an ideal data flow computer simulator.

2.2 Data flow Architectures

2.2.1 Data Flow Program Graphs

The data flow concept is a fundamentally different way of looking at instruction execution at the machine level. Execution proceeds according to the principles of data flow [Dennis80a, Davis82, McGraw80, Treleaven79]:

- a. a data flow operation may be executed if and only if all the required operands are available.
- (b. Operands are consumed to produce results. Thus, the input data no longer exists after the operation has been executed.
- c. The execution of a data flow instruction is functional and does not produce side effects.
- ~ d. Data dependencies between instructions define the only sequencing constraints, otherwise no control is imposed upon the execution sequence.

By adopting these principles, the asynchronous execution of programs can be achieved by data flow hardware organizations without the need for an explicit statement of concurrency in the programming language.

Data flow computers are quite different from conventional systems as they abandon the program counter and substitute data driven expression evaluation in its place. A data flow program can be described in a high-level applicative language such as VAL [Ackerman79] or ID [Arvind79]. A data flow program written in such a language can be directly translated into a directed graph (data flow program graph).

FIGURE 2.1
DATA FLOW OPERATOR NODE

FIGURE 2.2

GENERALIZED DATA FLOW OPERATOR

The data flow program graph is a two-dimensional representation of the data dependencies in the algorithm. It is composed of operator nodes (figure 2.1) representing some function and directed links connecting the nodes. The links define the data dependencies existing between the nodes and the conceptual paths over which the data items will flow. Data flows from the output of one node to the input of another in the form of data carrying entities called tokens.

Data flow operators consist of arithmetic, logical, control, identity, and structure operators. These operators are considered functional in the sense that they apply a function to the input operands to produce the results. These operators are a representation of the primitive capabilities (ie. a form of the machine code) for the architecture.

Any type of mathematical operations can be performed. As shown in figure 2.4, the nodes are often labelled by the primitive instructions they perform or alternatively, they can specify higher level functions

of a program (eg. square root) which can be executed in an asynchronous fashion. Multiple connections of nodes are used to generate a program graph (figure 2.4).

The generalized operator (figure 2.2) can be decomposed into a set of primitive operators suitable for implementation on a data flow processor. For most implementations, decomposition involves restricting the number of arguments to two and allowing a maximum of two output links.

Referring to figure 2.3, the flow of data can be viewed as the flow of tokens along the directed arcs of the graph from node to node. The action of consuming the input data, executing the operation and producing the result tokens is known as firing of the node. To be enable for firing, all of the required input tokens must be present on the input arcs. The firing of each enabled node within a graph proceeds independently of and concurrently with other nodes. In general, only one token is permitted on any arc. This restriction may limit the amount of possible concurrent activity. When the links are treated as FIFO queues or some other

identification mechanism is used, this restriction can be lifted; otherwise, some form of synchronization signals must be implemented. [Dennis80a, Oldehoeft77, Weng79].

(a) before

(b) after

FIGURE 2.3
NODE FIRING RULES

FIGURE 2.4

DATA FLOW PROGRAM GRAPH

Figure 2.4 shows a data flow program which calculates the roots X_1 and X_2 of a quadratic equation from the input values A, B , and C . Values are copied to more than one destination as required by the COPY operator (denoted by a circle on the graph). This program representation is aligned with the base level (assembly) language interpreted by the hardware.

Each primitive operator maps into a machine level instruction which can be executed asynchronously from all others provided the input data for that operation is available. Nodes such as $(4 * A * C)$ represent high level program functions and are replaced by equivalent primitive operators in figure 2.5. Parenthesis are used to indicate constants included in the instruction template. The value inside the square brackets indicates the origin of the operand that will be placed in that slot.

Program execution begins when data tokens arrive at A, B , and C . Data propagates through the graph as results become available.

Other node primitives can be introduced to conditionally control the routing of data tokens and implement high level language control constructs. Discussion of these operators is reserved to section 2.3.1.

The data flow graph can also be represented as a series of activity templates [Dennis 80a] where each template represents a node or a subprogram part. An activity template is created for each instruction to be executed. The general format of a template includes a field for the specification of the operation to be performed, input or receiver fields that accept incoming data tokens, and output or destination fields that specify the destination(s) of the result. Each output field of a template corresponds to the input field of another template. In this manner, templates are interconnected to produce the program structure. The quadratic root evaluation program is repeated in this form in figure 2.6.

INSTRUCTION NUMBER	OPERATION	INPUT PATH		OUTPUT PATH	
		1	2	1	2
1	COPY	A		6.2	13.2
2	COPY	B		4.1	3.1
3	EXPONENT	[2]	(2)	7.1	
4	NEGATE	[3]		9.1	
5	MULTIPLY	(4)	[6]	7.2	
6	MULTIPLY	C	[1]	5.2	
7	SUBTRACT	[3]	[5]	8.1	
8	SQRT	[7]		10.1	
9	COPY	[4]		11.1	12.1
10	COPY	[8]		11.2	12.2
11	ADD	[9]	[10]	15.1	
12	SUBTRACT	[9]	[10]	16.1	
13	MULTIPLY	(2)	[1]	14.1	
14	COPY	[13]		15.2	16.2
15	DIVIDE	[11]	[14]	X1	
16	DIVIDE	[12]	[14]	X2	

FIGURE 2.5
INSTRUCTION FORM OF A DATA FLOW GRAPH

FIGURE 2.6

ACTIVITY TEMPLATE FORM OF A DATA FLOW PROGRAM

2.2.2 Data Flow Computer Organization

A circular pipeline is the key component to many data flow architectures. The basic data flow computer organization (figure 2.7) consists of autonomous functional units for controlling the execution and processing the results. The basic structure includes: a token match store to determine those instructions enabled for execution; a unit to complete and fetch the data flow instructions from its program graph machine representation; and, a processing unit capable of concurrently executing many instructions. These units are connected by FIFO buffers which communicate the instructions, input data and results between elements of the processor.

A data flow program describing the computation is held in the instruction store as a collection of activity templates. The token matching unit receives result packets from the processing elements and collects the necessary number of operands required to enable an instruction. When the instruction is ready for execution (ie, all the operands are available)

the token match unit causes the instruction store to release the completed activity template to an available processing element. The processing section performs the operation specified by the operation code in the template and produces a result packet.

During execution of the program, a continuous stream of packets will flow around the ring. The basic mechanism will exploit the concurrency inherent in the program representation by selecting activity templates whose operands are available and initiating processing of the results. The speed of this organization will be limited by the bandwidth of the circular pipeline and the processing speeds of the functional units.

FIGURE 2.7

BLOCK DIAGRAM OF A DATA FLOW COMPUTER

2.3 Maximizing Concurrent Computation

2.3.1 Introduction

Although iterative and conditional control constructs are not integral to the data flow representation of many DSP and similar control algorithms, this section is included to introduce the notion of static and dynamic data flow computer organizations.

A loop or iterative construct of the form (while $P(x)$ do $F(x)$) can be graphed using the additional primitive operators of switch and merge (figure 2.8). The switch operator takes the input token and routes it to one of two outputs depending on the value of a boolean signal applied to the control input. The determinate merge operator selects from an available input token depending on the value of the applied signal. These operators are incorporated into the fundamental looping structure shown in figure 2.9.

(a) Switch Operator

(b) Determinate Merge Operator

FIGURE 2.8

NODE FIRING RULES FOR SWITCH AND MERGE OPERATORS

FIGURE 2.9
FUNDAMENTAL PROGRAM LOOP CONSTRUCTS

FIGURE 2.10
PROGRAM GRAPH FOR AN ITERATIVE SOLUTION

The iteration construct is represented by a cyclic data flow graph bounded by these special operators. A sequence of iterations is initialized by defining initial values for the loop variables which are placed on the input links. The relational operator code is responsible for testing for loop termination and controlling the generation of the new values which will be fed back to the beginning of the construct. Figure 2.10 shows a construct adapted from Boyer [Boyer84] to determine the square root of a number by Newton's method. The execution sequence for this graph is as follows:

a. Initial conditions: a token with value $x(0)$ arrives at instruction 1 and a true token exists at the control input.

b. Operator 1 fires sending tokens to 2, 3, 5, and 6.

c. Nodes 2, 3, 4, 5, 6, 7, and 8 fire as the operators become available. The logical operator 8 sends a boolean signal (true, false) to the control inputs of 9 and 1. If the boolean value is false (ie. the value of x has not converged sufficiently) the

switch operator (9) sends its value along the false path to merge operator (1). This operator in turn selects the false input to generate the next loop cycle.

d. When the convergence test evaluates to true, the switch operator outputs the result along the true path. A final true token will exist on the control input of operator (1) primed for the next root of x to be computed.

FIGURE 2.11
RECURSIVE COMPUTATION IN A PROGRAM GRAPH

Recursive programming techniques are useful for the solution of a large class of scientific problems. The recursive factorial algorithm (figure 2.11) due to Watson [Gurd80a] introduces the rectangular node symbol as a call to a named function - in this case the function itself. According to Watson and Gurd [Gurd80a, Gurd80b, Watson79], there are two approaches to solving this problem. The first technique is to copy the functional flow graph code each time tokens are directed to it. This technique requires runtime copying of graphs consuming both time and space. The alternative is to reuse a single copy of the code in such a way as to uniquely identify all ingoing tokens and route the results appropriately. Tokens that logically belong together can be labelled and the flow graph execution rules modified to permit input tokens with the same label to cause a node to fire. Since these tokens may need to be selected in an indeterminate sequence, strict queuing of tokens on arcs must not be used.

2.3.2 Instruction Processing in a Static Machine

A static data flow processor implementation can not support token labelling. Since subroutine calls are not supported directly, the compiler must replicate common code throughout the program. Program iterations can be expanded by the compiler when the iteration count is known. For example, a high level construct such as:

```
DO 100 J=1 TO 5
100  Z(J)=Y(J)*X(J)
```

can be expanded into elemental operators in the graph. When the compiler can expand these constructs at compile time, the expressions are said to exhibit static parallelism. The concurrency of these expressions is fixed and therefore static. However, when the loop count is not known until run time (ie. DO 100 J=1 TO N) the degree of concurrency can not be determined by the compiler and the expression is said to exhibit dynamic parallelism. This concurrency can be extracted at run time if the construct is programmed

using the merge and switch operators. Note, the degree of parallelism extracted by the hardware, will in general be limited to one unless special provision is made to "unravel" the loops.

The static machine implementation is suitable for many kinds of the data intensive calculations encountered in digital signal processing and control problems.

2.3.3 Instruction Processing in a Dynamic Machine

The architecture developed by Dennis [Dennis75] relies on finite buffering along the links to provide queues for the tokens. This architecture is a queued interpreter model [Arvind77] and is the basic structure followed by Rumbaugh [Rumbaugh75], Misunas [Misunas75], and Weng [Weng79]. A different approach taken by Arvind and Gostelow [Arvind79] defines a model commonly called the U-Interpreter

Arvind and Gostelow [Arvind77, Arvind78, Arvind80a, Arvind81a] recognized that iterative constructs in high level languages could be "unfolded" dynamically by the hardware in such a way as to maximize the amount of concurrent activity represented in a program graph. The unfolding-interpreter (U-Interpreter) uncovers parallelism in programs during execution by uniquely labelling independent activities as they are generated. The architecture proposed by Arvind, Kathail, and Pingail is illustrated in figure 2.12 and described in detail in [Arvind80a].

Each token contains a unique identifying label. Tokens (data value and label) are routed to processing elements on the basis of a mapping algorithm performed on the this label. Each processing element must perform an associative match in the waiting matching section and when both operands are available, an instruction is retrieved from the program memory. Data flow programs must be allocated to processing elements (PE) so that the instruction corresponding to an enable activity is available in the local memory. This must be achieved by loading a block of code on a group of PEs prior to the

execution of the code. For large problems, groups of PEs (physical domains) must be associated to minimize communication costs, and program loops and procedures must be restricted to logical domains and mapped onto physical domains. These activities must be performed in a semi-centralized fashion with minimal time overhead. Arvind recognizes some limitations with this scheme [Arvind80b] and states that a more flexible mapping mechanism is under development.

FIGURE 2.12
THE U-INTERPRETER

2.3.4 Static and Dynamic Performance Comparisons

Boyer investigated the performance of both static and dynamic architectures by simulation [Boyer84]. The static data flow processor is modelled after the Manchester cellular pipelined machine (see section 2.4).

The dynamic architecture supports loops and procedure calls. This necessitates a more complex controlling scheme although the fundamental component of this machine can still be based on a circular pipeline.

