EFFECT OF DATA FLOW ARCHITECTURE ON PROGRAMMING LANGUAGE DESIGN

BY

KRISHNAN VENKATARAMAN

Bachelor of Engineering

Bangalore University

Bangalore, India

1978

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1981

EFFECT OF DATA FLOW ARCHITECTURE ON
PROGRAMMING LANGUAGE DESIGN

Thesis Approved:

G. E. Hedrick
Thesis Adviser

J. R. Phillips

[illegible]

Norman N Durham
Dean of the Graduate College

PREFACE

This study is concerned with the aspects of data flow architecture. A survey of data flow processors is presented. The two broad classes of languages, procedural and applicative, are considered for the language design for the data flow architecture. Starting from the basic data flow program representation, the study extendends to the high level languages. Method for translating the conventional language to data flow representation is presented. Consideration is given to the conventional structured languages. A general discussion of usage of applicative language classes are presented, without considering specific syntax. The material presented can be extended to specific syntax design and its practical use can be studied from the given general discussions.

I wish to thank my adviser, Dr. G. E. Hedrick, for his guidence and encouragement through out the study. Also I express my appriciation to other commitiee members, Dr. J. R. Phillips and Dr. L. Johnson, for their engouragement.

A note of thanks is given to Dr. H. Olin Spivey from deptartment of bio-chemisty, for his great concern and support. Finally I express my gratitude to my parents, Mr. and Mrs. S. Krishnan, for their support, encouragement and sacrifices.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Ever since Von-Neumann proposed a model for computer architecture, there has been no change in the basic structure of computer organization until recently. The advancement in device technology is responsible for remarkable changes in the generations of computer, in terms of both cost and performance. Future generations of remarkable computing performance are awaiting, due to current research in the fields of micro-electronics and super-conductors. It has been anticipated that it is possible to come up with computer organizations having performance cabibilities of one hundred times better than existing machines, with the help of super-conducting devices (14).

The first generation computers had very primitive instructions, sufficient to handle the problems at that time. The complexity of the problems to be solved increased along with the generations of computers. The current models for computing machines are incapable of fully utilizing the inherent parallelism exhibited by the problem (31, 39, 48, 50, 51). In order to take advantage of inherent parallelism there have been several proposals which lead to the development of parallel processors. The main drawbacks of these machines are:

1. The model is based on the Von-Neumann model. As a result there is a sequential constraint due to the program counter.

2. The existing language models are based on the best adaptation of the current architecture. Hence they do not have ablity to specify the parallel structure of the problem.

3. The architecture is not transparent to the programmer. The programmer must know about the organization so as to partition the problem that could best exploit the parallel nature of the machine.

4. Because of the above reason presently available parallel processing machines, tend to be more or less a special purpose machine rather than a general purpose machine.

5. Efforts have been made to analyze the performance improvement of these type of multiprocessors relative to cost. The analysis showed that the performance is flat after a certain limiting cost. The cost is directly proportional to the number of processors that is being used in a multiprocessor.

In order to overcome these basic difficulties it is necessary to remove the sequencing constraint fixed by the program counter. There is a significant loss of time in fetching instructions and operands from the memory prior to the instruction execution. A new architecture is proposed in which there is no program counter, preventing an explicit sequencing constraint. The proposed machine executes an instruction whenever all of its operands are available. There is no need for instruction fetching, as required in current architectures. Whenever an instruction is ready for execution, it is directed to the processing unit for execution and the instruction is said to be enabled; i.e., ready to fire. The execution of an instruction is based on a basic firing rule. The firing rule is a rule on which the internal hardware is constructed to effect the correct flow of information. These are known as data flow architectures (16, 17, 21, 23, 49, 63). The main aims of this thesis are:

1. To bring out the basic aspects of data flow architecture and a survey of some of the proposed data flow machines.

2. To determine the basic programming method for data flow architecture.

3. To determine whether it is possible to use conventional programming languages on data flow machines and if so, what are the requirements for translation?

4. To determine whether it is possible to simplify the process of translation by using structured languages.

5. To introduce applicative language and its usefulness on data flow machines.

This thesis discusses the basic concepts of data flow machines. It includes survey of some of the proposed data flow machines. Then the effect of programming languages on these architecture is examined. To begin with, the basic aspects of programming on data flow machine is presented. Further, from the point of view of high level languages, a method for translating the conventional language to a data flow language is presented. The translation procedure is simplified by considering the structured languages. Language extentions are presented to facilitate problem specification, that could better utilize the property of data flow machines. Conventional languages are based on the objects, refering to a storage location either directly or indirectly. This concept of object is changed for better adaptation of high level languages on data flow machine. The objects represent values rather than a storage location and this results in a value oriented language. The function evaluation in conventional languages are not 'pure'; i.e., not evaluated by substitution. Lambda calculus languages are based on pure function evaluation. The aspects of value oriented languages and functional languages for data flow machine are presented.

Literature Review

Theoretical basis for the data flow architecture was done by Karp and Miller in 1966 (38). Between 1966 and 1974 there was some work on the representation of data flow programs and models. In 1975, a preliminary architecture for data flow processor was proposed by Dennis (21); this machine is designed to execute data flow programs (20). Misunas (47) extended this model to make it suitable for handling data structures.

In 1978, Texas Instuments developed a distributed data flow machine designed to execute FORTRAN programs (34, 46, 69). Meanwhile, attention is focused to develop data flow processors, taking advantage of the developments in microelectronics. In 1977, Arvind and Gostelow (7) proposed a data flow machine architecture that can be implemented by using large scale integration (LSI). The main idea of using LSI is to favor distributed processing among many processors without any need for partitioning the programs to exploit concurrency. In 1976, Davis (16, 17) proposed a data flow machine (DDM1 - data flow machine # 1) and built at Burroughs advanced system development organization. Rumbaugh (57, 58, 59), in 1977, proposed a data flow multiprocessor for LSI implementation. In 1979, Watson and Gurd (27, 33) gave an account of machine implementation for Manchester data flow architecture. In the following year (28), they gave the design aspects of a highly parallel data driven system (extended Manchester data flow architecture).

Late in 1960's and early 1970's, the concept of data flow has been given a prominant place in the development of optimizing compilers (3, 4, 8, 29, 30). A comprehensive study of these techniques is displayed by Shichi Yi-Lee (60). Considering the language issues for data flow machines, Ackerman (2) has given an account of the properties of the data flow languages based on VAL (Value oriented Algorithmic Language) and ID (Irvine Data flow). Studies on implementation of streams for concurrent computation is done by Dennis and Weng (22) in 1979. A procedure for the translation of a high level data flow language was developed by Allen and others (5).

Following the studies of lambda calculus by Landin and others (32, 40) in 1963, later studies have revealed that functional languages can be better adapted for data flow machines than conventional languages. There have been significant contributions from Backus (9), Berkling (10, 11), Turner (66, 67) and Friedman (24), in the studies of functional languages. In 1977, Treleaven (62, 63, 64, 65) proposed a multiprocessor reduction machine cabable of executing reduction language. In 1979, Mego (42) proposed a network of microprocessors that can execute reduction language. In 1980, attention is focused on very large scale integration (VLSI) for advances in data flow architecture, that can utilize the functional languages (35, 37, 41, 61).

CHAPTER II

DATA FLOW ARCHITECTURES

Data flow architectures can be classified into two types according to the characterstics of the information flow within the processor; viz., data driven architectures and demand driven architectures. A discussion of each of these types follows.

Data Driven

Data driven data flow processors execute data flow programs in a data driven manner. Data driven means that each instruction of a data flow program is enabled for execution if and only if all of the required operands are available. The basic data flow program is represented in the form of a directed graph, in which the nodes represent operators. The nodes are interconnected by arcs along which the data values are conveyed. The presence of data item in any arc is refered as token. The operator i.e., node, is enabled when tokens are present on all input arcs. When an operator is enabled the tokens at the input arcs are lost and the computed value is placed on all of the output arcs. The process of enabling the operator to place a result token on the output arc is called as firing. The operator is said to be fired when this action occurs. An operator is fired only when the following two conditions are met:

1. The tokens are present at all of its input arcs;
2. And no token is present at any of its output arcs.

A typical data flow machine has the following:

1. Memory - to store the data flow instructions.
2. Controller - to control the flow of instructions between memory and the processing units. This unit is responsible for finding enabled instructions and sending them to the operation unit for execution.
3. Processing unit(s) or operation unit(s) - to perform arithmetic and logical functions.

Various methods for the design of data flow architectures have been developed and implemented. The following section considers some of the methods that have been developed to implement data flow machines. The essential programming aspects of a typical data flow machine, present an idea of the programming requirements for using data flow architectures.

Consider the following example program:

```
Input a, b, c ;
     T := SQRT (b**2 - 4*a*c);
     X1:= (-b+T)/2*a;
     X2:= (-b-T)/2*a;
Output X1, X2;
```

It is necessary to generate data flow code in order to execute this program on a data flow machine. The program can be represented in the form of a directed flow graph in which the nodes specify the components of program and the unidirectional arcs specify the data paths. The minimum sequencing constraint is determined by the data paths. This directed graph representation is the usual representation for a data flow program. A data structure of desirable form must be chosen, so that the graphical representation of the program (shown in Figure 1) represents a coding scheme for the data flow machine. The data flow code consists of an operation, input paths, and output path, in the realization of graphical representation. Let there be a maximum of two paths for input and output arcs respectively, from any node. Further, let each node represent only a single operation. One possible coding scheme is shown in Table I. This represents the data flow code for the example program. The given scheme does not represent exact machine code, but it demonstrates the form of data flow implementation.

TI Distributed Processor

The data flow architecture developed by Texas Instruments Inc. (TI) is known as a distributed data processor (DDP). The system consists of processing elements (PE) interconnected by means of a packet switching network. Each processing unit contains arithmetic logic units (ALU), local memory, a controller and an I/O port.

The main features of this system (34, 46), shown in Figure 2 are:

1. A data directed hardware along with a host for software processing.
2. High performance obtained by exploiting the program parallelism
3. The use of high level language programming which makes the data flow hardware transparent to the user.

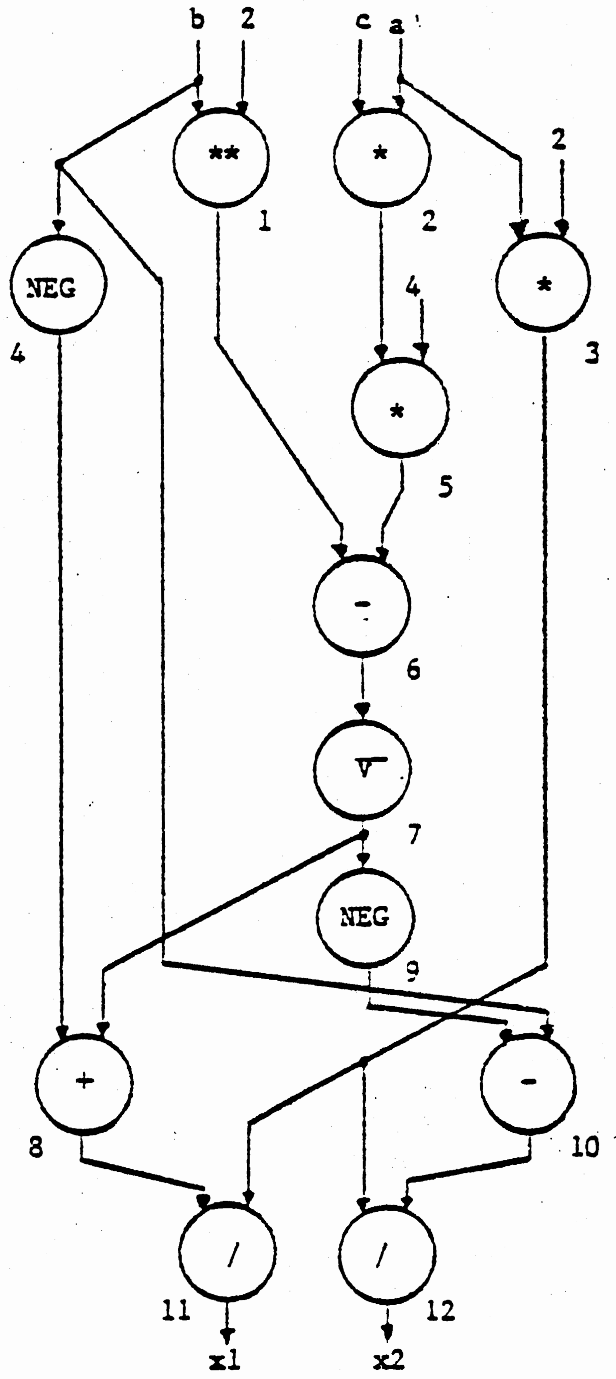


Figure 1. Data Flow Representation of the Program

TABLE I

A DATA FLOW CODING SCHEME

INSTRUCTION	OPERATION	INPUT PATH		DESTINATION PATH	
		1	2	1	2
1	COPY	1.1	-	4.1	10.2
2	COPY	-	2.2	3.1	-
3	**	1.1	(2)c	6.1	-
4	*	2.1	2.2	5.1	-
5	*	3.1	(2)c	12.1	11.2
6	*	5.1	(4)c	-	6.2
7	-	6.1	6.2	7.1	-
8	NEG	4.1	-	8.1	-
9	V	7.1	-	9.1	8.2
10	NEG	9.1	-	10.1	-
11	-	10.1	10.2	-	12.2
12	+	8.1	8.2	11.1	-
13	/	11.1	11.2	(X1)r	-
14	/	12.1	12.2	(x2)r	-

( )r	-	Defines the Result.
( )c	-	Defines Constant to be placed
	-	at the input paths.
i.j	-	Path j of node i.