The following tables (TABLE 2.3.1, TABLE 2.3.2, TABLE 2.3.3, and TABLE 2.3.4) summarize some of the data collected when these architectures were exercised by a linear operator, matrix multiplication, and Kalman filter program. These programs are used as stimuli in this work and are described in detail beginning in section 2.5.4. Note in the static architecture the significant number of COPY operators required. As discussed in section 2.5.1, this requirement is

mandated by the restrictive two operand instruction format. As well, note that over 60% of the operations performed in the dynamic architecture are for label management related activities.

	Linear		Matrix		Kalman	
	Freq	%	Freq	%	Freq	%
Copies	32	33.3	200	47.1	1884	45.7
Adds	32	33.3	100	23.6	937	23.2
Mults	32	33.3	125	29.4	1203	29.2
Others	0		0		78	1.9

TABLE 2.3.1

STATIC COMPUTER OPERATION

Linear Operator Matrix Multiply

Instructions	40	127
Instructions and subroutines	252	127
Operation Packets	2480	4800
Result Packets	3228	6799

TABLE 2.3.2

DYNAMICS OF THE U-INTERPRETER SYSTEM

	Linear Freq	Operator %	Matrix Freq	Multiply %
Merge	573	23.1	1122	23.4
Switch	573	23.1	1122	23.4
D	420	16.9	935	19.5
L	153	6.2	198	3.9
Other Ctl	251	10.0	484	10.2
Math Ops	510	20.6	949	19.8

TABLE 2.3.3

EXECUTION FREQUENCY OF OPERATIONS
FOR A DYNAMIC MACHINE

MATRIX MULTIPLY PROGRAM

Static Machine

Execution time	280 usec
Proc Utilization	19 %

Dynamic Machine

Execution time	2700 usec
Proc Utilization	47 %

TABLE 2.3.4

COMPARATIVE STATIC-DYNAMIC PERFORMANCE

FIGURE 2.13
THE MANCHESTER DATA FLOW COMPUTER

2.4 A Practical Data Flow Processor

This section is adapted from [Watson82] and describes the University of Manchester circular pipelined data flow processor. This material is included to provide the reader with an understanding of the operation of a practical data flow processor and provide the foundation for a discussion of some performance restrictions in this architecture in section 2.5.

The architecture described below follows the same broad data flow principles pioneered by Dennis [Dennis75]. Watson and Gurd [Gurd80a, Watson79, Silva81, and Watson82] however have devised a token labelling scheme to implement re-entrant code structures. This facility permits conditional and iterative program structures to be implemented without the extensive hardware support required in the Dennis machine.

Figure 2.13 shows the data flow structure developed at the University of Manchester [Watson79]. A token package is the fundamental unit of information. The token travels a circular pipeline asynchronously between the functional units described below:

a. Instruction Store

The instruction store is a random access memory that holds the base language description of the data flow program graph. Each node of the program graph is represented by a node description which contains an operation code and one or two addresses of nodes in the instruction store which are to receive the result tokens. Provision is made to include a constant in one of the address fields. Executable instructions are formed by the instruction store by assembling data values passed on by the matching unit with the node description into a packet which is forwarded to the processing subsystem.

b. Processing Subsystem

The processing unit accepts executable instructions from the instruction store. It routes the instruction to an available processing element, performs the desired operation on the supplied data values, and routes result packets forward to the ring switch. Token labels can be manipulated by the processing elements.

c. Ring Switch

The ring switch is the interface to the host computer. Because data flow computers are not capable of non-deterministic input and output operations [Rumbaugh75, Weng79] or of program generation and compiling, a host computer is required. The host computer performs initial program loading into the data flow machine, injects initial data values into the loop, and collects results.

d. Token Queue

The token queue is a first in first out (FIFO) buffer which equalizes data rates around the system and provides storage for the potentially large numbers of tokens that can be produced. Watson reports that the present implementation uses a microprocessor controlled circular buffer in RAM with a capacity for 16K tokens.

e. Matching Unit

Token matching is a critical part of the machine operation. A result token consisting of a computed result, a label, and the destination node description address is accepted by the matching unit. An associative match is attempted on the contents of the matching unit using the concatenation of the label and the destination as the search key. The binary matching nature of the CAM is used to identify executable tokens. If a match is not found no partner exists as yet and the result token is stored in the matching unit. If a match is found, the token and its match is removed

to form a token pair ("group package") which is forwarded to the instruction store. The data values are used to complete an executable instruction when combined with the node description stored in the instruction store.

The matching unit is associative in nature. A true content addressable memory with the capability to store 16K of result tokens would be prohibitively expensive. Instead a hardware hashing technique is used to index into a static RAM memory [Silva83].

A mechanism is included to permit tokens to bypass the matching unit. This mechanism is used for nodes which require only one input token and those instructions which will be supplied with a constant from the node store.

Conditional, iterative and recursive structures are implemented by the hardware using the labelling approach described by Arvind [Arvind77b].

2.5 Performance Constraints

2.5.1 Introduction

First this ~~section~~ presents the practical considerations concerning the use of content addressable memory in data flow processors. As noted above, the use of this hardware imposes instruction format restrictions which lead to performance penalties. The extent of these penalties are investigated for some program constructs. The execution profiles for these algorithms is presented when the instruction format restrictions are removed. These ideal performance profiles are used as objectives for the architecture evaluated in chapter three.

2.5.2 A Note on Content addressable Memory (CAM)

A content addressable memory (CAM) or associative memory [Eimbinder71] is a storage medium that can be accessed by searching for data content. A CAM element is addressed by associating input data or "keys" with some or all of the stored words. Because of its organization, the CAM is uniquely suited to do parallel

searches by association. The amount of logic required to provide this function precludes making a small memory cell and hence the packing density of CAM memory cells is not nearly equivalent to that of standard MOS or Bipolar TTL RAM cells. For instance, a typical Bipolar IC contains only 16 bits of content addressable memory! In 1977, Carlberg [Carlberg77] investigated the feasibility of constructing a large CAM containing 65K 48-bit cells. He concluded that the cost might exceed 2.3 million dollars. Foster reports on the results obtained using a 320 word by 256 byte CAM [Foster82].

As noted the binary match or no-match search outcome of the CAM has been used [Watson82, Arvind80a] to implement the instruction ready detection mechanism. This binary operation has led to an almost universal use of a two operand format primitive instruction. Multiple operand instruction formats can not be implemented without sacrificing the speed associated with the CAM mechanism.

A true hardware CAM implementation is desirable for high speed operation. Nevertheless, the practicalities of implementing a large CAM have dictated that pseudo-associative storage mechanisms be used [Carlberg77, Gurd80b, Watson82]. Silva describes the implementation used in the Manchester computer [Silva83].

Others have minimized the requirement to use associative memory and implemented the instruction detection mechanism using a RAM memory based technique. These approaches [Caluwaerta82, Aarnio80, Takahashi83, Cornish79] require that fields in the instruction template be maintained using some type of intelligent processor based memory controller. Most implementors attempt to localize the memory accesses but in some instances global memory must be utilized. No assessment of the performance limitations or the memory contention that can be expected is reported.

2.5.3 Imposed Sequentiality.

The restriction of two inputs for an operator input can force sequentiality into a program graph.

Consider for example forming the sum of a series:

$$Y = X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8$$

The static data flow graph of this summation generates a binary tree with a depth of three as shown in figure 2.14. Although all the operands $X1 - X8$ are assumed available, the lower portions of the tree do not become enabled until the intermediate data values are released from the higher levels. In general, a binary tree of depth $\text{LOG}_2 N$ will be required to compute the desired results. Further, the maximum horizontal parallelism of eight is not extracted. Only $N/2$ processors are required for maximum performance and these processors can not be fully utilized in the remaining stages of the computation. the minimum execution time of the binary tree is of $O(\text{LOG}_2 N)$.

FIGURE 2.14
BINARY TREE FOR A SERIES SUMMATION

Matrix multiplication is characterized by a large number of inner product multiplications followed by addition of the inner product partial results. A large degree of parallelism is possible in the matrix multiply as all of the inner product multiplies can be performed concurrently. Consider the matrix equation $C = A \times B$ where A, B , and C are $N \times N$ square matrices. Two aspects of the matrix multiplication deserve consideration:

1. Propagation of Matrix elements. Each element of the matrix A is required in N product operations. When the number of destination addresses that can be included in an instruction template is restricted, the element must be distributed using the copy operator. A binary tree can be formed to replicate the required number of copies. In general, a tree of $\lceil \log_2 N \rceil$ is required (where the symbol $\lceil$ is used to denote the next highest integer number).

2. Summation of the inner product pairs. Each element of the C matrix is formed by summing the N inner product pairs. Again, a binary tree must be fashioned when the number of operands is restricted to two.

The number of operations required in a sequential machine is N^3 multiplies and N^3 additions for the general $N \times N$ matrix multiplication. The overall time complexity is according to Bentley of the order of N^3 [Bentley79].

The data flow implementation on a restricted instruction format machine requires:

1. a binary tree of copy operators to a depth of $\lceil \log_2 N \rceil$ which executes in $\lceil \log_2 N \rceil$ steps to distribute the operands
2. a binary tree of add operators to a depth of $\lceil \log_2 N \rceil$ which executes in $\lceil \log_2 N \rceil$ steps to form the terms of the output matrix
3. N^3 multiplies which can execute concurrently.

The sequentiality imposed by steps 1. and 2. above can be reduced. Consider a machine with the capability to accommodate multiple input operands and multiple destination labels. Then, if we assume the input operands are brought into the processor by an input operator of this format, the propagation of operands requires one execution cycle. Similarly, one execution cycle is required for the addition. The processor can concurrently execute the N^3 multiplications followed by the N^2 concurrent additions.

FIGURE 2.15
SIGNAL PROCESSING APPLICATION
FOR A DATA FLOW PROCESSOR

2.5.4 Analysis of Typical Program Constructs

2.5.4.1 Introduction

This section details several selected program constructs considered typical of DSP and control applications.

Digital signal processing and control applications are characterized by data intensive calculations. Samples from a system are taken and input to the processing system where some calculations are performed and outputs produced. This system topology is ideally suited to a data flow processor (figure 2.15) where the "waves" of samples arrive, are injected into the program graph, computation is performed and the results are produced. As there are no conditional constructs, and the dimension of the problem is known, a static implementation of the program graph is suitable. As subroutine calls are not supported in this type of architecture, code segments are replicated, or

alternatively, the granularity of the machine instructions can be elevated to include functional operators. No structure mechanisms are required nor are array data sequencing mechanisms.

2.5.4.2 Linear Operators

Linear operators of the form $Y=AX+B$ where A, B, X and Y are matrices are common. The distribution of input elements can be constrained by fan-out constraints. In multi-operand instruction formats, the elements of matrix B can be included in the ADD operator that is performing the inner product summation for each element of C . This eliminates another sequential execution step in the program graph.

2.5.4.3 Matrix Operations

As noted in section 2.5.3, data flow program graphs for matrix calculations have a great deal of horizontal parallelism. This concurrency can be extracted and significant performance gains made.

2.5.4.4 Kalman Filter

The Kalman filter is a linear, mean square, error estimator which can be used in control applications to track moving targets. It processes a sequence of noisy observations to provide recursive estimates to minimize the mean square of the state estimates. The theory of the Kalman filter is discussed in [Geib74, Jazwinski70] and a data flow implementation described in detail by [Farooq84]. Figure 2.16 shows the filter in its data flow form. The circles represent vector operations, and the rectangles represent functions. Each of the circles and rectangles can be expanded to its own data flow graph. The links are drawn so as to preserve the order of matrix multiplication and to show the correct orientation of tokens entering into the operators. For each invocation of the graph, the values of A and H are constant. The inputs $P(k/k)$ and $X(k/k)$ are generated by the previous iterations. New values of range, elevation and bearing are used in each successive invocation of the loop. When the next iteration of the filter is invoked, the instruction nodes 1, 1A, 2, 3, 4, 4A, 6, 6A, 6B, 6C, 7, 8, and 13 can be computed

activities possible in a rather complex series of equations. Note that on the upper portion of the graph a large amount of horizontal parallelism is evident tapering to some sequential activity near the bottom. The implementation described in the next section eliminates the copy nodes (1, 1A, 4A, 6, 6A, 6B, and 6C) required in a two input operand format implementation.

FIGURE 2.16

KALMAN FILTER DATA FLOW GRAPH

2.5.5 An Ideal Static Simulator

2.5.5.1 Overview

An analysis of expected performance gains similar to those presented in section 2.5.3 can be performed for problems with a simple regular structure. To document the effects of restricted instruction formats on complex problems (like the Kalman filter) an idealized data flow processor simulator was constructed [Smith85a]. These results are used as goals to judge the performance of the architecture detailed in chapter three. The simulator ignores the effects of communication delays and extracts the maximum horizontal parallelism available at each step of the program execution.

2.5.5.2 Mechanism

The simulator program operates on the instruction template representation form of the data flow program graph. The simulator consumes the data flow program in a series of "execution cycles" which consist of an alternating sequence of:

1. searching through all instruction templates to find instructions which can be fired. The simulator applies the criteria that the operands are present and the output arcs are not blocked.
2. performing the instruction node operation and propagating the results to the destinations.

As implemented, the simulator can accommodate instruction formats with up to eight input operands and eight destination labels. Operands and results are brought into a program and results obtained from the simulator by including the data flow instructions IN and OUT in the program graph. The IN instruction causes an execution unit to seek a named input quantity. The OUT operator causes an execution unit to output a named result.

Associated with the simulator and also described in [Smith85a] is a rudimentary data flow program graph generator. This generator was used to compile the programs detailed in the next section. The linear operator and 5x5 matrix multiply programs are used to stimulate both the simulator and the simulated

architecture described in chapter three. Program graphs are initially generated in an unconstrained fan-in and fan-out form. The generator package includes utilities to re-map the unconstrained form into an instruction template form of the graph with a given number of input operands and/or destinations.

2.5.5.3 Simulation Results

a. Linear Operator. Program graphs for two versions of a linear operator of the form $Y=AX+B$ were created.

The programs differ in the dimensions chosen for the matrices as well as the maximum fan-in (operands) and fan-out (destinations) permitted. TABLE 2.5.1 names the programs and TABLE 2.5.2 gives the dimensions of the matrices used.

PROGRAM NAME	INSTRUCTION TEMPLATE FORMAT	
LOB22	2 operands	2 destinations
LOB44	4 operands	4 destinations
LOS22	2 operands	2 destinations
LOS66	6 operands	6 destinations

TABLE 2.5.1

LINEAR OPERATOR PROGRAM NAMES

PROGRAM NAME	DIMENSIONS			
	A	B	X	Y
LOB22	4x2	4x4	2x4	4x4
LOB44	4x2	4x4	2x4	4x4
LOS22	6x3	6x6	3x6	6x6
LOS66	6x3	6x6	3x6	6x6

TABLE 2.5.2
LINEAR OPERATOR PROGRAM FORMATS

All linear operator programs were executed on the ideal static simulator. The execution results are summarized in TABLE 2.5.3. Note that neither of the multi-operand program formats were optimized to permit elements of B to sum along with the inner product summation. Consequently, the execution time includes one additional cycle for this operation. Further, two cycles are included for the IN operations which input the elements of the matrices into the graph and the OUT operators which exit the elements of C. Hence, five cycles is the minimum (1 - IN, 1 - Multiply, 1 - Inner product add, 1 - add elements of B, 1 - OUT).

In addition; note that significant amounts of storage and execution time are required for the COPY operators.

	LOB22	LOB44	LOS22	LOS66
Storage				
(templates)	144	112	468	288
(slots)	576	496	1800	1332
Copy Operators	32 (22%)	0	144 (30%)	0
Execution				
(Cycles)	7	5	10	5
Max Parallelism				
(after IN)	28	32	76	108
Major Arithmetic				
Adds	32	32	108	72
Mults	32	32	108	108
% Total	44%	57%	46%	62%

TABLE 2.5.3

SIMULATED RESULTS - LINEAR OPERATOR (IDEAL)

b. Matrix Multiplication. The program graphs for a matrix multiplication operation of the form $Y=AB$ where A,B, and Y are all 8x8 and then all 5x5 square matrices. was compiled. The program names and formats are given in TABLE 2.5.4. Two versions of each 5x5 matrix multiply program were compiled. The version indicated as random has the instruction templates arranged so that all IN operations are injected first and COPY operators are inserted into the program file as close to the generation of their input operands as feasible. The simulated results are tabulated in TABLE 2.5.5 and TABLE 2.5.6. Note from the table that the number of machine execution cycles required when the instruction format is unconstrained is significantly reduced over that required for the 2 input operand and 2 destination format.

TABLE 2.5.7 lists several additional matrix multiplication programs. These programs multiply square matrices of the size denoted by the first two numbers of the program name. The remaining two numbers denote the maximum size of the input and destination instruction fields respectively. These results show

that the execution time required for these types of matrix operations can be constant providing the instruction template format can contain a number of operands and destinations equal or exceeding the dimension of the matrix. This result is for the ideal case where there is an adequate number of processors and the communication costs are neglected.

PROGRAM	INSTRUCTION TEMPLATE FORM		RANDOM
M8822	2 operands	2 destinations	no
M8888	8 operands	8 destinations	no
MAT22	2 operands	2 destinations	no
MAT22R	2 operands	2 destinations	yes
MAT55	5 operands	5 destinations	no
MAT55R	5 operands	5 destinations	yes

TABLE 2.5.4

MATRIX MULTIPLY PROGRAM NAMES

MATRIX MULTIPLY (IDEAL)

	M8822	M8888
Storage		
(templates)	1920	768
(slots)	7040	3968
Copy Operators	768 (40%)	0
Execution		
(Cycles)	12	4
Max Parallelism		
(after IN)	298	512
Major Arithmetic		
Adds	448 (23%)	64 (8%)
Mults	512 (26%)	512 (66%)
% Total	49 %	75 %

TABLE 2.5.5
SIMULATED RESULTS - 8x8 MATRIX MULTIPLY

MATRIX MULTIPLY (IDEAL)

	M5522	M5555
Storage		
(templates)	450	225
(slots)	1700	1125
Copy Operators	150 (33%)	0
Execution		
(Cycles)	9	4
Max Parallelism		
(after IN)	94	125
Major Arithmetic		
Adds	100 (22%)	25 (11%)
Mults	125 (27%)	125 (55%)
% Total	50 %	66 %

TABLE 2.5.6

SIMULATED RESULTS - 5x5 MATRIX MULTIPLY

PROGRAM	EXECUTION CYCLES	MAX. PARALLELISM
		(After IN inst)
M33.22	6	25
M33.88	4	27
M44.22	7	55
M44.88	4	64
M55.22	9	94
M55.88	4	125
M66.22	10	150
M66.88	4	216
M77.22	11	217
M77.88	4	343
M88.22	12	298
M88.88	4	512

TABLE 2.5.7

SIMULATED RESULTS - MATRIX EXECUTION TIMES

c. Kalman Filter. Table 2.5.8 compares the program structure and execution results for two implementations of the Kalman filter. Program KAL22 has a constrained instruction format with two operands and two destination fields allowed. The detail of the model is described in [Boyer84, Farooq84, Smith85b]. When the instruction template format is not constrained significant performance improvements can be realized.

DATA FLOW KALMAN FILTER (IDEAL)

	KAL22	KAL88
Storage		
(templates)	4239	1831
(slots)	15187	8849
Copy Operators	1852 (43%)	163 (9%)
Execution		
(Cycles)	73	28
Max Parallelism		
(after IN)	211	251
Major Arithmetic		
Adds	955	244
Mults	1210	1202
% Total	51%	79%

TABLE 2.5.8

SIMULATED RESULTS - KALMAN FILTER

2.6 Chapter Summary

This chapter demonstrates the effect of restricting the number of input operands and the number of destinations that can be included in an instruction template. The amount of horizontal parallelism that can be extracted from an algorithm can be maximized when these restrictions are relaxed and a multi-operand and multi-destination instruction template is used. The number of instructions required to describe the program graph is reduced which results in a significant execution time improvement. These improvements are demonstrated using an ideal static simulator which shows that some programs can be executed in one third the time. Further, provided the COPY operators can be eliminated from the program graph, and that sufficient processors are available, the execution time required for some types of algorithms is not dependent on the dimension of the problem.

2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

Chapter Three

A PRACTICAL MEDIUM SCALE PROCESSOR

3.1 Introduction

In this section, a data flow computer for medium scale applications is described. The following are the key features of the architecture:

- a. A variable length instruction format is incorporated.
- b. The computer organization distributes the function of assembling and recognizing completed instructions.

c. The architecture is regularized providing the opportunity to adjust the number of processing elements to obtain a desired computational capability. The components are functionally modularized and are suitable for fabrication using VLSI technology.

3.2 Description of the Architecture

3.2.1 Overview

This machine structure is proposed to show the practicality of implementing a data flow processor that can have a variable format instruction template. The design avoids a CAM or memory controller based design by distributing the instruction formation and instruction ready detection mechanisms.

The data flow program graph is translated into a static instruction template representation by the compiler. These instruction templates are loaded into available storage in one of the template storage units

from which they are injected into one of the the slotted rings. (figure 3.4). A slotted ring communication mechanism transports instruction templates within the machine. Each processing element is connected to the slotted rings so that the contents of passing templates can be "observed". The processing elements can contribute to the completion of passing templates, using stored results from previous computations. Processing elements can execute completed templates. The results from these computations are saved internally and used as described above to complete passing instructions.

The templates storage units provide storage for large programs whose instruction templates can not all fit on the ring. They perform housekeeping functions by removing executed templates and injecting new instruction templates into the ring mechanisms.

The computer is slaved to a host for program loading and control. The host could provide input data and collect results but a more efficient operation in control and DSP applications results when the inter-domain converters are directly connected into the data flow processor.

This following section describes the Slotted Ring Machine (SRM). The machine consists of a slotted ring, a number of connected processing elements, and two template storage units arranged as shown in figure 3.1

FIGURE 3.1
SLOTTED RING ARCHITECTURE

3.2.2 The Slotted Ring

Hasegawa describes a multitrack bus (MTB) capable of concurrent data exchanging [Hasegawa81]. Hasegawa claims his architecture was adapted from a double track railway system and is configured as a pair of Unilateral bus lines (tubes) situated in opposite directions. The FIFO register like characteristics of the tube are desired in the MTB operation to transport data in quanta.

The slotted ring communication mechanism is similar in operation. In this implementation, a ring slot is considered as a 48 bit parallel word. These words are passed from connection to connection. Figure 3.2 shows a portion of the ring structure.

The data quantum chosen for communication purposes is the instruction template. Each template (figure 3.3) consists of a header, a variable number of operand fields, and a variable number of destination fields laid out as detailed in the following paragraphs:

FIGURE 3.2
RING SLOT MECHANISM.

A horizontal rectangular box representing the header slot of an instruction template.

HEADER SLOT

A horizontal rectangular box representing one of the operand slots in the instruction template.

0 to N
OPERAND SLOTS

}

A horizontal rectangular box representing another operand slot in the instruction template. A horizontal rectangular box representing one of the destination slots in the instruction template.

0 to N
DESTINATION SLOTS

}

A horizontal rectangular box representing another destination slot in the instruction template.

FIGURE 3.3
FORM OF AN INSTRUCTION TEMPLATE

Type	S	Label	Color	In-Count	Opcode	Age
------	---	-------	-------	----------	--------	-----

FIELD	SIZE	CODING	REMARK
Type	3	01	instruction template
Status	2	00 01	template in progress template executed
Label	16		instruction template label ID field
Color	7		for recursive invocations and multi-programming
In-Count	10		-(number of reqd operands)
Opcode	8		designates the operation to be performed
Age	2	00 01 10 11	moving R-L and untouched R-L and used moving L-R and untouched L-R and used

TABLE 3.2.1

HEADER SLOT CODING

a. Headers. Each instruction template consists of at least a header slot. The header designates the form of the template which follows on the ring. TABLE 3.2.1 details a bit allocation scheme.

Every slot on the ring contains a type field which designates the function of that particular ring slot. Only four of the eight types are defined and used in this machine (header, empty, operand, destination). This basic set is easily expanded to include system control and error status packages.

The 2-bit status field is used to indicate if the template has been executed by one of the processing elements that it had passed. This enables the templates stores to remove executed templates from the ring.

The label and color field are used to identify the instruction template package. This information is used by processing elements to determine if any of their previously calculated operands can be added to a passing template.

The opcode field designates the operation to be performed by a processing element. The age field is coded by the templates stores and processing elements to indicate stagnant templates. Stagnant templates are defined as templates which have circulated the ring structure without any type of action being performed on them. They are temporarily removed from the ring to permit other instructions to circulate. This simple mechanism prevents deadlock which can result when all slots of the ring are occupied by instruction templates which can not be completed by any of the results contained in the processing elements.

The in-count field contains the number of operands required to complete the instruction package. It is initially set to the required number and decremented each time an operand is added. A zero in this field indicates an instruction which is ready to execute.