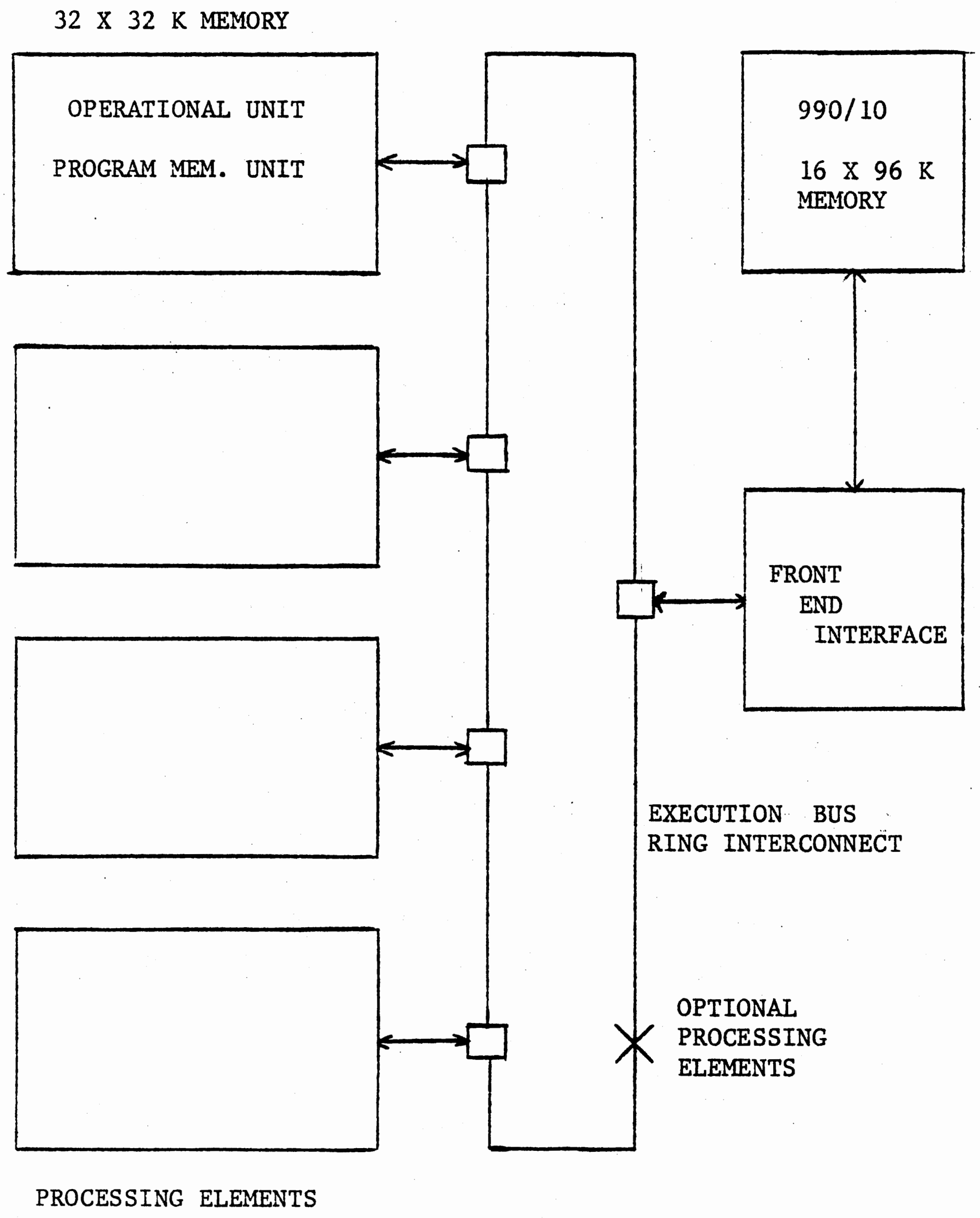


Figure 2. Block Diagram of DDP System

Each processing unit corresponds to a simple data flow processor. Figure 3 shows the layout of simple TI-type data flow processor, consisting of an operation unit and a program memory unit. The operation unit performs arithmetic and logical functions, I/O operations and provides data memory access. The program memory unit maintains an instruction queue which contains instructions ready for execution. The memory unit includes both memory and an update controller unit. Each instruction packet is composed of an operation code specifying the instruction task,an input operand list, a list of successor instruction addresses, and a predecessor count; it is controlled by an update controller. Initially, the predecessor count determines the number of input operands that are required by the instruction prior to execution. The count is decremented by one whenever an input operand arrives at the instruction packet. When the count becomes zero, then the instruction is placed at the rear of the instruction queue. The instructions from the front of the instruction queue are sent to an operation unit for execution. The result obtained from the operation unit is then distributed to all of the successor instruction packets.

An advantage of this system over other data flow systems is that hardware expansion is simple. Hardware expansion is the expansion of the number of operation units of the system in order to obtain a multiprocessor environment. By increasing the number of operation units, it is possible to improve the exploitation of program concurrency. The operation units and memory units are interconnected by means of instruction bus and result bus. The instruction bus connects the program memory unit to the operation unit and it carries instruction packets. The result bus connects the operation unit to the program memory unit and it carries result packets. The processing element, composed of an operation unit and a memory unit, forms the basic building block for hardware expansion. Communication among the various procesing elements is controlled by means of a packet communication network. A typical parallel layout of the processing units is shown in Figure 4.

The TI DDP system includes both the hardware system described above and a separate software system. The basic components of the TI software system are as follows:

1. Compiler: A FORTRAN cross compiler which executes on a TI Advanced Scientific Computer (ASC) and produces object modules for the TI DDP.

2. Link editor: Executes on the ASC to combine mutiple object modules into an executable program for the DDP.

3. Allocator: Partitions programs among the processing elements.

4. Loader: Loads programs into the processing elements assigned by the allocator.

5. Performance monitor: Collects performance statistics while the program is running.

6. Dump formatter: Formats the program dumps for easy readability.

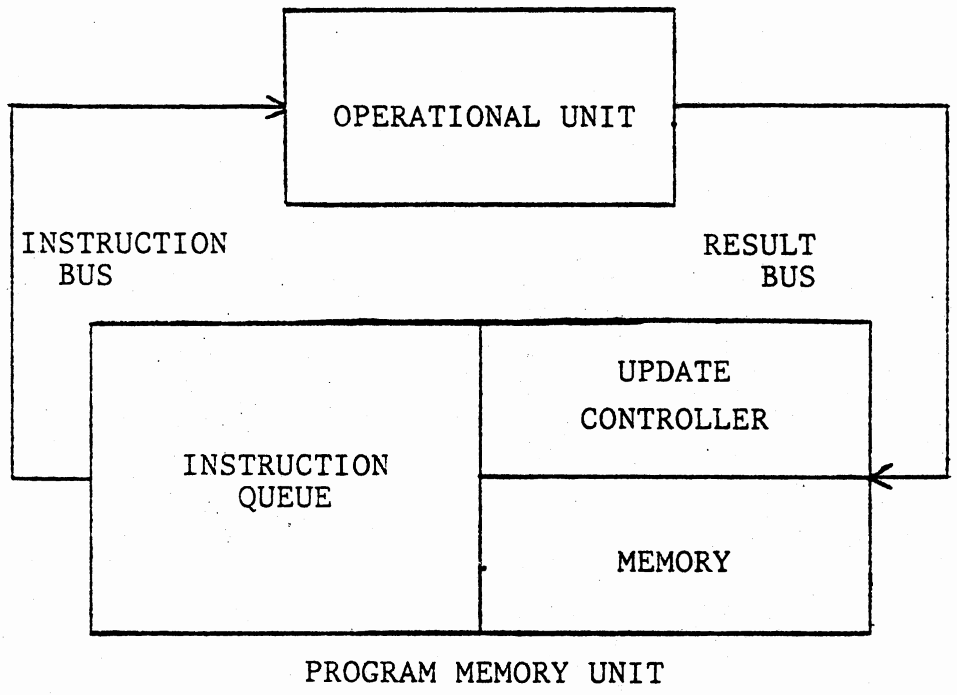
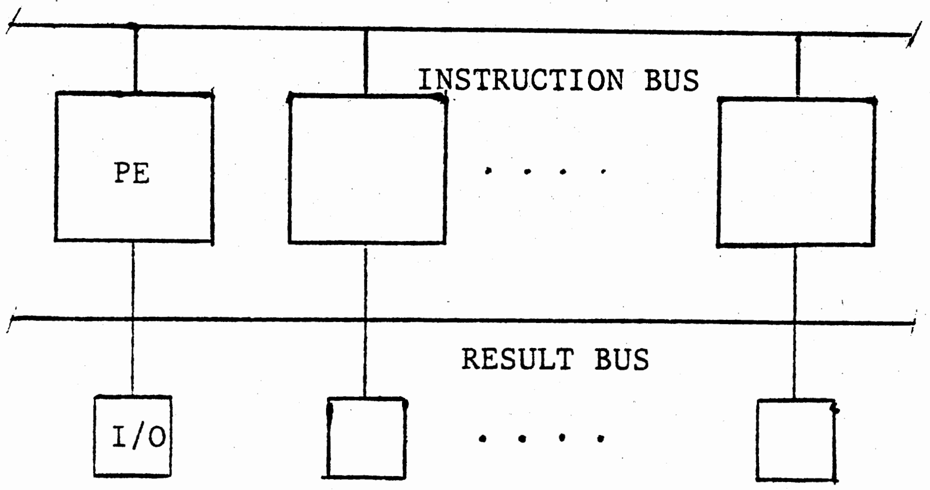


Figure 3. Simple Data Flow Processor



PE - Processing Element; I/O - Input/output unit.