Note that the contents of the header permit a processing element to determine whether or not it will interact with the passing instruction template. The label, color, opcode, and ih-count fields combine to permit the determination of whether the template is/ ready for execution, whether an operand can be added, whether the template is complete (or will be completed by the addition of operands), and whether the processor can execute the opcode specified.

b. Operand Slots. When the instruction package is created, the compiler allocates slots for the inclusion of the required number of operands. Each operand slot represents one of the input operand ports for the mathematical or other function or primitive instruction to be performed. The data flow graph designates the named operator and the port that the operand is to enter. These ports are numbered from one beginning with the first operand slot.

The 5-bit mod field in the operand slot (TABLE 3.2.2) is used to designate if the operand is resident, if it is a constant, or if the slot is empty. In addition, three bits are allocated to type the data. Typed data fields reduce the total number of arithmetic opcodes required and permit type conversions, when required, to be performed at runtime by the processing elements.

Forty bits are allocated to contain the data value. This is sufficient for a reasonable sized integer representation; as well, this field can accommodate a single precision floating point representation.

c. Destination slots. TABLE 3.2.3 shows the bit allocations for the destination slots. Multiple destination slots can be included in each instruction package. The single bit mod field indicates if either one or two destinations are included in the slot. Each destination contains the label of the node operator and the port that the result is to enter.

Two types of destination slot formats are proposed. When two destinations are included, the number of available bits restricts the port range to 32. This is sufficient for most applications, however when a large range of ports is required (for example to accommodate a high level function operator) then a single destination field is adopted with the additional bits used to expand the port range.

The compiler produces an instruction template package with the header and destination fields completed and slots allocated for the operands (figure 3.3). The size of the package is not constrained by the ring mechanism.

Type	Mod	Operand
------	-----	---------

FIELD	SIZE	CODING	REMARK
Type	3	010	operand slot
Mod	5	xxxx0 xxxx1 xxx0x xxx1x ---xx	not a constant constant in this slot operand not present operand present 3 bits used to code operand type (tag)
Operand	40		40 bit integer or floating point format

TABLE 3.2.2

OPERAND SLOT CODING

Type	Mod	Destination 1	Port	Destination 2	Port
------	-----	---------------	------	---------------	------

FIELD	SIZE	BIT	REMARK
Type	3	011	destination slot
Mod	3	xx0 xx1	one destination format two destination format
Dest	16		destination label ID
Port	6		destination port number

TABLE 3.2.3
(DESTINATION SLOT CODING

3.2.3 Processing Elements

The number of processing elements attached to the slotted ring mechanism determines the amount of work that can be performed by this data flow processor. The number of processing elements connected into the structure determines the size of the slotted ring. This number can be tailored to produce the required performance by trading off the amount of horizontal parallelism that the hardware extracts from the program graph against the cost of the processing elements and their connections to the ring.

Each processor can be considered a general purpose processor in the sense that it can execute every type of operation code. Applications of the SRM do not preclude the use of special purpose processing elements such as dedicated input or output devices and other inter-domain converters.

FIGURE 3.4
BLOCK DIAGRAM OF A PROCESSING ELEMENT

FIGURE 3.5

BLOCK DIAGRAM OF THE ARITHMETIC PROCESSING UNIT

FIGURE 3.6
BLOCK DIAGRAM OF THE INTERFACE PROCESSING UNIT

Each processor must be connected to at least one ring slot. Multiple connections into to the same ring structure or separate connections into a number of different rings are of course possible. Four connection strategies are considered in this work and outlined in chapter four.

Each processor (figure 3.4) is logically divided into two main sections: the slot interface unit (figure 3.6) and the arithmetic processing section (figure 3.5). These units function concurrently.

The arithmetic processing unit operates on the operands contained in the instruction templates. The combination of the operation code contained in the header and the type information coded in each operand slot specify the transformation to be performed. This scheme permits the granularity of the data flow program graph to be coded at any level ie.. primitive instructions and higher level functions can be intermixed. In addition, instructions can operate on many data types; for example, only one ADD instruction

is required. Data typing, and conversions when necessary are performed at the processing unit. Results are placed in result memory for use by the interface processing units.

The interface processing unit(s) operate concurrently with the processing unit. An interface processing unit is attached to each ring connection interface to scan the passing instruction templates. Information coded into the header slot of each instruction template makes it possible to determine if the template can be executed or if a pre-calculated result can be added. The processing flow diagram is shown in TABLE 3.2.4

TABLE 3.2.4
FLOW DIAGRAM FOR THE
INTERFACE UNIT OPERATION

Various ring connection strategies can be postulated. The ring interface can be considered to be two, separate ports which look toward adjacent neighbours. "Slots" are accepted, buffered, and processed asynchronously to both ports. This permits an interface unit to be constructed using economical components but restricts the functional speed of the slotted ring and can add considerable delay.

Alternatively, the ring connections can be considered to be observing one 48-bit register of a series of cascaded registers representing the ring structure. The contents of these registers can be "pumped" synchronously from slot interface to slot interface. This strategy demands that the interface processing unit can examine a slot, decide on a course of action, and modify the contents if necessary within allotted ring cycle time.

When a fast interface processing unit is required, the labels (and possible colors) of pre-calculated results must be stored in a local content addressable memory or a register block augmented with additional match detect circuitry. The CAM "match" bit is used to

index into a conventional RAM location where the result is stored. This mechanism permits a label association to determine if a result is available for contribution to the passing template and minimizes the amount of CAM required. If the match bit is asserted, the interface unit has at least one slot time in which to fetch the data value and prepare to insert it into the appropriate slot.

3.2.4 Template Storage Units

The template storage units are intelligent buffers inserted into the slotted ring mechanism. They provide additional instruction template storage, a mechanism for injecting instruction templates, and an instruction prioritizing and garbage collection mechanism. Token storage units can also participate in the management of instruction template labels (colors). For dedicated operation, the instruction template form of the program graph must be re-used for each new set of data. This is easily arranged by having executed instruction templates from the previous execution re-injected into the ring with a new color field. The color fields are managed by a globally accessible color register.

The logical composition of the template storage units is shown in figure 3.7. There must be at least one template storage unit. In dedicated operations, the template storage unit acts as the data flow computer controller. It initializes the system at startup and injects the initial instruction templates. In addition, it must handle all abnormal conditions. When the data flow processor is slaved to a host, the

template storage unit acts as an interface for control and program loading purposes. The number of template storage units or placement in the ring is not restricted.

The template storage units can be used to load the data flow program onto the ring. As the order of instruction execution is determined by the availability of the data operands, all units can participate in loading segments of the program.

(In normal operation, instruction templates pass from the input port to the output port without processing. To prevent a deadlock condition which results when none of the circulating instruction templates can be completed, the template storage units remove instructions templates which have circulated once through the entire ring into a FIFO holding buffer. This permits all templates of the graph to be circulated in front of the processing elements.

FIGURE 3.7
BLOCK DIAGRAM OF THE TEMPLATE STORAGE UNITS

Further, any instruction templates which are ready for execution and enter the input port are immediately moved to the output port. Any executed templates are normally removed from the ring.

The template storage unit is the logical place to include system error handling. The template storage unit can initiate system messages to ascertain status and collect operating statistics. It can handle system error messages received from the processing elements and communicate with the host processor.

3.2.5 Discussion

This architecture addresses the need to implement a medium scale data flow processor. The maximum performance that can be achieved will be limited by the bandwidth capabilities of the ring communication network. For a system with 50 processors, average PE execution times of 20 usec, and an average instruction template of 4 words, a ring operating speed of the order of 10 Mhz could sustain processing at the rate of approximately 3.2 million instruction templates per second. For higher performance, to some degree, additional ring structures can be added between the template storage units and terminated by all PEs.

No mention has been made of data arrays. As data flow is a side effect free language, data arrays are consumed when operated on to produce a new data array (even when one element is changed!). Various schemes have been proposed to eliminate the overhead involved and maintain the integrity of the data structure. The most promising involves defining I-Structures [Ackerman77, Arvind81b] and providing a global memory

accessible by all PEs and embedding pointers to parts of the I-Structures in the instruction templates. These structures are not considered in this work as this mechanism is not required for the types of DSP programs being considered. However, it seems feasible to incorporate the same type of structure controllers described by Arvind and a structure memory into this architecture.

In a similar manner, the architecture could be extended to support dynamic data flow programs. This involves primarily extending the instruction set to include the label management operators necessary and including a structure controller. These extensions are not considered in this work.

3.3 Chapter Summary

In this chapter, a data flow processor capable of supporting variable length instruction formats has been described. A variable length instruction does not require a fixed number of operands or destinations. Consequently, the instruction format can be varied by the compiler to suit the program graph.

The instruction ready detection mechanism is distributed thus facilitating concurrent instruction formatting and "ready" checking activities in the machine. This technique eliminates the requirement to have a CAM or to use the potentially performance restrictive memory controller approach.

In addition, the ring structure permits the number of PEs in the system to be varied to meet a given performance specification. Finally, the components of the computer are modularized to permit implementation using current VLSI technology.

In the next section, a simulation model of the processor is described.

Chapter Four

SIMULATED PERFORMANCE OF THE MODEL

4.1 Introduction

This chapter describes a discrete event driven simulation model used to evaluate the slotted ring architecture. Section 4.2 gives an overview of the modelling technique.

The simulation scenarios are outlined and the results tabulated in sections 4.3. These findings are used to compare with the ideal performance criteria in chapter five.

4.2 The Simulation Model

4.2.1 Overview

The slotted ring architecture was simulated using a discrete event driven simulation modelling approach. Events are defined which change the state of the modelled system; these events are created during the simulation and are chronologically ordered and executed by the simulator. Concurrent activity is modelled by executing several events without advancing the simulated time. The details of the model are described in [Smith85b].

4.2.2 SLAM Model

SLAM (Simulation Language for Alternative Modelling) was used to implement the models. This language [Pritsker79] provides facilities to permit modelling in several modes. In the discrete event driven mode, the language provides the basic housekeeping functions required to manage the chronological selection of the next event and accumulate statistics.

The form of the model, the definition of the events, the entities and the mechanism for processing the events and changing the values of the entities are described by the user. Subroutines (preferably written in FORTRAN) to effect each of the defined events are combined with the SLAM housekeeping and statistics collection routines to produce a FORTRAN program to simulate the system.

The simulation model described in detail elsewhere [Smith85b] contains processing for the major events of clocking the ring, servicing template slots as they pass processing elements, and servicing template slots as they enter the template storage units. The entities in the model represent control information used to manage the simulated instruction templates.

4.2.3 Control of the Simulation

A basic program model was developed so that it could be easily manipulated to represent the different topologies detailed in section 4.3. Values are input to the SLAM program from control files to affect the characteristics of individual simulations. Provision has been made to change:

- a. simulated execution time. The maximum length of the simulation time in terms of ring shift cycles can be specified.

- b. instruction execution time. The amount of simulated time in terms of ring shift cycles can be designated for each instruction type. This permits input and output instructions to be assigned execution times greater than numerical processing activities.

c. size of the result memory. Each processor has a fixed result memory which is used to store the calculated results. The size of this memory impacts on the cost of the processor (some of this storage is CAM or hardware registers). When the result memory is full, the processor can not do any additional work.

d. number of processing elements. The total number of processing elements connected into the ring affects the amount of total work that can be performed by the data flow computer. In addition, for all but the adaptive ring model, the total number of processors also affects the ring associated communication delays.

e. number of active program loaders. All models contain two templates storage units. Each template storage unit has the capability to inject instruction templates from the program graph into the machine for execution.

f. statistics collection. Several input controls can be exercised to affect the amount and the detail at which statistics are collected.

g. monitor flags. Many trace and monitor controls have been installed to permit the detailed tracing and monitoring of the machine operation. Using these flags, the experimenter can dump the contents of the ring, selected templates stores or processing elements; trace the execution of instruction templates as they propagate through the machine; and, monitor the progress of the simulator on long computer runs.

4.3 Modelling

4.3.1 Model Instrumentation

Using the facilities of SLAM, software instrumentation is included in the simulation models to observe the architecture and record any measured values.