Figure 4. Parallel Layout of Processing Units

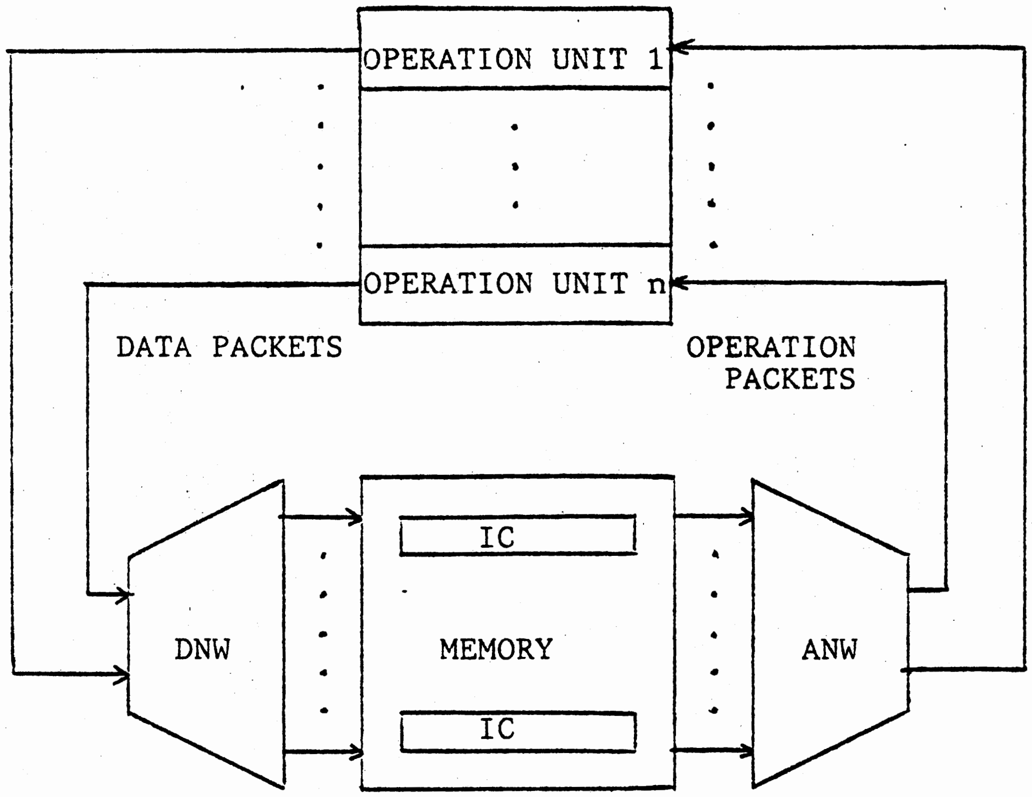
Dennis Data Flow Architecture

The elementary data flow processor suggested by Dennis (21) is shown in Figure 5. The elementary data flow program, represented in the form of directed graph, is stored in the memory within the processor, and is executed in the data driven manner.

The memory, called a cell block, is organized into instruction cells. The structure of the cell block shown in Figure 6, maintains an association table and an instruction stack along with the instruction cells. Each instruction cell is composed of three registers. The first register holds the instruction, specifying the operation to be performed on the data. The operands are held in the other two registers.

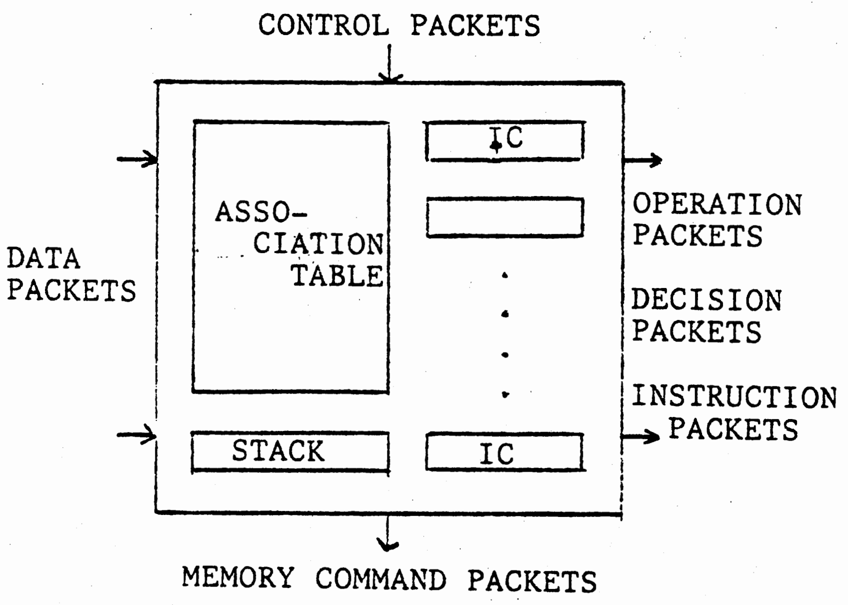
The instruction has three fields containing an operation code and two destination addresses respectively. A cell containing an instruction is enabled, when it receieves all of its necessary operands. Each instruction cell has its corresponding entry in the association table, which maintains the status of the instruction cells. An instruction cell may enter any one of the following as its status, depending on the associated condition of the cell:

1. 'free' - the cell does not contain any instruction.
2. 'engaged' - the cell contains an instruction and waiting for the arrival of the operands.
3. 'occupied' - the cell has received its necessary operands and can be enabled.

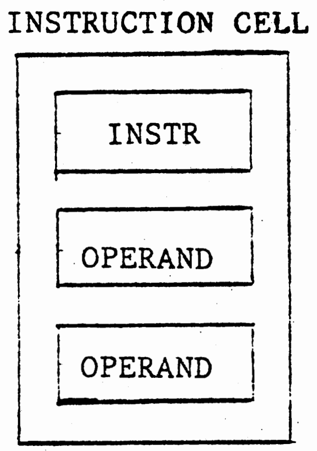


DWN - DISTRIBUTION NETWORK
AWN - ARBITRATION NETWORK
IC - INSTRUCTION CELL

Figure 5. Elementary Data Flow Processor



(a)

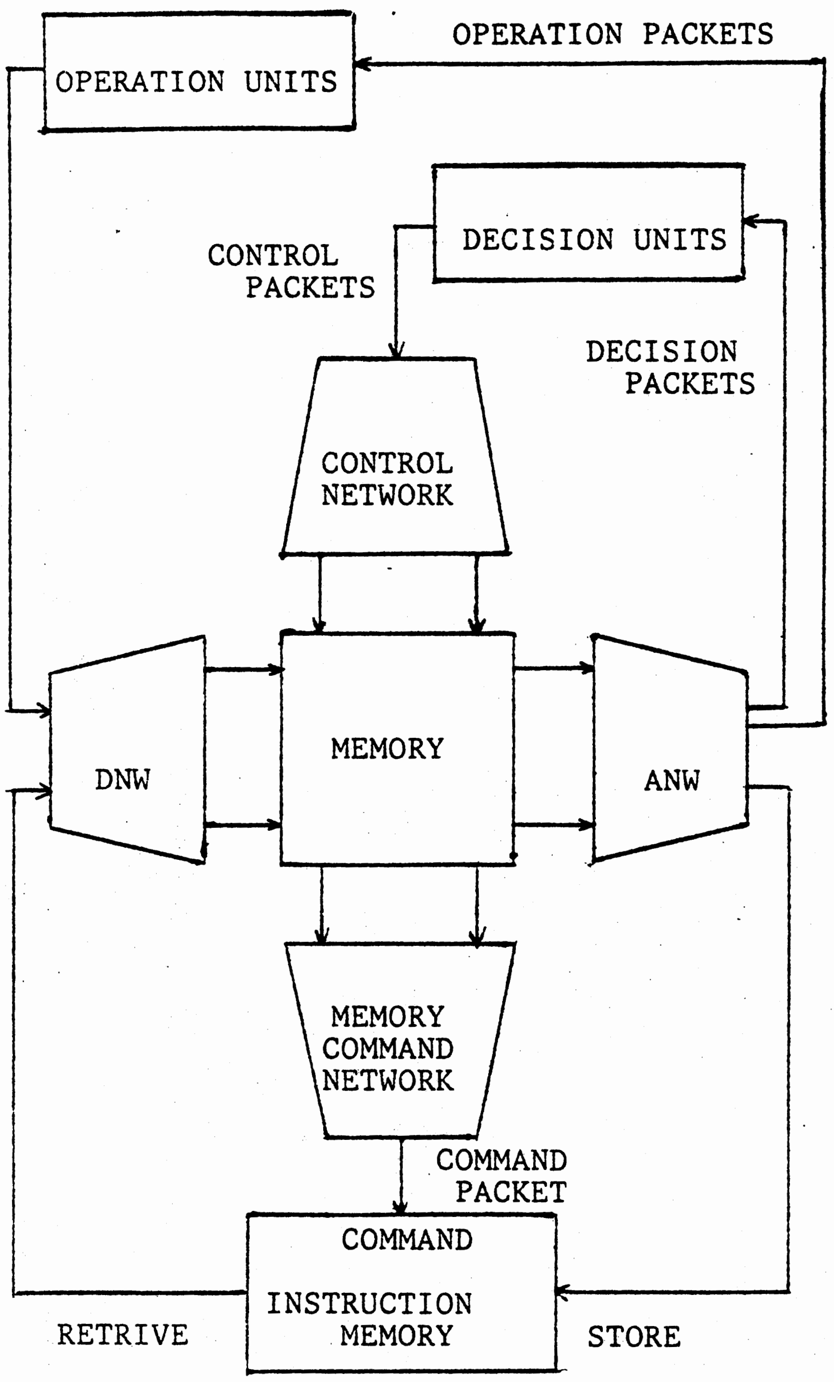


(b)

Figure 6. Structure of Cell Block

The stack maintains the order of the enabled instruction cells. An enabled cell signals the arbitration network that it is ready to send its contents to the operation unit. The operation code of each instruction consists of two parts: one, a specification of the operation unit, and two, a specification of the function. The main purpose of the arbitration unit is to accept the instructions that are enabled simultaneously, and to send only one instruction packet at a time to a particular operation unit. Similarly, the distribution network accepts the data packets from the operation unit and stores them in appropriate register cells in the memory.

The elementary data flow processor is extended to handle decision operators by using decision units and control networks. It is also essential to incorporate a multilevel memory system; such that, only the active instructions of the program occupy the instruction cells of the processor memory. The extended organization is shown in Figure 7. The instruction cells are arranged to act as a cache for the most active instructions of the data flow program. Instruction memory can be slower than the instruction cells. As the computation proceeds the required instructions are retrieved from the auxilary memory, termed the instruction memory, and placed into instruction cells.



DNW - DISTRIBUTION NETWORK
ANW - ARBITRATION NETWORK

Figure 7. Basic Data Flow Processor

Instruction Cell Formats

Instruction packets containing the instruction for arithmetic operations, known as operation packets, are processed in operation units. Control packets contain the instructions for decisions and boolean functions are processed in decision units. There are two types of control packets; viz., gate type control packets and value type control packets.

The gate type packet performs a gating function at the addressed operand register; i.e., the packet is transfered to an operand register if the operator has the truth value. The value type of control packet performs a boolean operation on the operands of the instruction, and provides a boolean value to the addressed operand register. Each operand register has both a gating code and either a data receiver or a control receiver. The gating code performs gating action by comparing its truth value with the truth value of incoming token. The different types of gating codes and their functions are shown in Table II.

The structure of the receiver is shown in Figure 8. The receiver receiving the result packet with data value is termed as data receiver, where as the receiver receiving result packet with boolean value is termed as control receiver. The receiver consists of a gate field, a value field and a data field. The instruction register of the control packet has a 'result tag' associated with the destination field. The result tag indicates whether the control packet is a value type or a gate type. The different instruction cell formats are shown in Figure 9.

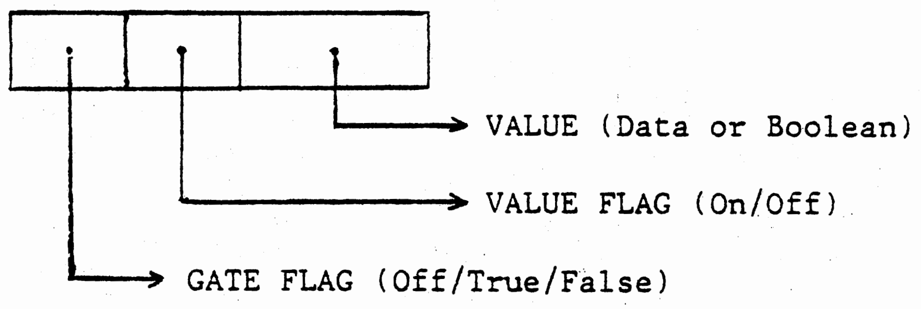
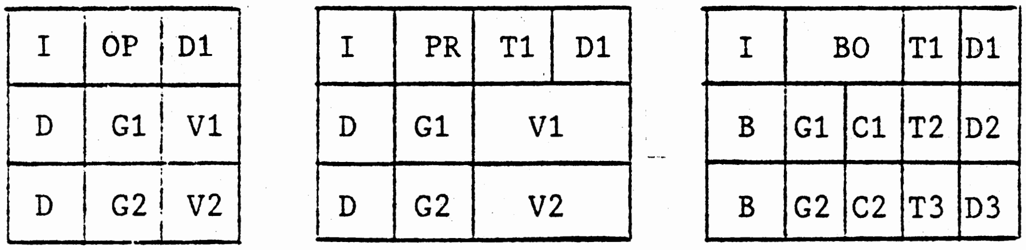


Figure 8. Structure of Receiver

A memory command packet is an instruction to the instruction memory either to store an instruction cell in instruction memory or to retrieve an instruction cell from the instruction memory. The input port through which these commands are placed is known as the command port. The memory command packet requests either the retrieval or storing of an instruction packet of the form (a, x); where, a is the address and x is the data. In the case of a retrieve command the value 'x' is retrieved through the retrieve port and stored in the instruction cell, specified by the address 'a'. In the case of store command the value to be stored, given by 'x' is placed at the store port of instruction memory. The value is stored at the location of the instruction memory specified by the address 'a'. The instruction memory address consists of two parts; viz., a major address and a minor address. One complete cell block is associated with each possible major address. The instruction cells inside the cell block are addressed by the minor address. When the instruction packets are sent from the cell block, the minor address is appended to the corresponding major address.



(a) OPERATOR (b) DECIDER (c) BOOLEAN OPERATOR

OP	- Operation Code
PR	- Predicate Code
BO	- Boolean Operation Code
D1, D2, D3	- Instruction Destination Addresses
T1, T2, T3	- Result Tags
G1, G2	- Gating Codes
C1, C2	- Control Receivers
V1, V2	- Data Receivers

Figure 9. Instruction Cell Formats

TABLE II

GATING CODES AND THEIR FUNCTIONS

Gating code	Action
no	The operand is not gated no gating function.
true	The operand value is accepted if and only if it is associated with a true gate packet.
false	The operand value is accepted if and only if it is associated with a false gate packet.
cons	The operand is a constant value.

Data Flow Architecture Based on Token Labelling

Two types of data flow machines which are based on token labelling architecture are:

1. Arvind data flow machine.
2. Manchester data flow machine.

The concept of token labelling is based on the re-entrant data flow programs. A re-entrant data flow program can be interpreted in two ways:

1. The iterations can be queued and executed one after another as they appear in the program. This is termed as queued interpretation.

2. The different instances of iterations can be executed in parallel and this is termed as unfolding or unraveling interpretation.

If all the instances of the iterations are independent and free from side effects then all the instances can be executed concurrently. In general, if the multiple instances are not free from side effects then it is necessary to distinguish the different instances. A unique label can be assigned to all of the tokens of each different instance to differentiate among them. The operator of any instance will be executed only if all the input tokens having the same label arrives at its input arcs. This is the basic principle used in the hardware of the token labeled data flow architecture.

Arvind Data Flow Machine

The Figure 10 shows the data flow architecture proposed by Arvind and Gostelow (7). It has an array of processing elements. The processing elements of the same column are linked to a common communication bus. Different columns of processing elements communicate among themselves by means of a token communication system. The processing elements can be allocated dynamically. Each column of processing elements is connected to a communication switch in the main communication ring. Each communication switch contains an associative memory.

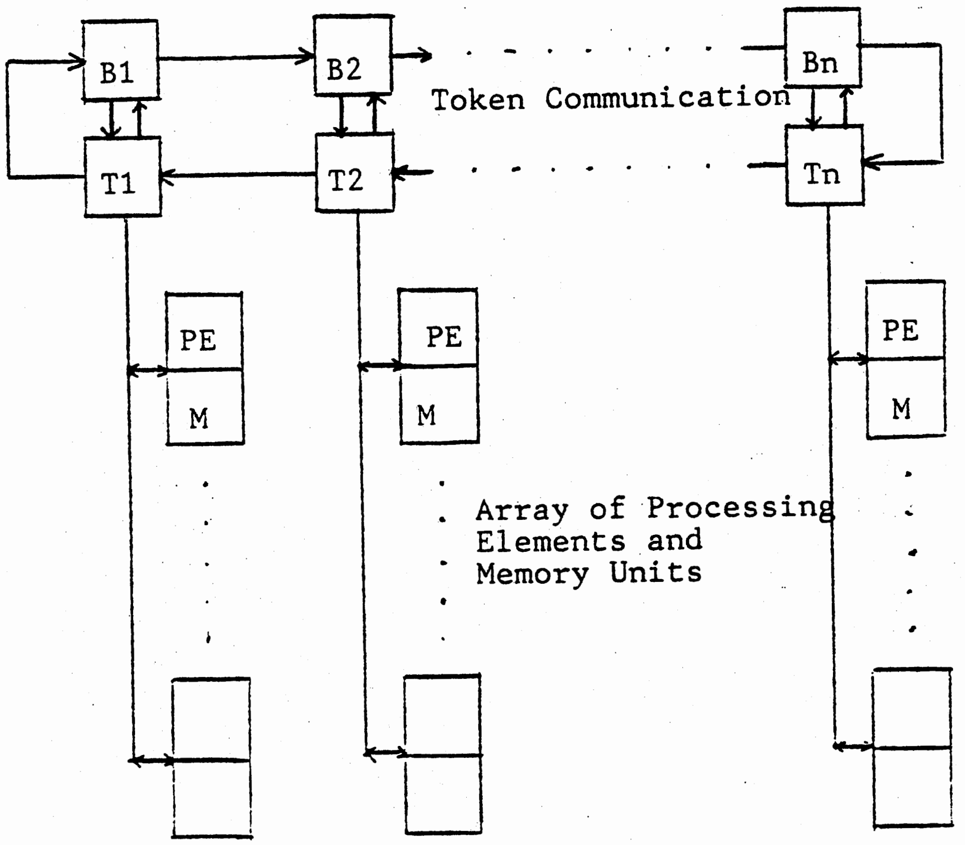


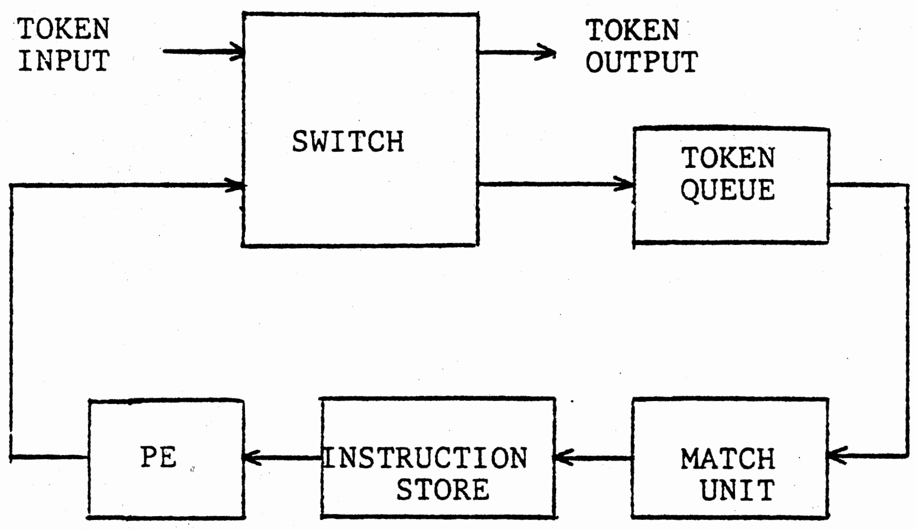
Figure 10. Arvind Data Flow Machine

The associative memory maintains a table of the activity names of the allocated processors in the column associated with it. The communication switch transfers the tokens to and from the communication ring to the column of the processing element that is associated with it. The ring bus of the main communication system is implemented by means of shift registers. The tokens are circulated in the communication ring until it finds a match in any one of the association table maintained in the associative memory. Each processor in a column of processor elements has an independent memory, which is both local to the processor and global to the overall system. An instruction queue and a token queue are maintained in the memory. The main purpose of the queue is to overcome the deadlock due to having both a finite number of storage elements and a finite number of tokens circulating in the communication ring.

Manchester Data Flow Architecture

The basic organization of Manchester data flow architecture (28, 33) is shown in Figure 11. The basic units; viz., token queue, match unit, instruction store, processing elements, and a comunication switch, are connected in a ring. The ring structure implies that the individual functions of different units can be overlapped.

The processing elements communicate to the other peripherals through the communication switch. The peripherals may be another identical ring, a host machine or an input/output device. The token queue is a buffer used to

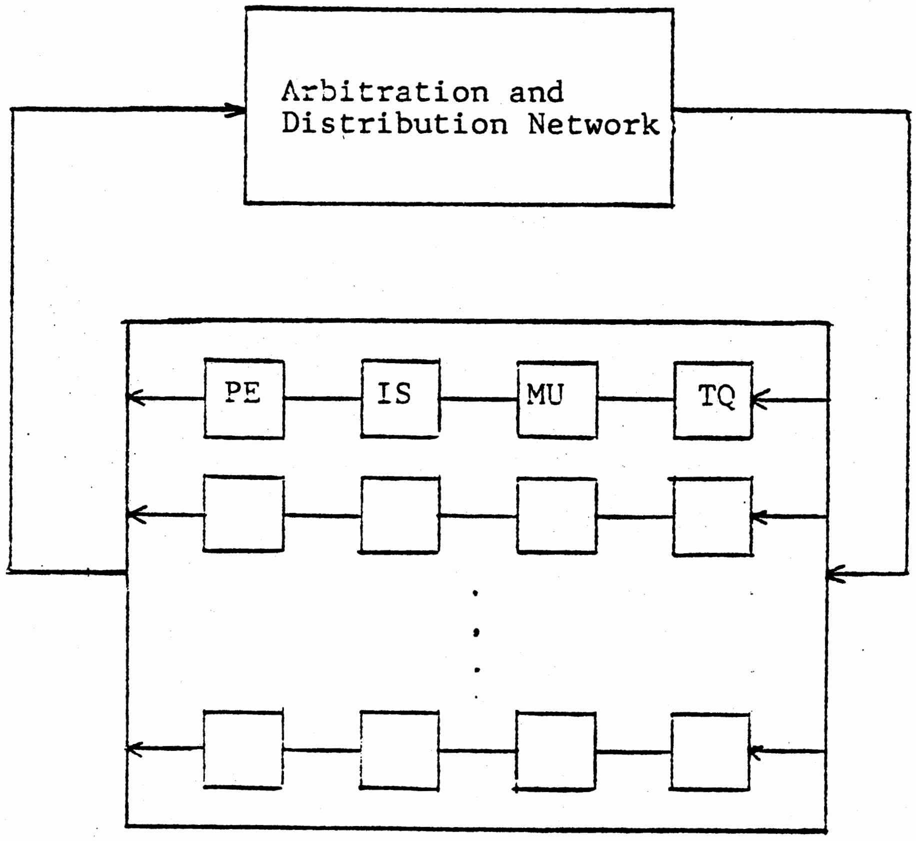


PE - Processing Elements

Figure 11. Manchester Data Flow Organization

prevent the deadlock due to the limitation of the number of tokens that can flow in the ring. The buffer represents a first in first out queue for the tokens. Tokens carry the data value and the destination instruction. Each token is identified by a unique label associated with it. The token label is concatenated with the destination instruction of the token to form the association field. The association field is searched in the associative memory for a match. If a match is found then the token is sent to the instruction store along with the match. If a match is not found then the token is stored in the associative memory. If the operator requires only one input token (or an unlabelled token), then the token bypasses the match unit since it does not require any match.

The instruction store maintains a queue of instruction that are to be executed. It sends the instructions from the queue to the free processor in the processing elements for execution. This basic organization can be extended for an array of processing elements consisting of several rings. The different rings communicate among themselves by means of an arbitration network and a distribution network. The extended organization is shown in Figure 12.



PE - Processing Unit
IS - Instruction Store
MU - Match Unit
TQ - Token Queue

Figure 12. Extended Manchester Architecture

Demand Driven

Demand driven means that, an instruction is executed if and only if the computed result is required. In this type of data flow model the flow of operands among various tasks are implemented in a demand driven manner. This model is also known as a reduction machine or an applicative system. The machine language used for this model is termed either a reduction language, or an applicative language. A machine language program is essentialy a functional program based on constructions appropriate for binary tree representations. Unlike the previous model, the machine level representation for demand driven machines is a linear representation of the flow graph. The two types of data flow architectures share the common feature that both of the models can be programmed using an applicative language.