The following paragraphs define the quantities that can be measured.

a. result waiting times. The length of simulated time that a result waits in a processing element until it is added to an instruction template.

b. time to complete an instruction. The time required for an instruction template to be completed on the ring. This time can be zero if the instruction is injected in a ready to execute form (eg. IN and OUT instructions).

c. time waiting execution. This is the time that a completed instruction must wait until execution is started. Most instructions have no wait time as they are completed at a processing element that is not busy and can begin execution immediately.

d. elapsed time to execution. This is the accumulated time from the initial injection of the instruction template until the time that execution is started.

e. program execution time. This is the elapsed time from the start of program injection until the last output results have been calculated.

f. result memory queue utilization. These figures are collected to show the amount of available result memory space that is consumed by the program. During times when all available space is in use, the processing elements can not do any additional work. In addition, statistics are collected for the maximum and minimum numbers of result entries that are made into the result memories.

g. ring resource utilization. Statistics are collected for the number of available ring slots that are occupied by ready, executed and not ready instruction templates.

h. template storage unit utilization. The amount of token storage buffers used is recorded. Template storage units contain three buffers. These buffers are used to store templates, admit templates from the program graph and permit ready instruction

templates to bypass as yet incomplete instruction templates. Statistics for the number of ready and not ready instruction templates in the template storage unit are maintained. As well, the number of ready instruction templates that are buffered temporarily (overflow) are counted.

i. instruction queue length. The number of program instructions that remain to be injected into the machine is counted.

4.3.2 Description of the Simulated Topologies

The following paragraphs describe the four simulation topologies that were used.

a. FULL RING MODEL. This computer architecture consists of two template storage units separated by two slotted ring mechanisms. Both rings rotate in the counter clockwise direction. A number of processing elements can be connected between opposite slots on the ring. Figure 4.1 shows the organization.

b. FULL RING MODEL WITH VARIABLE SIZE RESULT MEMORY. This model has the same topology as the full ring. However, the simulation runs used processing elements with result memory with the capacity for 16, 64 and 128 result tokens. The full ring model has a default value of 16.

c. SHORT RING MODEL. The short ring model is similar to the full ring model shown in figure 4.1 except that the left to right slotted ring is removed. This decreases the round trip transit delay for instruction templates.

d. TWO LOADER FULL RING MODEL. The topology of this model is identical to figure 4.1. In these simulation runs the program templates were injected by both template storage units. Consequently, processing elements at both ends of the slotted ring were able to begin execution on instruction templates immediately.

FIGURE 4.1

(figure 3.1 repeated for convenience)

SLOTTED RING MACHINE CONFIGURATION

4.4 Summary of Simulation Results

A 5x5 matrix multiplication program was run on all topologies, using between five and two hundred processing elements. The two loader topology consistently yielded the lowest execution times and was selected as the topology to exercise with the linear operator program. These programs were run varying the number of processing elements. The minimum execution time achieved in each case was selected for tabulation. The 8x8 matrix multiply and the Kalman filter program were not used due to the excessive amount of simulation time that would be required.

	LOB22	LOB44	LOS22	LOS66
Execution				
(Cycles)	1267	747	3686	1284
Processors				
used	25	25	50	100
Avg Proc				
Utilization	43 %	58 %	23 %	19 %
Avg/Max Result				
Wait Time	86/219	62/144	297/747	301/547
Max Time to				
make inst ready	1140	611	3440	1060

TABLE 4.4.1
SIMULATED RESULTS
LINEAR OPERATOR (SRM)

These program compute $Y=AX$ where A, X , and Y are five by five square matrices. The data flow graphs are compiled into four instruction template representations as given in TABLE 4.4.2. The random format files are organized to remove any compiler bias that might exist. The compiler reformats unconstrained program graphs into constrained forms by adding any additional instructions onto the end of the program file. In some cases, the SRM architecture can not perform any work until some of these instructions are injected into the ring. The random format files consist of all IN operators at the beginning followed by instruction templates selected at random from the compiler produced program file.

PROGRAM NAME	PROGRAM FORMAT	RANDOM
MAT55	5 operands 5 destinations	no
MAT55R	5 operands 5 destinations	yes
MAT22	2 operands 2 destinations	no
MAT22R	2 operands 2 destinations	yes

TABLE 4.4.2

MATRIX MULTIPLICATION PROGRAM FORMATS

The results of the simulations are summarized in the tables included in this section (TABLE 4.4.1 and TABLES 4.4.3 - 4.4.6). Graphs for the execution profiles, processor utilization, and ring utilization are include in appendix A. Full simulation results are in [Smith85c, Smith85d, Smith85e, Smith85f, Smith85g, Smith85h, Smith85i]

	MAT22	MAT55	MAT22R	MAT55R
FULL RING	4990	1362	4219	1583
FULL RING				
RM=64	4990	1359	-	-
FULL RING				
RM=128	4990	1359	-	-
SHORT RING	5735	1592	4622	1628
TWO LOADER	3659	1084	3904	1152

TABLE 4.4.3
SQUARE MATRIX MULTIPLY
EXECUTION TIMES

	MAT22	MAT55	MAT22R	MAT55R
FULL RING	429/1450	359/1000	420/1540	388/759
FULL RING				
RM=64	395/1450	298/759	-	-
FULL RING				
RM=128	395/1450	298/759	-	-
SHORT RING	617/1450	402/759	498/1540	390/1000
TWO' LOADER	286/746	211/502	280/864	209/559
				min/max

TABLE 4.4.4
SQUARE MATRIX MULTIPLY
RESULT WAIT TIMES

	MAT22	MAT55	MAT22R	MAT55R
FULL RING .	8.9%	12.6%	13.9%	27.6%
FULL RING				
RM=64	8.9	21.4	-	-
FULL RING				
RM=128	8.9	21.4	-	-
SHORT RING	15.3	13.8	9.5	26.8
TWO LOADER	9.7	39.6	18.6	37.9

TABLE 4.4.5
SQUARE MATRIX MULTIPLY
PROCESSOR UTILIZATION

	MAT22	MAT55	MAT22R	MAT55R
FULL RING	59.1%	29.4%	59.4%	35.6%
FULL RING RM=64	59.1	31.0	-	-
FULL RING RM=128	59.1	31.0	-	-
SHORT RING	76.7	53.1	75.1	56.9
TWO LOADER	65.7	56.0	69.1	54.0

TABLE 4.4.6
SQUARE MATRIX MULTIPLY
RING UTILIZATION

4.5 Chapter Summary

This chapter described the discrete, event driven simulator used to model the data flow architecture. The results of simulation runs using the linear operator and matrix multiply programs were tabulated. Chapter five relates the SRM performance characteristics with the ideal criteria established in chapter three.

Chapter Five

COMPARISONS AND DISCUSSION

5.1 Introduction

In the following sections, the machine code of the data flow program graph for a constrained instruction template representation is contrasted with the machine code representation that can be achieved when the instruction template format is not restricted. The reduction in the number of operators required combined with the removal of the sequentiality in the program structure results in a significant performance improvement which is detailed in section 5.3.

5.2 Program Comparisons

The tables included in this section compare the program size, number of COPY operators required and the number of additional program operators that are required because of the instruction template format.

The programs used for both the ideal and the SPM simulations are identical. Using a variable instruction template format results in a program which is more compact, has a smaller number of arithmetic operations to perform, and removes any imposed sequentiality from the execution profile of the program graph.

The number of templates required for both versions of the linear operator is reduced when the multiple format instruction templates are used (TABLE 5.2.1 and TABLE 5.2.2). These reductions are primarily due to the elimination of the COPY operators. In the LOB program, the COPY operators are required to distribute the elements of the matrix to the multiplications. The dimensions of the matrices involved permit the add operations to be performed without any instruction format imposed sequentiality using the two input

operand instruction. Even so, fourteen percent more horizontal parallelism is extracted. The LOS program requires ~~when~~ the distribution of matrix elements and the elimination of the sequentiality imposed by the two operand ~~ADD~~ instruction format. In the multi-operand version of this program the COPY operators are eliminated and there is a reduction in the number of ADD operations required. The hardware can extract approximately forty-two percent more concurrency from this program representation.

The five by five and eight by eight square matrix multiplication programs (TABLE 5.2.3 and TABLE 5.2.4) benefit in the same manner. The COPY operators are eliminated and the ADD operators reduced. Eliminating the ADD operator binary tree imposed sequentiality results in the extraction of seventy-one percent additional concurrency for the 8x8 matrix multiply program.

The Kalman filter (TABLE 5.3.5) requires input operands to be distributed to as many as 12 operators. The ideal simulator was implementation constrained to accommodate eight input operands and eight destinations. Not all COPY operators are eliminated from the unconstrained program representation. This results in only a marginal increase in the horizontal parallelism that can be extracted.

	LOB22	LOB44	RATIO
Templates			
Required	144	112	.77
Copy Operators	32 (22%)	0	
Max Horizontal			
Parallelism	28	32	1.14
Major Arithmetic			
Adds	32 (22%)	32 (28.5%)	
Mults	32 (22%)	32 (28.5%)	
% Total	44%	57.0%	1.29

TABLE 5.2.1
PROGRAM CHARACTERISTICS.
LINEAR OPERATOR VERSION 1

	LOS22	LOS66	RATIO
Templates			
Required	468	288	.615
Copy Operators	144 (30%)	0	
Max Horizontal			
Parallelism	76	108	1.420
Major Arithmetic			
Adds	108 (23%)	72 (25.0%)	
Mults	108 (23%)	108 (37.5%)	
% Total	46%	62.5%	1.360

TABLE 5.2.2

PROGRAM CHARACTERISTICS
 LINEAR OPERATOR VERSION 2

	M8822	M8888	RATIO
Templates			
Required	1920	768	.395
Copy Operators	768 (40%)	0	
Max Horizontal			
Parallelism	298	512	1.710
Major Arithmetic			
Adds	448 (23%)	64 (8%)	
Mults	512 (26%)	512 (66%)	
% Total	49%	75%	1.530

TABLE 5.2.3
PROGRAM CHARACTERISTICS
8x8 MATRIX MULTIPLICATION

	MAT22	MAT55	RATIO
Templates			
Required	450	225	.500
Copy Operators	150 (33%)	0	
Max Horizontal			
Parallelism	94	125	1.329
Major Arithmetic			
Adds	100 (22%)	25 (11%)	
Mults	125 (27%)	125 (55%)	
% Total	50%	66%	1.320

TABLE 5.2.4.

PROGRAM CHARACTERISTICS

5X5 MATRIX MULTIPLICATION

	KAL22	KAL88	RATIO
Templates			
Required	4239	1831	.431
Copy Operators	1852 (43%)	163 (9%)	
Max Horizontal			
Parallelism	211	251	1.189
Major Arithmetic			
Adds	955 (22%)	244 (13%)	
Mults	1210 (28%)	1202 (66%)	
% Total	51%	79%	1.540

TABLE 5.2.5
PROGRAM CHARACTERISTICS
KALMAN FILTER

5.3 Performance Comparisons

The following sections compare execution time performance obtained with the ideal simulator with that obtained from the two loader SRM architecture. In TABLE 5.3.1 the potential improvement is tabulated as the ratio (restricted format execution time/un-constrained format execution time).

The figures detailed in chapter two for the ideal simulator are in terms of the number of execution cycles required to traverse the program graph from entry to exit. The figures tabulated in chapter four are from the simulated SRM taking in to account the time required to perform the actual arithmetic activities and the communication times required for the intermediate results to propagate through the machine. If a machine cycle for the SRM is considered to be the time required for an instruction template to completely traverse the ring (a function of the number of processing elements connected) we can use this in calculating a figure of merit for comparing the performance of the SRM against the ideal. This figure

of merit is given in TABLE 5.3.2 and is calculated as the ratio (number machine cycles taken in the ideal simulation/the number of machine cycles required by the SRM).

	IDEAL	SRM
Linear		
Version 1	1.40	1.69
Linear		
Version 2	2.00	2.87
Matrix		
Multiply	2.25	2.69

TABLE 5.3.1
POTENTIAL PERFORMANCE IMPROVEMENT RATIO
IDEAL AND SRM MACHINES

	FORMAT	
	CONSTRAINED	UNCONSTRAINED
Linear		
Version 1	.2762	.3346
Linear		
Version 2	.2712	.7788
Matrix		
Multiply	.2899	.3472

TABLE 5.3.2
FIGURE OF MERIT
IDEAL AND SRM MACHINES

5.3.1 Linear Operator

TABLE 5.3.1 indicate that a performance improvement of 1.40 is attainable. A performance improvement ratio of 1.69 was achieved for version 1 and 2.87 for version 2. First, note that the ratio 1.40 is calculated with the ideal execution figures from TABLE 2.5.3. These figures include 1 cycle for input and an additional cycle for output. Thus the actual improvement ratio when only the arithmetic operations are concerned could be as high as 2.5. Secondly, the communications delays in the SRM affect longer input programs more than short programs. When these two This bias accounts for the seemingly un-attainable improvement ratio of 2.87.

TABLE 5.3.2 gives the calculated figures of merit for the SRM architecture. A figure of merit of 1. would indicate that the ideal performance had been obtained on the SRM.