Any program can be modelled as having a set of inputs and set of outputs mapped by a function. A conceptual computational model can have a set of main processors so that there are as many as the number of output elements in the function. Each output processor demands the required arguments from its sub-processors. A simple arithmetic function demonstrates the concept of demand driven architectures. From the basic definition the computation of arithmetic expression demands results from other sub-expressions, which in turn demands results from others. A sub-expression can become a reducible expression if the sub-expression represents either a constant or a definition. The propagated demand travels along the expression until it hits a reducible expression. The values produced are now propagated backwards, along the same path on which the demands traveled. This is implemented by representing the expression in the form of a binary tree. The root node of the tree places the demand on its successors, which in turn places demands on its successors, etc., until the leaves are reached. The reduction of the sub-expressions is now propagated in the opposite direction until the final result appears at the root node.

The tree representation is analogous to the syntax trees that are generated during the compilation of conventional languages. It is possible to have a list representation of the trees; e.g., postfix, infix, or prefix. Thus, it is possible to have a demand driven model based either on trees or on list representations of trees. The variations in list representations allows user specified languages. Because of the reduction property, some machines of this type are called reduction machines as well as demand driven machines.

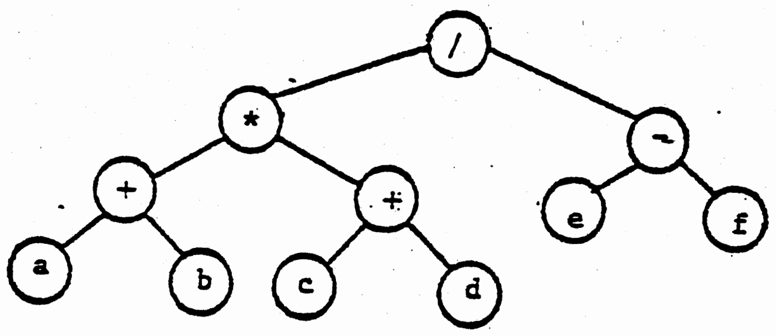
Reduction Machine

Reduction machines are a class of demand driven machines that process the reduction lanuages most naturally. A multiprocessor reduction machine architecture has been proposed by Treleaven (63). It scans for the the reducible sub-expressions and evaluates them concurrently with other reducible sub-expressions. Each processor is controlled by a state transition table for the given reduction language.

Reduction Lanuage

A reduction language uses the list representation of the binary tree structure. It can also be defined as a sub-class of functional languages. The simple example shown in

Figure 13. illustrates different list representations of a simple expression. This illustrates why the language could be user defined. In general, a tree has an operand on its root node, operating on the subexpressions on its left and right successors as shown in Figure 14. In the general case, the root node represents an operator applied to a function on one of its successors and the required operands on the other. Some of the aspects of this type of language are discussed in Chapter IV..



Expression: ((a + b) * ((c + d))/(e + f)
Infix: a + b * c + d / e - f
Prefix: / * + a b + c d - e f
Postfix: a b + c d + * e f - /

Figure 13. List Representations

Machine Organization

The multiprocessor reduction machine (63) consists of the following units:

1. A main memory where the definitions of sub-expressions are stored.
2. The processing units
3. A shift register which holds the expression to be evaluated; The surplus part of the expression resides in a backing unit, which communicates on either side of the shift register.

Main Memory

The basic organization of the reduction machine is shown in Figure 15. A main memory, common to all of the processors, is used to store the definitions. Either an arbitration unit or a packet switching network is essential to communicate with the various processors simultaneously. The main commands for memory operations are load (L) and store (S) operators.

A reducible load sub-expression is of the form (L n); where, n represents a definition name, L performs a memory operation load. This command loads the defintion of the sub-expression from the memory and the sub-expression is said to be reduced or evaluated. An evaluated expression can be stored as a definition in the memory, using the store operator S. An evaluated or reduced expression is of the form (S n v), where v represents the value of the definition name n. Another important use of main memory is to pass parameters between the processors. The parameter

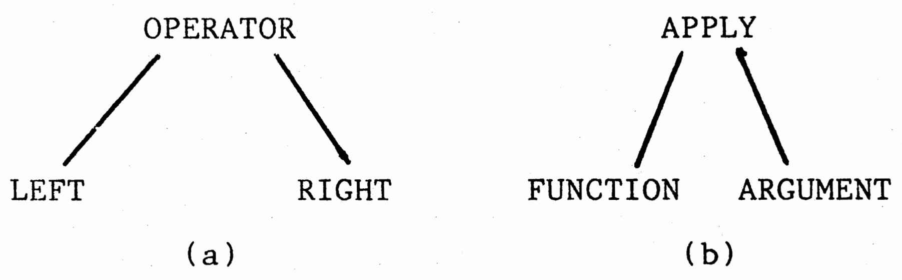
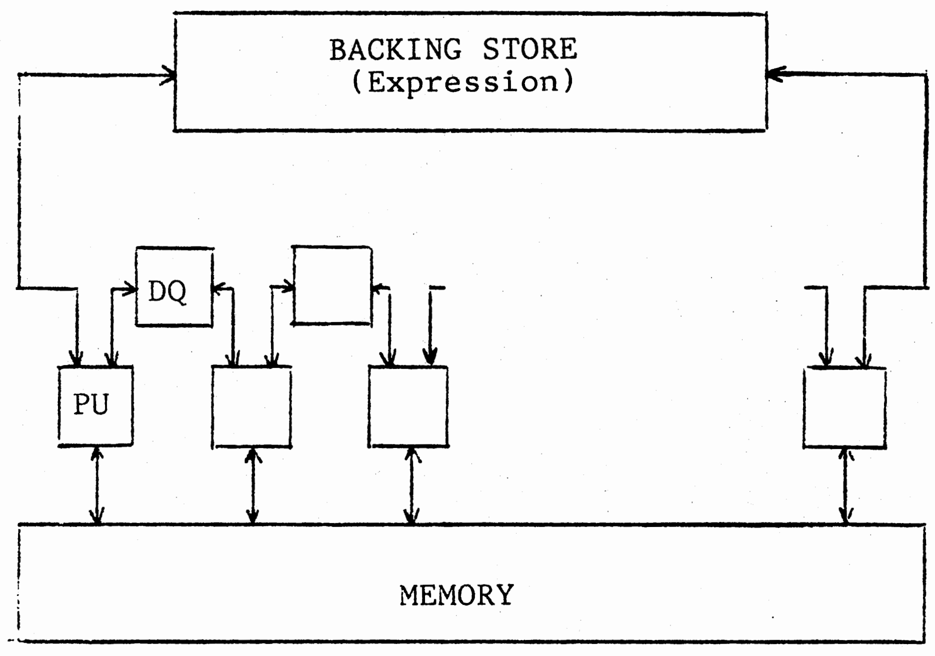


Figure 14. General Representation of Expressions and Functions



DQ - Double Ended Queue PU - Processing Unit

Figure 15. Basic Organization of a Reduction Machine

passing is implemented by means of apply actors, which use load and store. The apply actor can implement both call by value and call by name as illustrated below:

(apply, f, x) - call by name

(apply, f, (L x)) - call by value

Memory organization of reduction machine is simple when compared to data driven models.

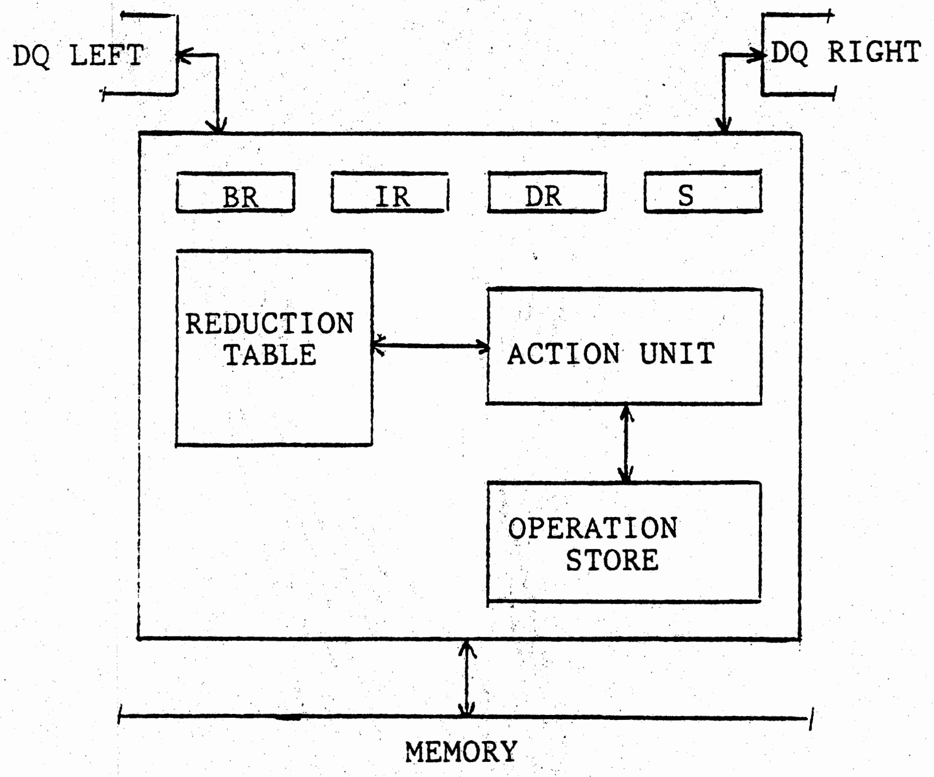
Shift Registers

The shift register is segmented and a processing unit is inserted in between the segments. Each segment of the shift register is a double ended queue along which the expression travels. The two segments of the shift register communicate with the adjacent processors on either side. The segment of the shift register on the right and left of the processor are called as the right shift register and the left shift register, respectively. The shift registers act as a buffers to store the part of the expression between two processing units and mantain the order of the expression. Only one processor can read a queue at a time. As a result two processors can never access the same item simultaneously.

Processing Unit

The organization of each processing unit is shown in Figure 16. The processing unit consists of:

1. Four registers named as follows:
   a. Buffer register
   b. Input register



BR - Buffer Register   IR - Input Register
DR - Direction Register
IR - Input Register
SR - State Register

Figure 16. Organization of Processing Unit of the Reduction Machine

c. Direction register
d. State register

2. A processor controlled by a reduction table, called the action unit.

3. An operator store.

Each processor unit (PU) serves as path for connecting the left and right shift registers. The PU takes in the items of the sub-expressions from the left or right shift register, depending on the direction, indicated by the direction register. The reduction table defines a new direction for the direction register and the actions to be taken by the action unit. The contents of the input register is transfered to the buffer register, and a new item is taken into the input register. The contents of the buffer register either are retained in the buffer register or they are sent to the shift register segment in the opposite direction, depending on the state defined in the state register. The state specified in the state register informs the action unit whether a reducible expression is found or not found. When the state register defines the state that a reducible expression is found, the contents of the buffer register and the input register are sent to the action unit for processing.

The common features of most data flow schemes are summarized:

1. An operation is executed as soon as all its input values are available; i.e., activity by availability.

2. When an operation is executed the input data values are lost, but it can produce more than one output value.

3. There is no concept of data storage such as variables as used in conventional machines and the operations are not allowed to retain internal state representation.

4. There are no control flow specifications; i.e., no sequencing constraints are given beyond the data flow specification.

The main features of demand driven machines are:

1. An instruction is executed only if the result of the computation is required.

2. As in data driven machine, there is no control flow specification. The control sequence is implied.

3. The machine level program is at higher level, when compared to data driven.

4. Architecture is simple, since it is not necessary to update the operand counter of the instruction, when it receives an operand.

CHAPTER III

DATA FLOW BASE LANGUAGE

Elements of Data Flow Constructs

In the data driven data flow machine the instructions are executed only on the availability of all the required operands. The sequential representation of the instructions has no significance. The program representations used are designed to reveal the data dependencies and to express the concurrency present in the computation. The basic program representation at the machine level, is termed as data flow base language. A data flow base language (20) is desired, so that other high level languages can be translated to a common data flow notation. The data flow base language represents a program as a directed graph with the functional operators as nodes. The functional operators are called as actors and are connected by means of links. The incident arcs and the exit arcs of the actors are known as input links and output links respectively.

Link

The data flow graph contains are some non-functional nodes known as link nodes. There are two types of links: data links and control links.

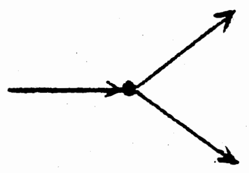
The data links carry data packets; whereas, the control links carry control packets. The data dependencies in the program are specified by the links. The link nodes divert the control or data values in different directions of the link. The representation of data and control link and their firing rules are shown in Figure 17.

Actor

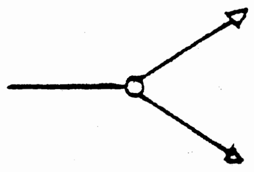
An actor is a functional node of the data flow graph; it represents the operation to be performed on the input tokens placed at the input arcs of the node. There are three types of actors: data actors and control actors and boolean actors. Different types of actors along with their firing rules are shown in Figure 18.

There are two types of data actors; operator actors and decider actors. The operator actor operates on the data packets placed on its input links and place a result packet at its output arc. Therefore, the input and output links of an operator actor are always a data link. The decider actor places a control value at its output control link, acting on the data values appearing at its input data links. The operator of a decider actor is termed as a predicate.

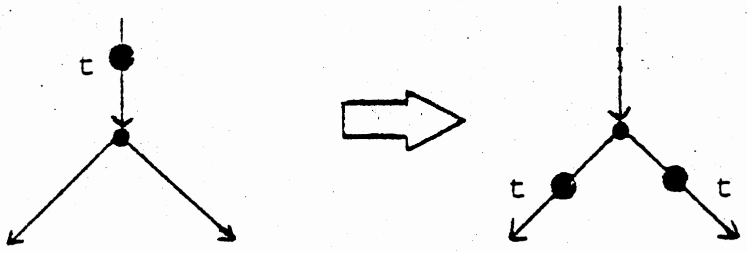
There is no computation involved in the case of a control actor. The flow of data packets from the input link to the output link of the node is controled by a control token. As a result this actor must contain a control link in addition to the input and output data links. The control actors and merge actor are illustrated in Figure 19.



(a) Data Link

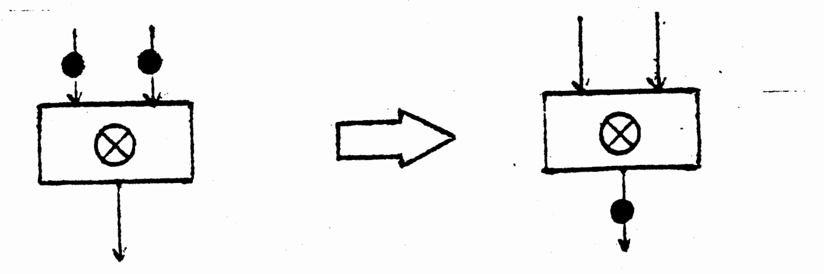


(b) Control Link

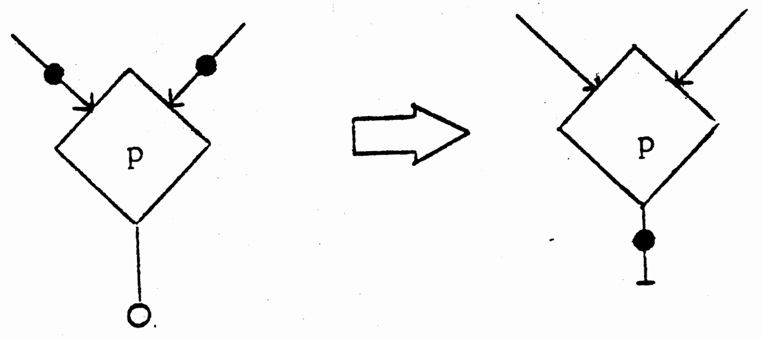


(c) Firing Rule

Figure 17. Data Link and Contol Link

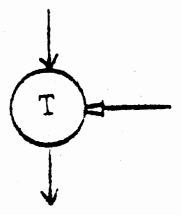


(a) Operator

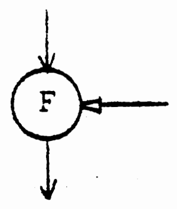


(b) Decider

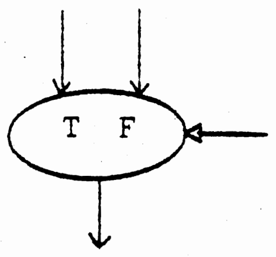
Figure 18. Actors and Their Firing Rules



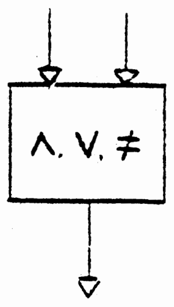
(a) True Gate



(b) False Gate



(c) Merge Actor



(d) Boolean Actor

Figure 19. Control, Merge and Boolean Actors

The boolean operator performs a boolean operation on the input control tokens appearing at the input control links. The result is a control token placed at the output control link of the node.

Data Flow Schemas

A program represented in the form of directed graph is termed a data flow schema. A data flow program is said to be well-formed if and only if it produces only one set of tokens at its output links, when input tokens are present at its input links. There are some restrictions in using the actors in the data flow programs. There is a possibility of creating either a deadlock or non-determinestic behaviour, when arbitrary interconnection between various actors is permitted. This situation can be avoided by using well-formed data flow program representation. Under certain circumstances the data flow program representation may not result in a well-formed schema. A typical example is the case of a iteration construct. Such situations can be handled by using apply actors, to implement the well formed schemas indirectly. A well-formed data flow construct is illustrated in Figure 20. The Figure shows a conditional construct, where the output is always a single set of result for a given set of input, depending on the value of the conditional construct.

Apply Actor

The action of the apply actor is illustrated in Figure 21. Input to the apply actor is a set of tokens from a well-formed data flow schema. The actor is enabled when all the input tokens are present at its input arcs.

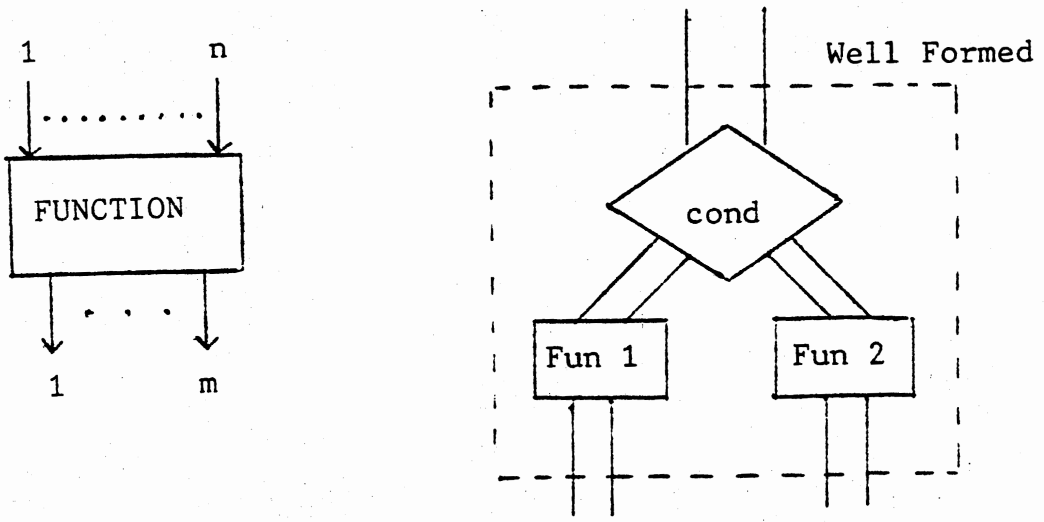


Figure 20. Well Formed Data Flow Schema

The effect of an enabled apply actor is to replace the actor with the specified well-formed data flow schema representing a function. The Figure 21 illustrates how a m-input apply actor is applied to a fuction of m-input arcs. This type of replacement action enables iteration and recursion to be implemented. The action of the apply actor is analogous to call by value in conventional languages. The apply actor is mainly used for procedure activations, and a simple procedure activation is illustrated in Figure 22. The apply actor activates the function F, when the actor receives all of its input tokens. The results of the activated function are placed at the output arcs of the apply actor.

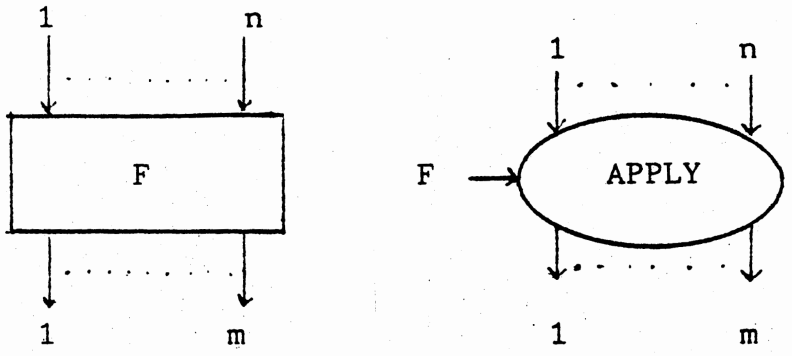


Figure 21. Apply Actor

Data Flow Structures

The values in any data flow program can be represented in the form of a structure, which is a finite, acyclic, directed graph having one or more root nodes. A data flow structure is a set of ½selector : value¼ pairs; where the selector is a distinct identifier represented as an integer or a string, and the value is the data flow value (47). The value may be either an elementary or structured value. An elementary value represents the value of the data item which may be an integer, real, character or boolean value. A structured value is an identifier which represents a structure. The branches of the structure are labelled with the selector. A node in the structure could represent either an elementary node or a structured node depending on the value that is held by the value field of the structure. Simple arrays of single and two dimensions could be implemented as shown in Figure 23 and Figure 24 respectively.