Version 2 of the linear operator program results in a figure of merit of .77 when the variable format instruction template is executed on the SRM. This is to be expected as forty-two percent more concurrency was predicted.

5.3.2 Matrix Multiplications

TABLE 5.3.1 indicates that a performance improvement of 2.25 was predicated using the ideal simulator. A performance improvement of 2.69 was achieved when the execution times for the 5x5 matrix multiplication programs are compared. Again, the figures in TABLE 5.3.1 include the one input and one output cycle which tend to bias the estimate on the low side. When these cycles are arbitrarily eliminated, an improvement ratio of 3.5 is predicted.

5.4 Chapter Summary

This chapter has compared the predicted performance improvements obtained with the ideal simulator, with those achieved in the slotted ring machine. Performance improvements were realized when imposed sequentiality was removed by utilizing a variable instruction template format that could accommodate multi-operands and multiple destinations. A reduced execution time resulted due to the smaller

number of operations, to be performed and the fact that sequentiality was eliminated from sections of the program graph and more horizontal parallelism could be extracted from the program graph by the hardware.

Chapter Six

SUMMARY AND RECOMMENDATIONS FOR FURTHER WORK

6.1 Summary of Work Presented

The introduction to this work established as goals for contemporary high speed computer architectures, a computing machine which would be possible to program in a manner such that the performance potential could be exploited; which would achieve high performance at an acceptable cost; and, which would exploit the potential of LSI technology. The data flow architecture described in this work meets these challenges.

LSI technology has made small autonomous processors possible. Data flow architectures provide the mechanism for connecting many autonomous processors so that they can co-operate on a single task.

By its nature, programming a data flow architecture insulates the programmer from a requirement to have a detailed knowledge of the computer structure. The algorithm is not mapped onto the hardware in the sense of traditional multi-processor process allocation. Moreover, the programmer need only describe the data interactions for the hardware to recognize and extract the maximum execution concurrency possible. In chapter two it is demonstrated that when the instruction format is constrained by the hardware implementation, the maximum potential horizontal parallelism can not be extracted by the computer, resulting in a less than optimum execution time. These types of machine representation imposed sequentiality can be removed by utilizing multi-operand and multiple destination type instruction formats. For efficient resource utilization, the instruction formats should not be fixed in size

allowing the program translation mechanism to use the minimum sized template with the correct number of input operands and destination slots for each node operator in the program graph.

The data flow architecture depends on the instruction ready detection mechanism to provide a steady stream of instruction templates for concurrent execution. This mechanism is implemented by either associative memory, hardware register and supervisory logic, or a software controlled recognition scheme. None of these mechanisms is readily adaptable to supporting a variable length, multi-operand and multi-destination instruction template. In chapter three, a data flow architecture is proposed that distributes the instruction forming and ready instruction detection mechanisms in the computer. All processing elements participate in the formation of instructions by adding previously calculated result values to slots in the instruction template provided by

the data flow program compiler. This mechanism effectively unburdens a centralized ready detection mechanism and provides the foundation for use of the variable length instruction format.

There are many varieties of proposed data flow architectures. The introductory chapter broadly categorizes the existing architectures into hardware capable of supporting either static or dynamic program representations.

Although the hardware for dynamic program representation has the potential to extract more concurrency from iterative program expressions, the overhead in label management detracts from the potential execution time by consuming considerable machine resources. Computers designed to execute static program representations are therefore more cost-effective solutions for typical problems encountered in engineering work.

The architecture described in chapter three is capable of interpreting static data flow program representations. Further cost effectiveness is achieved by utilizing a slotted ring communication mechanism. Using this processor inter-connection strategy distributes the instruction formulation and ready detect mechanisms throughout the machine; makes variable size multiple operand. Multiple destination instruction templates possible; and, requires only the number of processing elements to achieve a specific performance criteria to be attached.

6.2 Areas of Investigation for Future Work

Each of the following sections discusses aspects of this computer architecture which could be enhanced or better characterized by additional investigation. In some cases, a recommendation for an implementation strategy is made.

6.2.1 Conditional and Loop Structures

None of the program constructs selected for exercising the SRM architecture involved conditional or looping constructs. The SRM as implemented does not support the additional operators required for these programming constructs. The architecture however can easily support the required switch, merge, and relational type operators. The instruction sets of each processing element would be expanded to include the execution of these program nodes types. Boolean signals would be considered as other data values and directed to the required port number.

Segments of conditional program graphs are often not executed and consequently, these instruction templates remain as un-executed templates in the machine. It is desirable to remove these instructions so that they do not continue to circulate on the ring. One method to accomplish this would be to provide a system template which would be enabled at the final exit from a loop or conditional construction and would be used to inform the template storage units of the instruction template labels that were to be removed.

6.2.2 Data Structures

Data flow languages are side effect free languages. This means that data structures are not altered during the execution of a program but instead, new versions of the data structure are created each time an element is modified. Without control, this could lead to an inordinate amount of processor resources utilized for copying and storing of these entities. Arvind suggests the use of I-Structures to avoid this replication. However, the implementation involves a globally accessible structure memory and associated controller to effect the control mechanisms.

This architecture can incorporate a dual mechanism to accommodate the data structures. When the data structure is relatively small, the elements can be included in the multi-operand instruction template. Alternatively, a structure pointer could be included in the instruction template and processed at either a special purpose processing element attached to the

ring, or a globally accessible structure memory could be attached to all processing elements and their instruction set extended to support structure operations.

6.2.3 Error Control and Interrupt Environment

No attempts were made to use system control packets in this simulation. Obviously, for a practical implementation, some mechanisms are required for control and monitoring of the system components. This can be effected in a straightforward manner by two obvious means.

Additional control and status lines can be associated with the processing elements inter-connecting bus. One of the template storage units or alternatively an additional unit could be the centralized system controller. The control lines could be used to initialize the processing elements at startup, interrogate the status periodically, and perform any other system related tasks. The status

lines could form a bus that would provide information to the system controller when interrogated. Processing elements could signal abnormal status conditions using a system controller interrupt system.

Alternatively, system control packets could be used as the system communications mechanism. System packets could be addressed to single or multiple processing elements and would provide the communications mechanism for the dialogue between the system controller and the processing elements. This mechanism fits into the existing structure, does not require any additional hardware or bus connections, but suffers in that the expected speed of service will be effected by the ring utilization and ring transfer speed.

6.2.4 Implementation Issues

A great number of hardware implementation issues have not been addressed. First, and perhaps the most important is the reasonable expectation for the speed of operation. The arithmetic processing units in the processing elements can be implemented using

contemporary VESI microcomputers and associated floating point accelerators resulting in an expected operating speed for all operations of less than 10 usec. An increase by perhaps an order of magnitude might be achieved by the use of microprogrammed bit-slice components.

The computational power of this architecture is a function of both the granularity of operations and the speed at which the instruction template slots can be injected. As the granularity of operations increases (ie. all high level functional operators) the computational burden shifts onto the processing elements. If the instructions executed are predominately all primitive, the architecture must be capable of moving large numbers of ring slots quickly by the processing elements. It is reasonable to expect to achieve operation of the ring at a 100 nsec rate if high speed bipolar components are used in the interface processors. Lower rates of ring operation will reduce the requirement for this type of implementation strategy with an associated cost saving.

The relationship of ring operating speed and processing element speed with varying intermixes of instruction granularity needs to be investigated. This type of investigation is required before a commitment to a high speed ring implementation is made.

Another implementation issue concerns the use of and amount of associative memory required in the interface processor. If the implementation of this subcomponent was a VLSI package, then it seems reasonable to incorporate as much CAM as practical. Alternatively, the result label holding registers could be implemented with discrete hardware registers and associated comparison logic.

6.2.5 Extensions to the Basic Architecture

Only four topologies of the basic structure were exercised. The two loader architecture consistently yielded the lowest execution times. This is due to this architectures ability to introduce the ready

instructions generated by the compiler (in most cases the IN instructions) into the computer at a higher rate than the one loader method. This observation suggests that program injection mechanisms is critical.

An increased number of template storage units would probably enhance the performance potential. The ring could be divided into quarters or eighths. This would facilitate system control, prioritizing templates on the ring, garbage collection and other system related functions.

Another aspect of the basic architecture that needs further investigation is the inclusion of color registers. This mechanism is used to uniquely label instruction templates and permit the re-execution of a program graph. The color registers would be a global accessible register managed by the template storage units. These units would color templates as the program was initially injected and re-color and duplicate the instruction templates to invoke additional instantiations of the program. The mechanism would

facilitate executing several different data flow programs on the machine simultaneously and would serve to level off the peaks and valleys in the processors utilization curves.

6.2.6 Program File Organization

The actual program file organization needs more investigation. Some attempts were made to remove the compiler and program generation utilities bias in constructing the program files.

The order of instruction template injection is extremely important. It seems reasonable that given the template inter-dependencies, the speed of the ring rotation cycle, and the instruction execution times, that the program file could be organized by the compiler or a utility so that templates are injected into the machine after the required operands have a chance to be calculated.

The program could be considered as a circular list with all the compiler generated ready instructions collected together at the "beginning". Dependent instructions should then be assembled into the circular

list in the next available positions commensurate with the execution delay required to produce the required operands. Such a scheme was not investigated and would probable result in reduced execution times for all the sample programs which exhibited long instruction make ready times.

LITERATURE CITATIONS

[Aarnio80] Aarnio P., Hartimo I., Kronloef K., Simula O., and Skyttae J., "On the Use of the Data Flow Principle in Digital Signal Processing", Helsinki University of Technology, March 1980.

[Ackerman77] Ackerman W. B., "I-Structures Memory for Data Flow Computers", Sm Thesis, Laboratory for Computer Science, MIT, Technical Report TR-186, 1977.

[Ackerman79] Ackerman W.B. and Dennis J.B., "VAL -- A Value Oriented Algorithmic Language: Preliminary Reference Manual", Laboratory for Computer Science, MIT, Technical Report TR-218, July 1979, 80 pgs.

[Adams68] Adams D.A., "A Computation Model with Data Flow Sequencing", Computer Science Dept., School of Humanities and Sciences, Stanford University, Technical Report CS 117, Dec 1968, 130 pp.

[Arvind77] Arvind and Gostelow K. P., "A Computer Capable of Exchanging Processors for Time", Information Processing 77, North Holland, 1977 pp 849-853.

[Arvind78] Arvind and Gostelow K. P., "Data Flow Computer Architecture: Research and Goals", University of California at Irvine, Technical Report TR-113, Feb 1978.

[Arvind79] Arvind, Gostelow K.P., and Plouffe W., "An Asynchronous Programming Language and Computing Machine", Technical Report TR-114a, Department of Information and Computer Science, University of California, Irvine, 1979.

[Arvind80a] Arvind, Kathail V., and Pingail K., "A Data Flow Architecture with Tagged Tokens", Laboratory for Computing Sciences, MIT, Technical Report TM-174, Sept 1980.

[Arvind80b] Arvind and Thomas, "E-Structures, an Efficient Data Type for Functional Languages", Laboratory for Computer Science MIT, technical memo 178, Sept 80.

[Arvind81a] Arvind, Gostelow K.P., "The U-Interpreter", IEEE computer, VOL 25, Feb 1982, pp 42-49..

[Arvind81b] Arvind and Kathail V., "A Multiple Processor Data Flow Machine that Supports Generalized Procedures", Proceedings of the 1981 Symposium on Computer Architecture, 1981.

[Backus78] Backus J., "Can Programming be Liberated from the Von Neumann style? A Functional Style and its Algebra of Programs", CACM, VOL 21, No. 8, August 1978, pp 613-644.

[Barnes68] Barnes G. H., Brown R. M., Kato M., and Kuck D. J., Slotnick D. L., and Stokes R. A., "The ILLIAC IV Computer", IEEE Transactions on Computers, C-17, No 8, August 1968, pp 746-757.

[Bentley79] Bentley J. L., "An Introduction to Algorithm Design", computer, VOL 12, No. 2, Feb 1979., pp 66-78

[Boyer84] Boyer H. D., "Investigation and Simulation of Data Flow Computers", M.Eng. Thesis, RMC of Canada, Kingston, Ont., May 1984.

[Carlberg77] Carlberg J. r., "A Paged Hardware Associative Memory: Preliminary Report", DTNRSC Report 77-0081, US Govt Research Agency, August 1977.

[Caluwaerta83] Caluwaerta L. J., Debacker J., and Peperstraete J. A., "A Data Flow Architecture with a Paged Memory System", IEEE Proc of ninth Annual Symposium on Computer Architecture, 1983, pp 120-127.

[Cornish79] Cornish M., "The TI Data Flow Architectures: The Power of Concurrency for Avionics", IEEE Digital Avionics Systems Conference, Fort worth USA, Nov. 1979, pp 19-25.

[Davis82] Davis A.L. and Keller R.M., "Data Flow Program Graphs", IEEE Computer, VOL 25, Feb 1982, pp 26-40.

[Dennis75] Dennis J.B. and Misunas D.P., "A Preliminary Architecture for a Basic Data Flow Processor" Proceedings of the 2nd Annual Symposium on computer Architecture, Houston Texas, Jan 1975, pp 126-132.

[Dennis80a] Dennis J.B., "Data Flow Supercomputers", Computer, VOL 13, November 1980.

[Eimbinder71] Eimbinder J. editor, "Semiconductor Memories", John Wiley and Sons, 1971.

[Farooq84] Boyer H.D., Farooq M., Smith D.R., and Plant J.B., "Implementation of the Kalman Filter on a Simulated Data Flow Machine", Midwest Circuits and System Symposium, February 1984.

[Foster82] Foster C.C., "Construction and Evaluation of Content Addressable Memories", US Army Research Office Report DAAG-29-79-G-0046, University of Massachusetts, June 1982.

[Fuller77] Fuller S. H., Siewiorek D. P., and Swan R. J., "Computer Modules: An Architecture for Large Digital Modules", AFIPS Conference Proceedings, VOL 46, 1977, pp 637-643.

[Gelb74] Gelb A., editor, "Applied Optimal Estimation", the Technical Staff, The Analytical Science Corporation, The MIT Press, Cambridge, Mass., 1974.

[Glushokov74] Glushokov V. M., Ignatyev M. B., Myasnikov V. A., and Torgashev V. A., "Recursive Machines and Computing Technology", Information Processing 74, VOL 1, J. L. Rosenfeld (editor), North Holland, NY, 1974, pp 65-70.

[Gurd80a] Gurd J. and Watson I., "Data Driven System for High Speed Parallel Computing - Part1: Structuring Software for Parallel Execution", Computer Design, June 1980 pg 91.

[Gurd80b] Gurd J. and Watson I., "Data Driven System for High Speed Parallel Computing - Part2: Hardware Design", Computer Design, July 1980 pg 97.

[Hasegawa81] Hasegawa M., Nakamura T., and Shigei S., "Distributed Communication Media - A Multitrack Bus Capable of Concurrent Data Exchanging", IEEE Proceedings of the Eighth Annual Symposium on Computer Architecture, Minneapolis Minn, May 1981, pp 367-372.

[Jazwinski70] Jazwinski, A. H., "Stochastic Processes and Filtering Theory", Academic Press, New York, 1970.

[Kronloef81] Kronloef K., Skyttae J., Simula O., and Hartimo I, "Simulation of a Digital signal Processing Architecture - Based on the Data Flow Principle", Helsinki University of Technology, TKK-F-A457, Sept 81.

[McGraw80] McGraw J. R., "Data Flow Computing - Software Development", IEEE Transactions on Computers. VOL C-29, December 1980., pp 1095-1103.

[Maekawa81] Maekawa, M. "Optimal Processor Interconnection Topologies", IEEE Proceedings of the Eighth Annual Symposium on Computer Architecture, Minneapolis Minn, May 1981, pp 171-185.

[Oldehoeft77] Oldehoeft, A. E., et al, " The Design of a Software Simulator for a Data Flow Computer", TR-77-2, Department of Computer Science, Iowa State University, Ames, Iowa, March 1977, 23 pgs.

[Pritsker79] Pritsker A. A., and Pedgen C. D., "Introduction to Simulation and SLAM", Systems Publishing Corporation, John Wiley and Sons, New York, 1979. [Rodriquez69] Rodriquez J. E., "A Graph Model for Parallel Computation", Laboratory for Computer Science, MIT, Technical Report TR-64, Sept 1969, 120 pp.

[Rumbaugh75] Rumbaugh J. A., " A Parallel, Asynchronous computer Architecture for Data Flow Programs", TR-150, Laboratory for Computer Science, MIT, Cambridge, Mass., May 1975, 320 pgs.

[Silva81] da Silva J. G. D., and Woods J. V., " Design of a Processing Subsystem for the Manchester Computer", IEE Proceedings, VOL, 128, No. 5, Sept 1981, pp 218-224.

[Silva83] da Silva J. G. D., and Watson I., "Pseudo-Associative Store with Hardware Hashing", IEE Procs, VOL 130, Pt E, No 1, Jan 1983, pp 19-24.

[Seeber63] Seeber R.R. and Lindquist A.B., "Associative Logic for Highly Parallel Systems", AFIPS Conference Proceedings, 1963, pp 489-493.

[Smith85a] Smith D. R., "A Compiler and Utilities for a Data Flow Computer", EE Tech Report 85/9, Department of Electrical Engineering, Royal Military College of Canada, Kingston, June 1985.

[Smith85b] Smith D. R., "Simulation of a Slotted Ring Computer - Volume 1 The Models", EE Tech Report 85/10, Department of Electrical Engineering, Royal Military College of Canada, Kingston, June 1985.

[Smith85c] Smith D. R., "Simulation of a Slotted Ring Computer - Volume 2a Full Ring Model Part 1 and Part 2", EE Tech Report 85/11, Department of Electrical Engineering, Royal Military College of Canada, Kingston, August 1985.

[Smith85d] Smith D. R., "Simulation of a Slotted Ring Computer - Volume 2b Full Ring Model Part 3 and Part 4", EE Tech Report 85/12, Department of Electrical Engineering, Royal Military College of Canada, Kingston, August 1985.

[Smith85e] Smith D. R., "Simulation of a Slotted Ring Computer - Volume 3 Full Ring Model with Variable Result Memory Size Part 1 and Part 2", EE Tech Report 85/13, Department of Electrical Engineering, Royal Military College of Canada, Kingston, August 1985.

[Smith85f] Smith D. R., "Simulation of a Slotted Ring Computer - Volume 3 Full Ring Model with Variable Result Memory Size Part 3 and Part 4", EE Tech Report 85/13, Department of Electrical Engineering, Royal Military College of Canada, Kingston, August 1985.

[Smith85g] Smith D. R., "Simulation of a Slotted Ring Computer - Volume 4 Short Ring Model Part 1 and Part 2", EE Tech Report 85/14, Department of Electrical Engineering, Royal Military College of Canada, Kingston, September 1985.

[Smith85h] Smith D. R., "Simulation of a Slotted Ring Computer - Volume 4 Short Ring Model Part 3 and Part 4". EE Tech Report 85/14, Department of Electrical Engineering, Royal Military College of Canada, Kingston, September 1985.

[Smith85i] Smith D. R., "Simulation of a Slotted Ring Computer - Volume 5 Two Loader Model", EE Tech Report 85/15, Department of Electrical Engineering, Royal Military College of Canada, Kingston, September 1985.

[Takahashi83] Takahashi N. and Amamiya M., "A Data Flow Processor Array System for Large Scale Scientific Calculation", Systems, Computer and Controls, VOL 14, No. 6, Nov-Dec 1983, pp 78-91.

[Treleaven79] Treleaven P.C., and Hopkins R. P., "A Recursive Computer Architecture for VLSI", Proc of the ninth Annual Symposium on Computer Architecture, April, 1982, pp 229-238.

[Watson79] Watson I., and Gurd J., "A Prototype Data Flow Computer with Token Labelling", NCC Conference Proceedings, June 1979, pp 623-638.

[Watson82] Watson I., and Gurd J., "A Practical Data Flow Computer", Computer, VOL. 5, Feb 1982, pp 51-57.

[Wulf78] Wulf W. A., and Harbison S. P., "Reflections in a Pool of Processors - An Experience Report on C.mmp/Hydra", AFIPS Conference Proceedings, VOL 47, 1978, pp 939-951.

[Weng79] Weng K.,- "An Abstract Implementation for a Generalized Data Flow Language"; Laboratory for Computer Science MIT, technical report TR-228, May 1979..

Appendix A

SUMMARY OF SIMULATION RESULTS

Figure	Title
A.1.1	Execution Profile, Full Ring, RM=16
A.1.2	Processor Utilization, Full Ring, RM=16
A.1.3	Ring Utilization, Full Ring, RM=16
A.2.1	Execution Profile, Full Ring, RM=64
A.2.2	Processor Utilization, Full Ring, RM=64
A.2.3	Ring Utilization, Full Ring, RM=64
A.3.1	Execution Profile, Full Ring, RM=128
A.3.2	Processor Utilization, Full Ring, RM=128
A.3.3	Ring Utilization, Full Ring, RM=128
A.4.1	Execution Profile, Short Ring, RM=16
A.4.2	Processor Utilization, Short Ring, RM=16
A.4.3	Ring Utilization, Short Ring, RM=16

3 OF/DE 3

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

- A.5.1 Execution Profile, Two Loader, RM=16
- A.5.2 Processor Utilization, Two Loader, RM=16
- A.5.3 Ring Utilization, Two Loader, RM=16

.NOTE: The n/m label designating curves in these graphs refers to the number of operands (n) and the number of destinations (m) included in each instruction template.

figure A.1.1

Execution Profile, Full Ring, RM=16

FR PROCESSOR UTILIZATION

RESULT MEMORY RM=16

O 2/2 FORMAT
 Δ 2/2 RANDOM
 + 5/5 FORMAT
 X 5/5 RANDOM

figure A.1.2

Processor Utilization, Full Ring, RM=16

figure A.1.3

Ring Utilization, Full Ring, RM=16.

FULL RING MODE
RESULT MEMORY RM=64

figure A.2.1

Execution Profile, Full Ring, RM=64

figure A.2.2

Processor Utilization, Full Ring, RM=64

figure A.2.3

Ring Utilization, Full Ring, RM=64

figure A.3.1

Execution Profile, Full Ring, RM=128

figure A.3.2

Processor Utilization, Full Ring, RM=128

figure A.3.3

Ring Utilization, Full Ring, RM=128

figure A.4.1

Execution Profile, Short Ring, RM=16

figure A.4.2

Processor Utilization, Short Ring, RM=16

figure A.4.3

Ring Utilization, Short Ring, RM=16

TWO LOADER RING

○ 1/2 FORMAT
 △ 1/2 RANDOM
 + 1/5 FORMAT
 x 1/5 RANDOM

figure A.5.1

Execution Profile, Two Loader Ring, RM=16.

figure A.5.2

Processor Utilization, Two Loader Ring, RM=16

2LDR RING UTILIZATION

figure A.5.3

Ring Utilization, Two Loader Ring, RM=16

Appendix B

SIMULATOR SOFTWARE

This appendix describes the simulator models used to obtain the results presented in this thesis. The complete software descriptions and listings are contained in "A Compiler and Utilities for a Dataflow Computer" [Smith85a] and "Simulator for a Slotted Ring Data Flow Processor" [Smith85b].

B.1 The Ideal Simulator

A collection of three extended PASCAL programs implements a basic data flow computer. The simulated processor has the capability to execute data flow programs with up to eight input operands and eight destinations in the instruction template. A primitive input and output capability is included which uses files as external objects from which data values are obtained for input into the modelled graph, and to

which the calculated results are forwarded. Output values are counted when they are moved to the external file medium and this count can be used to trigger a mechanism to halt the simulation.

The simulation program interacts with the user to determine the object code file name and the run parameters. The object code file is prepared using an interactive compiler and massaged into the desired instruction template format. Initially, the code file is generated from the program graph using the optimum number of operand and destination slots in each instruction. Then the number of input operands and the destinations can be specified and the utilities will add copy operators and additional arithmetic operators as required to model the constrained program graph.

During execution, the simulator passes through the object code file checking each instruction template to determine if the necessary operands are present so that the instruction can be fired. If so, the destination template(s) for the computed results are checked to determine if the receiving operand slots are empty. This action is necessary as no queuing is permitted on

the connecting arc in the program graph; consequently, the instruction can not fire if the result can not move into the destination template. If the destination slot is not full, the template is marked ready for execution. No actual simulated arithmetic operations are performed at this time.

When all the potentially ready templates have been found, this simulator executes these instructions with the available processors. The user can specify the number of processors that are to be made available so that the minimum execution time for the algorithm can be determined (an arbitrarily large number of processors) or can restrict the processing resources to investigate the sensitivity of the algorithm execution time to the number of available processors. All instructions are assumed to take one simulated execution cycle and all results are propagated into the destinations immediately making them available for the next simulated cycle.

The execution of the program operators is modularized so that the functionality of the simulator can be easily modified. At present, the simulator can support the basic arithmetic operations (add, subtract, multiply, divide, sin, cos, tan, arcsin, arccos, arctan, square, and copy) some representative program control operatives (less_than predicate, and switch) and the input and output operators (in, out).