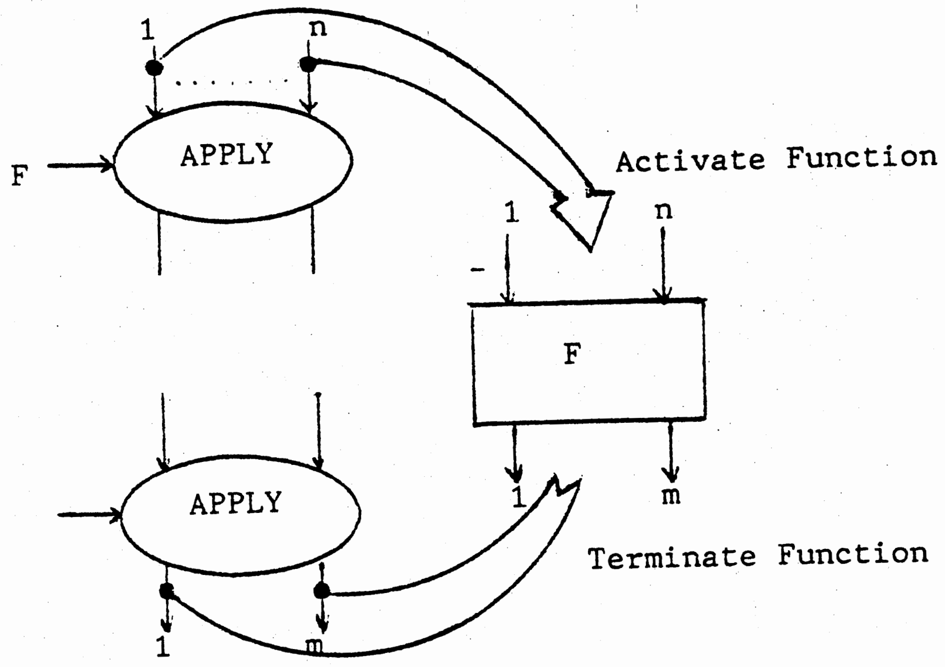


Figure 22. Apply Actor Illustration

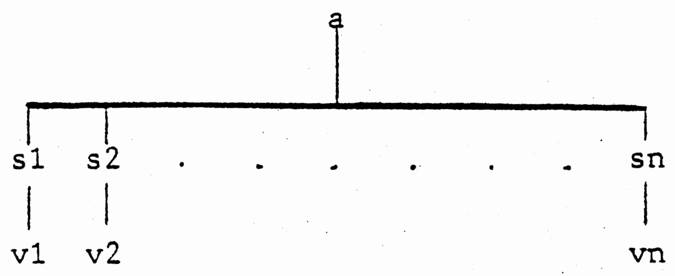


Figure 23. Single Dimension Array Implementation

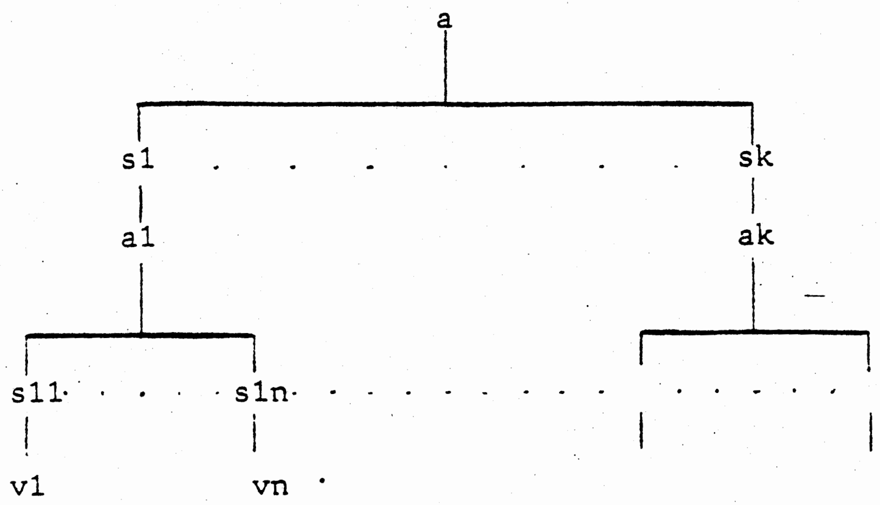


Figure 24. Two Dimension Array Implementation

Select Actor

The select actor selects a structure value corresponding to a selector specified to the actor. The function of the select actor is illustrated in Figure 25. The select actor has two input arcs, through which the actor receives the structure identifier (b) and the selector value (s1) respectively. The actor selects the data value corresponding to the selector value and place the data value at the output arc of the actor.

Append Actor

The append actor creates a new structure value without modifying the original structure. In order to avoid the cyclic structures a reference count can be maintained for each node of the structure. The reference count specifies the number of references to that node. The reference count is decremented by one, for every reference to a node and if the reference count becomes zero, then that node will be deleted from the structure. The function of the append actor is illustrated in Figure 26. The append actor has two input arcs, through which the two structure identifiers (a and b) are placed as input tokens respectively. The actor creates a new structure under the name c. The new structure identifier is placed at the output arc of the actor.

Procedure Implementation and Activation

A procedure may represent one or more functions. A function can be defined as a set of instructions (26). In order that a function to be independent it has to exhibit the following properties:

1. Asycnchrony: The asynchrony property is viewed from the basic property of data flow that an instruction is executed only when all of its required operands are available.

2. Free from side effects: If the data dependency within the function have the same sequencing constraints then it is said to be free from side effects. A procedure represents a function for the given program. From this point of view a procedure is free from side effects, since it does not have any data dependency other than the set of input parameters that are passed to it. The results of the procedure are presented in the form a set of output parameters, must be different from those of the input parameters. It is assumed that the all the computations within the procedure have a scope within the procedure itself. In order to avoid the modifications of the values that are passed by the input parameters the procedure call is implemented similar to call by value in conventional languages.

These properties serve as an important tool for analyzing the parallel execution of procedures or functions. One can visualize the parallelism exhibited by a procedure in two different ways: static parallelism and dynamic parallelism. The static parallelism is the inherent parallelism within the program, emphasized by the graphical representation. Considering the simple example shown in Figure 27, the two functions F1 and F2 can be executed concurrently. The parallelism exhibited by a re-entrent function or a procedure at execution time is termed as dynamic parallelism. The reentrant functions can be interpreted in two different ways: queued interpretation and unraveling interpretation.

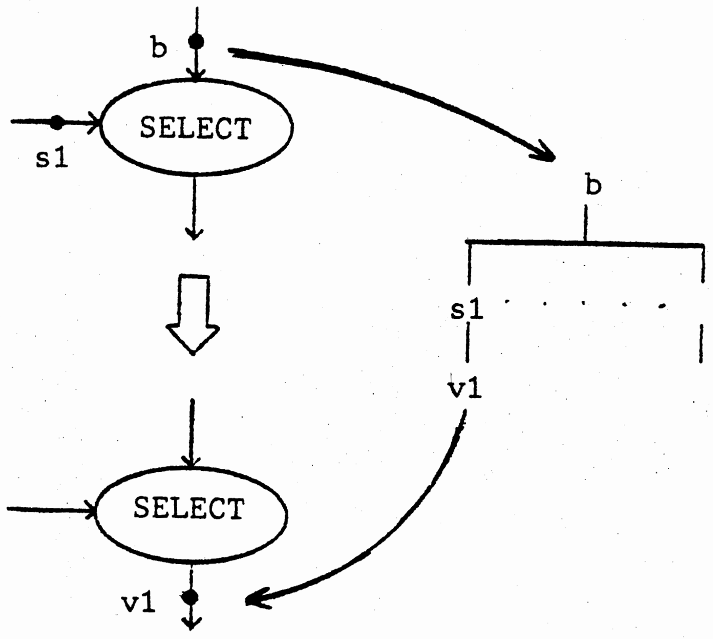


Figure 25. Select Actor

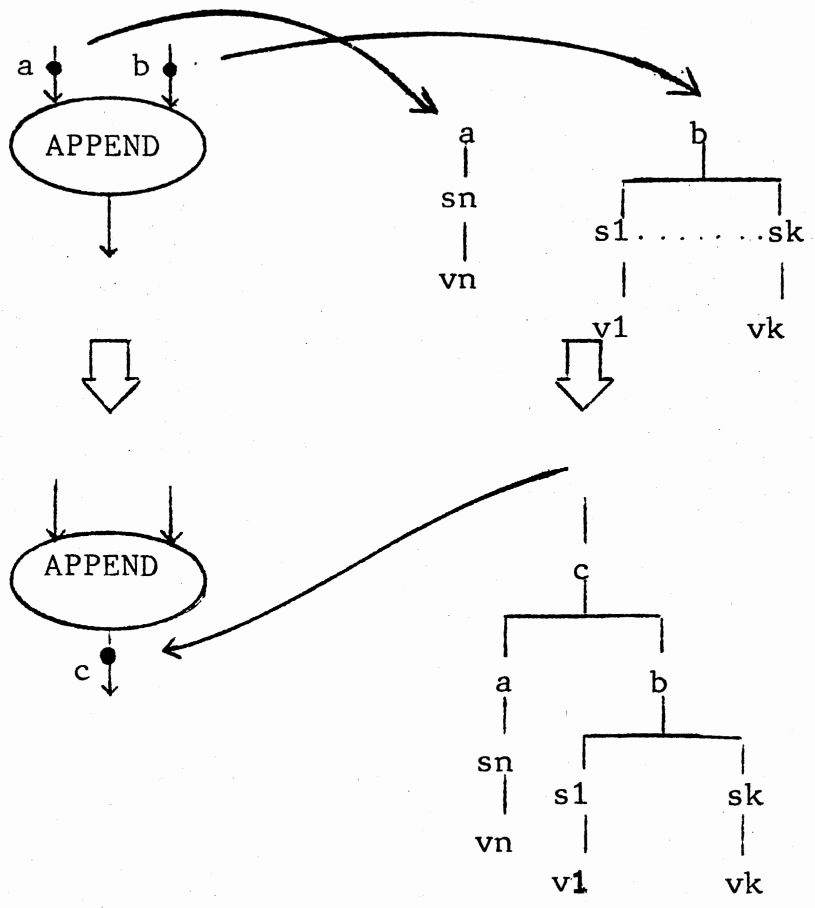


Figure 26. Append Actor

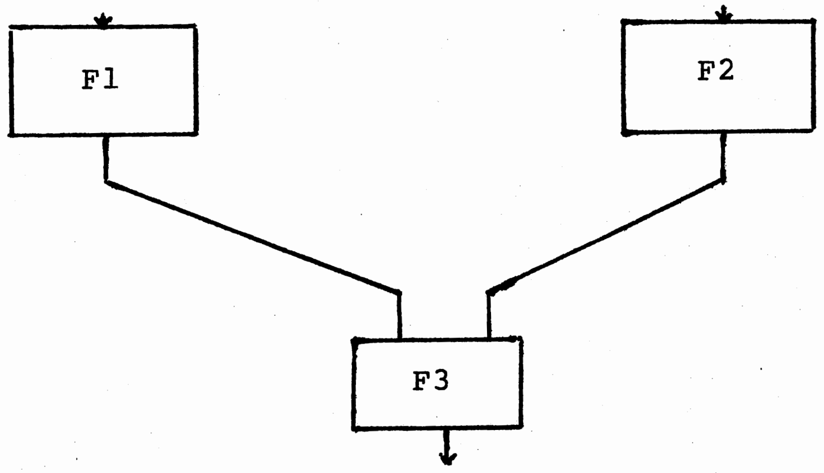


Figure 27. Example to Illustrate Static Parallelism

A queued interpretation considers different instances of the function asynchronously in the same sequence as they appear in the program. The unraveling interpretation exploits the concurreny dynamically. This interpretation could be either implemented in hardware as in the Manchester data flow architecture (33) or by means of a structure implementation with corresponding actors. A token labeled architecture is used in Manchester data flow organization. In general the dynamic parallelism exhibited either by a procedure or by a function within a procedure, that has several instances of execution. It is possible that these multiple instances are free from side effects, and in such a situation, executing all the instances concurrently is worthwhile. One of the methods suggested by Gostelow (26) is to unfold all the side effect free instances and execute them simultaneously. When the multiple instances are not free from side effects then a pipelined method can be used to exploit the dynamic parallelism. In order to implement a pipelined feature there are certain other criteria to be considered in addition to the use of an apply actor.

1. The execution of an instance is defined as an activity. In order to distinguish multiple instances of the same function it is necessary have a unique identification for each activity, referenced as the activity name (26).

2. The destination of the result packets for different instances of the activity may be different. Therefore, it is necessary to group tokens according to the different activity names. One of the methods is by means of colored tokens as suggested by Dennis (20).

3. A particular instance can be executed only if all the input tokens of the corresponding instance

arrives at its input arcs. In other words execution of instance should exhibit the asynchrony property independently.

Dynamic parallelism can be accomplished best by means of a structure representation as suggested by Fanell and others (22). Procedures in a data flow schema can be represented in the form of a data flow structure known as a procedure structure. The same method can also be utilized in a case of re-entrant functions. The components of this structure are the instructions representing a structure value. These components are assigned by unique selector name, usually an integer. Figure 28 illustrates the representation of the procedure structure. The number of components n, depends on the number of activities. Each instruction structure has two components; viz., an elementary value representing the actor, which specifies the action of the instruction and a structure value for destination fields. The selector values of destination field depends on the number of destinations to which the result packet is to be distributed. Each destination field has three components:

1. Selector value of the destination instruction;
2. Input arc of the destination actor;
3. A count of number of operands required by the destination instruction.

In order to implement pipelined concurrency of multiple instances, a structure for each activity must be maintained. Each activity is identified by an activation record. The structure builds up dynamically as the number of activations of the procedure increses. Each activation record has two components, one of which has an elementary value denotes the procedure to which it belongs, and the other has a structure value. The number of selector values in the structure correspond to the number of instructions in the activity. Each instruction selector holds another structure value termed as operand record. The operand record has one elementary value to announce the arrival of all the input operands of the corresponding instance. The number of selector values of the operand structure depends on the number of input arcs (or operands). The activation structure representation is illustrated in Figure 29. It is necessary to define the following operations on the activation structure, in order to implement procedure activations.