B.2 Slotted Ring Simulator

This section describes the implementation and operation of a SLAM based simulator for the slotted ring data flow architecture described in Chapter 3. The processor (figure B.1 which is figure 3.1 repeated for convenience) consists of template storage units, a slotted ring communication system, and a large number of processing elements. The data flow program consists of a number of instruction templates which are loaded into the template store(s) and then injected into the ring. The instructions are completed by the addition of intermediate results as they circulate on the slotted ring.

FIGURE B.1
SLOTTED RING ARCHITECTURE

B.2.1 The Slotted Ring Processor Model

A simulated processor has been implemented to permit its operation to be examined and its performance to be estimated under various conditions. The parameters which are felt to be of primary interest are the number of processing elements, the manner in which the program is injected, and the size of the result memory. The model was assembled to permit these parameters to be varied. Software instrumentation was provided to observe the operation of the template storage units, ring and processing elements.

The discrete event-driven SLAM model consists of the main program and several user supplied subroutines. These routines are detailed below.

PROGRAM SRM

This program is the driver program for the simulation. The makeup of a generalized SLAM discrete event driven simulation model is described in detail in "Introduction to Simulation and SLAM" [PRIT79]. the purpose of the main program is to setup the initial arrays and

to invoke the SLAM processor. The model uses labelled common areas to represent the internal storage of each of the SRM functional devices, the instruction execution times are parameterized and set to default values. The ring is modelled as two arrays that are used to hold the integer and floating point values associated with each slot of the slotted ring.

The processor storage areas are held in arrays with the index into the array indicating the processing element number on the ring. Up to 1024 processors can be used. The simulation is stopped when all required output values have been computed. An output count is maintained on programmer declared files which are used to receive the calculated results.

SUBROUTINE INTLC

This subroutine is called by SLAM to perform any user specified initialization at the start of each simulation run. A user supplied directive file is read to specify display, trace, input and output setup commands. This subroutine creates the initial event which drives the rest of the simulation.

SUBROUTINE EVENT

This subroutine simulates the action of the model. Eight event types have been defined which simulate the activities of the ring rotation, right and left template storage units, and the processing elements. Events are used for software instrumentation permitting the user to trace the operation of individual units or follow instruction templates in the machine.

SUBROUTINE OPUT

This routine is called by SLAM at the completion of a simulation run. It is used to collect the final statistics and the contents of all SLAM gathered statistics and write them to an external file for subsequent statistics gathering and graph generation.

SUBROUTINES DSPRING, DSPTOKEN, DSPPE, and UMONT

These subroutines perform housekeeping operations for the display event on the ring, template storage units, and the processing elements respectively. They are used primarily to interpret the contents of status and state indicators and internal memory of the functional elements of the data flow machine and print formatted english language interpretations of their activity.

SUBROUTINE TSTORERIGHT

This subroutine simulates the activity of the right and left template storage units. The routine maintains three internal buffers used to route and store the incoming instructions to templates according to their completion state and age. The units load program from external object code files, and remove executed templates from the ring.

SUBROUTINE SERVICER

This routine is a major routine in the machine. The actions of the processing elements are simulated by this code. Each processing element consists of the slot interface unit(s) and the arithmetic processor, which operate asynchronously. The result memory is kept in two large FORTRAN arrays which hold the integer part (destination), and floating point part (value) of the calculated results. Due to the size of these arrays, they are declared VIRTUAL and managed by FORTRAN in external disk file storage.

Model Creation and Operation

This SLAM model was created on a HONEYWELL DPS8 computer system running the CP6 operating system version C00. the general procedure for creating and running a SLAM model involves:

1. compile the collection of user written subroutines using the FORTRAN compiler.
2. link the compiled subroutines and the supplied SLAM library into an executable run unit.
3. make the necessary input files for the simulation. These include the input files for the external values, the output files used to collect produced results, the statistics gathering files, and the object code file.
4. produce an IBEX command file to acquire the necessary machine resources, make the file assignments and control the execution of the simulator. The model is normally executed in the batch mode due to its memory size and CPU time requirements.

Appendix C

BIBLIOGRAPHY

[Ackerman82] Ackerman W.B., "Data Flow Languages", IEEE Computer, Feb 1982 pg 15.

[Allan80] Allan S.A., and Oldehoeft A.E., "A Flow Analysis Procedure for the Translation of High Level Languages to a Data Flow Language", IEEE Transactions on Computers, VOL C-29 No. 9, Sept 1980 pp 826-831.

[Allen76] Allen F.E. and Cocke J., "A Program Data Flow Analysis Procedure", Communications of the ACM, VOL 19, No. 3, Mar 1976, pp 137-147.

[Alpert83] Alpert D., Carberry D., Yamamura M., Chow Y., and Mak P., "32-Bit Processor Chip Integrates Major System Functions", Electronics, July 14, 1983 pp 113-119.

[Amamiya83] Amamiya M., Hasegawa R., Namamura O., and Mikami H., " A List-processing-oriented Data Flow Machine Architecture", Proceedings of the 1982 National Computer Conference, AFIPS 1982.

[Arvind81b] Arvind, and Kathail V., " A Multiple Processor Data Flow Machine That Supports Generalized Procedures ", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, Minneapolis Minn, 1981, pp 291-302.

[Bell71] Bell, C.G. and Newell A., Computer Structures and Examples, McGraw-Hill, N.Y., 1971

[Berg69] Berglund, Glenn D., "A fast Fourier Transform Hardware Implementations - An Overview," IEEE Trans. Audio ElectroAcoust., VOL. AU-17, pp 104-108, June 1969

[Bernhard82] Bernhard R., "Computing at the Speed Limit", IEEE Spectrum, July 1982, pp 26-31.

[Brock78] Brock J.D. and Montz L.B., "Translation and Optimization of Data Flow Programs", Laboratory for Computer Science, MIT pp 46-54

[Burkowski81] Burkowski, F. J., "A Multi-User Data Flow Architecture", IEEE Proceedings of the Eighth Annual Symposium on Computer Architecture, Minneapolis Minn, May 1981, pp 327-340.

[Caluwaerta83] Caluwaerta L. J., Debacker J., and Peperstraete J. A., "Implementing Streams on a Data Flow Computer with Paged Memory", ACM/IEEE Proc of Tenth Annual Symposium on Computer Architecture, May 1983, pp 76-83.

[Cassola82] Cassola R.L., "A Floating Point Module for Military Computers", Computer Design, Feb 1982 pp 67-76.

[Chin84] Chin, C. and Hwang K., "Packet Switching Networks for Multiprocessors and Data Flow Computers", IEEE Transactions on Computers, VOL C-33, No. 11, November 1984, pp 991-1003.

[Davis83] Davis A. L., "Computer Architecture", IEEE Spectrum, Nov 83.

[Dennis80b] Dennis J.B., Boughton G.A., and Leung C.K.C., "Building Blocks for Data Flow Prototypes", Proc of the 7th Annual Symposium on Computer Architecture, May 1980, pp 1-8.

[Enslow74] Enslow P.H., Multiprocessors and Parallel Processing, John Wiley and Son., N.Y., 1974 pg 26

[Farrell80] Farrell E. P., Ghani N., and Treleaven P. C., "A Concurrent Computer Architecture and a Ring Based Implementation" Proc of the Sixth Symposium on Computer Architecture, Apr 1980, pp 1-11.

[Flynn72] Flynn, M.J., "Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers VOL. C-21, pp 948-960, September 1972

[Gostelow79] Gostelow K. P. and Thomas R. E., "Performance of a Data Flow Computer", Tech Report 127a, Department of Information and Computer Science, University of California, Irvine, 1979.

[Highbie73] Highbie, L.C., "Supercomputer Architecture", Computer, December 1973, pp 48-58

[Hoshina83] Hoshina T. and Kawai T. et al, "PACS: A Parallel Microprocessor Array for Scientific Calculations", ACM Transactions on Computer Systems, VOL 3, No. 3, 1983, pp 195-221

[Gajski82] Gajski D. D., Padua D. A., and Kuck D. J., "A Second Opinion on Data Flow Machines and Languages", IEEE Computer, VOL 25, Feb 1982, pp 58-69.

[Kishi83] Kishi M., Yasuhara H., and Kawamura Y., "DDDP: A Distributed Data Driven Processor", ACM/IEEE Proc of tenth Annual Symposium on Computer Architecture, May 1983, pp 236-242.

[Koo70] Koo J.T., "Integrated Circuit content Addressable Memories", IEEE Journal of Solid State Circuits, VOL SC-5, October 1970, pp 208-215.

[Kronloef83] Kronloef K., "Execution Control and Memory Management of a Data Flow Signal Processor", ACM/IEEE Proc of tenth Annual Symposium on Computer Architecture, May 1983, pp 230-235..

[Kuck70] Kuck D. J., " A Preprocessing High Speed Memory System", IEEE Transactions on Computers, VOL C-19, No. 9, September 1970, pp 793-802.

[Kuchta82] Kuchta, M., "Developing High Efficiency Digital Signal Processors from Digital Signal Processing Algorithms", Microprocessor Systems development Laboratory, MSDL Tech Note 82-3, pp 11, Carleton University

[Marczynski83] Marczynski R. W., " A Data Driven System Based on Microprogrammed Module", ACM/IEEE Proc of tenth Annual Symposium on Computer Architecture, May 1983, pp 98-106.

[Manuel83a] Manuel T., editor, "Western Europe Looks to Parallel Processing for Future Computers", Electronics, Jun 1983 pp 111-113.

[Manuel83b]: Manuel T., editor, "A Plethora of Projects in the U.S. try Data Flow and other Architectures", Electronics, Jun 1983, pp 107-110.

[Manuel83a] Manuel T., editor, "Advanced Parallel Architectures Get Attention As Way To Faster Computing", Electronics, June 1983 pp 105.

[Manuel83b] Manuel T., editor, "Japan is Busy Trying to Make Manufacturable Data Flow Computers", Electronics, Jun 1983 pg 114.

[McGraw82] McGraw J.R., "The VAL Language: Description and Analysis", ACM Transactions on Programming Languages and Systems, VOL 4, No. 1, Jan 1982, pp 44-82.

[Misunas75] Misunas D. P., "A Computer Architecture for Data Flow Computation", MSc Thesis, MIT department of Electronics and Computer Engineering, 1975.

[Nelson78] Nelson, D. L. and Gordon R. L., "Computer Cells. - A Network for Data Flow Computing", IEEE Proceedings of the fall COMPCON 1978, pp 296-301.

[Requa83] Requa J. E., "The Piecewise Data Flow Architecture, Control flow and Register Management", ACM/IEEE Proc of tenth Annual Symposium on Computer Architecture, May 1983, pp 84-89.

[Srini81] Srini, V. P., " An Architecture for Extended Abstract Data Flow. ", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, Minneapolis Minn, 1981, pp 303-325.

[Souza81] Souza, R. J. and Balkovich E. E., " Impact of Hardware Interconnection Structures on the Performance of Decentralized Software", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, Minneapolis Minn, 1981, pp 357-365.

[Star82] Star Technologies, ST-100 Array Processor System Overview, publication 900000000, preliminary edition, 1982.

[Takahashi83] Takahashi N., and Amamiya M., " A Data Flow Processor Array System: Design and Analysis", ACM/IEEE Proc of tenth Annual Symposium on Computer Architecture, May 1983, pp 243-250.

[Tokoro83] Tokoro M., Jagannathan J. R., and Sunahara H., " On the Working Set Concept for Data Flow Machines", ACM/IEEE Proc of tenth Annual Symposium on Computer Architecture, May 1983, pp 90-97.

[Thomas81] Thomas " I-Structures for Efficient Data Structures", Laboratory of Computer Science, MIT, Technical Memo TM-178, 1981.

[Trealeven81] Treleaven P. C. and Hopkins R. P., "Decentralized Computation ", IEEE Proceedings of the Eight Annual Symposium on Computer Architecture, Minneapolis Minn, 1981, pp 279-290.

[Treleaven79] Treleaven P.C., "Exploiting Program Concurrency in Computing Systems", IEEE Computer, VOL 12, Jan 1979.

[Treleaven82] Treleaven P.C., Hopkins R.P., and Rautenbach P.W., "Combining Data Flow and Control flow Computing", The Computer Journal, VOL 25, Feb 1982.

[Veen76] Veen A.H., "Reconciling Data Flow Machines and Conventional Languages", Conference on Analysing Problem-Classes and Programming for Parallel Computing, Proceedings of CONPAR 1981.

END. / FIN

31 03 87