1. Create activation(P): Creates a new activation record for newly invoked instance. The newly created activation record has only one component P, the procedure to which it belongs.

2. Insert(A,i,k,v): Inserts a value v to the k th input arc of the i th instance of the activity record A of the activation structure. The 'arrival' count keeps the count of number of input required to enable the execcution of an instance. The result packets are associated with the a number of input count of the destination along with the result token. This count is compared with the updated 'arrival' count and the instance is enabled when they become equal.

3. Remove(A,i): When an instance is ready to be enabled its operand record is removed from the activation structure.

4. Free(A): The activation record A is relesed to the a available memory list, when it is no longer required.

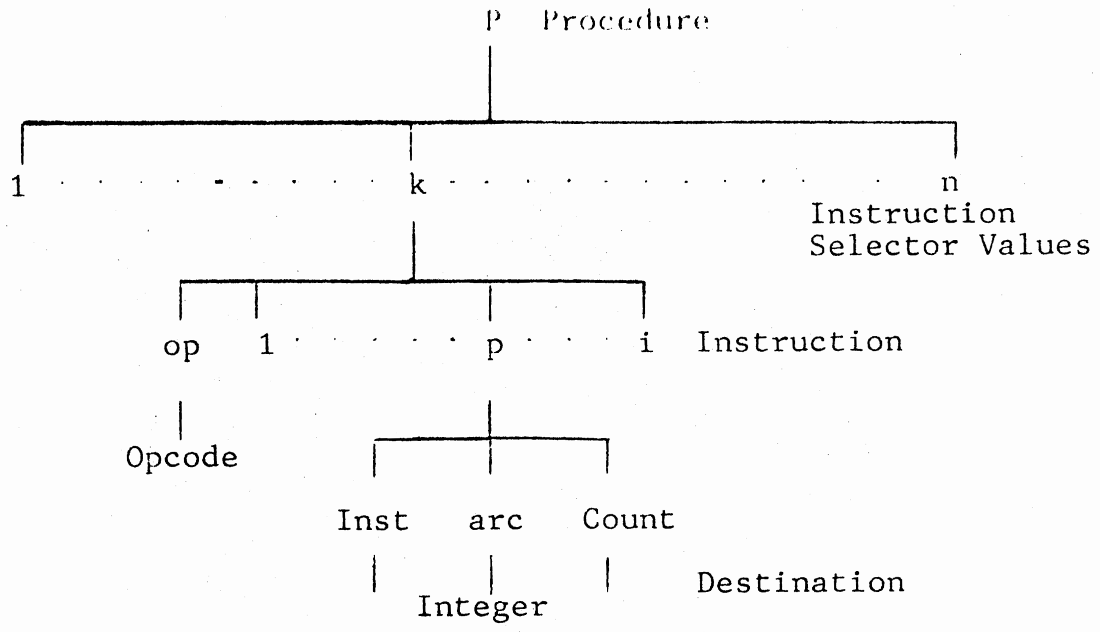

P Procedure
1 . . . . k . . . . n
Instruction Selector Values
op 1 . . . . . p . . . i Instruction
Opcode
Inst arc Count
Integer
Destination

Figure 28. Procedure Structure

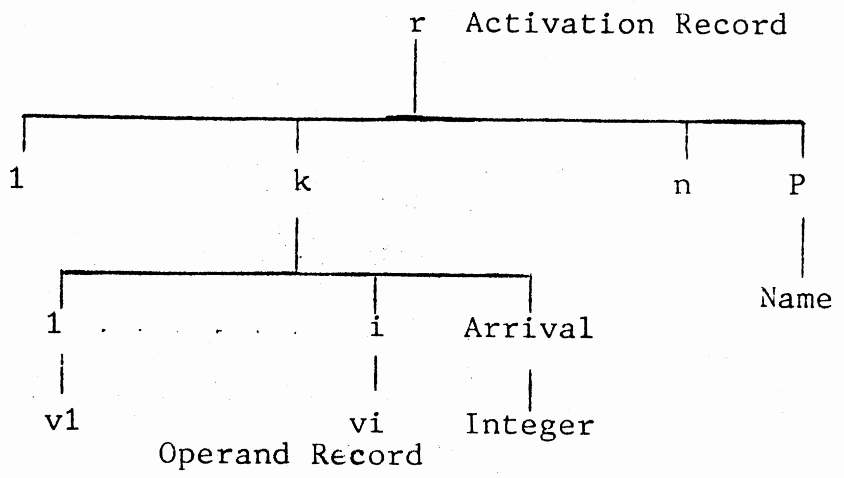


Figure 29. Activation Structure

The procedure activation takes the following sequence of steps:

1. A procedure activation is requested by means of an apply actor.

2. The request causes the creation of an activation record, marking the arrival of a new instance and its corresponding input parameter values.

3. The computed result tokens of different instances create corresponding result structure.

4. The output arcs of the apply actor receives the result structure of a completed activity and the corresponding activation record is freed.

Streams

A stream is a sequence of values of the same type, each being passed successively. In other words, stream is a fragmented data structure and each fragment represents only one element. Streams can be used to implement the data flow structure effeciently.

The main bottleneck in the data flow structure representation can be illustrated by using the following example. Consider two functions, one of which creates an array element by element and the other reads this array element by element. In such a situation the second function has to wait untill the action of the first function terminates, since both functions can not use the same structure simultanously. The stream representation is analogous to the list representation in LISP. The various stream operations (22) can be defined as follows:

1. cons(c,s): Constructs a stream 's' with 'c' as its front element and the remaining elements are the

elements of the stream. A special case of this operation is to create a new stream, when 's' is empty. An empty stream is denoted by ( ).

2. first(s): Choose the first element of the stream 's'. The value is undefined if the stream 's' is empty.

3. rest(s): Choose all the elements of the stream other than the first element. The value is undefined if the stream is empty. The result is empty if the stream has only one element.

4. empty(s): Returns a boolean value of true if the stream 's' is empty.

In summary, the value held by a stream is a sequence. If there is no element in the sequence then the stream is empty. The stream operators are independent of the value of elements of the stream; but the operations on the elements depends on the data type. Cons and rest stream operations results in a stream. The result of first operation depends on the usage in the language; that is, either it may result a value as a part of an expression or it may assign a value to an identifier. The empty stream operation is used to test a stream whether it is empty. The result is true (boolean value) if the stream is empty.

Symbolic Data Flow Representation

In addition to the graphical representation of the programs for data flow machines, symbolic representation is also used, for such programs. The primitive instruction in symbolic representation consists of three fields. The instruction types are classified into seven basic types: DEF, EVAL, MERGE, SWITCH, TG, FG, and APPLY. The second field corresponds to a single identifier. An identifier represents an unique element of the set of class grouped by a function of two dimensions: value and link(path). The computation on any identifier always produces another identifier. For example, the conventional strategy such as

$$i := i + 1$$

is not true in the symbolic representation. If an identifier carries the same value but changes its path then it denotes a different identifier. Semantically, these can be refered as value oriented objects. The third field specifies the operation to be performed. Apart from regular expressions there are special instructions that can be specified in the third field. A STOP to terminate the execution and RETURN to return the result of computation. An identifier not only represents a value that it carries, but it also defines the link. For any value there can be exactly one link. Programs must be constructed adhering to this restriction. Special symbols are prefixed with the identifier to differentiate whether they represent data values or a control values or a definition names. If the identifier is not prefixed with any of the special symbols, then it represents a data value (a member of a set of defined class of numbers). If the identifier is prefixed with the special symbol '@' then it represents a control value (a set of two elements - true or false). If an identifier is prefixed with a special symbol '&' then it defines a function name or a program name.

The DEF type of instruction is used to define or to initialize the values, and thus, has no significance once the program is machine coded. It indicates, that the identifier in the second field is to be initialized with the value specified in the third field. The EVAL instruction indicates that the identifier specified in the second field is to take the value obtained after evaluating the expression specified in the third field. The type of value computed depends on the type of identifier specified in the second field. The instruction types TG and FG corresponds to true and false gates respectively. The instruction types APPLY, MERGE, and SWITCH correspond to apply, merge and switch actors respectively. The instruction representation is illustrated with example in Appendix B.

The symbolic data flow representation presented here is primitive. This can be extended to a powerful and efficient instruction representations. As in conventional assemblers, it may be possible to implement macros to simplify programming task. The importance of this representation is in the area of intermediate representation during compilation of high level language for data flow machines.

Aspects of data flow language can be summarized as follows:

The data flow language represents the machine level representation of a program on a data driven machine. The program representation is in the form of the directed graph. The nodes of the data flow graph represents the instruction, called actors, that is to be executed by the data flow machine. The arcs of the data flow graph represents the data dependencies and indicates the path along which the result of the instruction must travel. The structure operations can be implemented by means of structure actors. Subroutine calls can be implemented by apply actors. The data flow language can also be represented symbolically, which is analogous to conventional machine language.

CHAPTER IV

CONVENTIONAL LANGUAGES FOR DATA FLOW PROCESSOR

Machine level programming on a data flow processor requires a graphical representation of the program. The development of such a representation is tedious. Another important factor to be considered is that the various existing software techniques must be available to use the data flow processor easily and economically. Conventional languages can be used as the high level language for data flow machines. It is assumed that a host machine is available to generate data flow code for the data flow processor.

The main goal of the compiler is to translate the conventional language to a data flow notation to be used at the data flow machine level. The different phases of a conventional compiler are lexical analysis, syntax analysis, intermediate code generation, an optional optimization process, and machine code generation. The optimization process is essentially a flow analysis of the source program represented in an intermediate form. The flow analysis is classified in to two types: data flow analysis and control flow analysis. Control flow analysis deals with the construction, representation, structure, properties of a flow graph or control flow graph. It is a pre-requisite for data flow analysis. Data flow analysis is used to obtain more accurate representation of control flow. It is essentially a pre-execution process of collecting and ascertaining information. The information deals with the data values or the quantities that are preserved, modified and used in the program. The main aim of a conventional optimizing compiler is to utilize the data flow information, in order to obtain optimized machine code that would improve the running time. The analysis ultimately results in the construction of a data flow graph. The data flow graph has no significance at the machine level of the conventional host machine. For the data flow machines, the flow graph representation of the program is significant at the machine level as well as at the execution level. Therefore, the techniquies for the conventional optimization process can be used to translate the conventional programs into data flow machine language.

While considering the compilation process for a data flow machine, the initial phases; i.e., lexical and syntax analysis, remain the same. The optional optimization process, is no longer optional and serves as an important phase of the compilation step. The main uses of flow analysis are:

1. To obtain a graphical representation of the program that could be used for data flow analysis.

2. Restructure the graphical representation according to the data dependencies and sequencing constraints.

3. To exploit the inherent parallelism of the program.

Lexical and Syntax Analysis

Lexical analysis, also known as scanning, seperates the tokens[1], builds a symbol table which contains information about constants, variables, and labels that are used in the program. The syntax analyzer or parser groups the tokens[1] into syntactic structures according to the rules of the high level language. The main functions of this phase are:

1. To detect syntax and semantic errors in the source program
2. To create a parse tree that could be used in the subsequent phases.

An intermediate representation of the program is obtained by using the parse tree. Detailed treatment about these phases could be found in references (3, 4, 15, 52, 53).

Data Flow Restructuring

Motivation

The motivation of restructuring is to find the data flow graph based only on the data dependencies between various nodes of the directed graph. The flexibility of flow of control in the program is a major hurdle in achieving the motivation. The first step towards the restructuring phase is to construct a flow graph.

[1]The term token used here has no relation to the "tokens" that are discussed in data flow architecture. Here it means a group of characters of source program giving a logical meaning such as keywords, identifiers, etc.

<u>Definition:</u> A flow graph is a triple G = (N, A, s) where (N, A) is a directed graph and there is a path from initial node 's' belongs to N, to every node.

N - is a set of nodes

A - is a set of arcs.

Logically, the source program is a flow graph with each statement of the program representing one node and the direction of the control flow between the statements representing the arcs of the flow graph. In order to simplify the flow analysis all the statements where the flow of control is unidirectional can be grouped together, to represent a single node of a program flow graph. Such a group of statements are called blocks. In other words, a basic block is a sequence of statements either at source or intermediate level, which may be entered only at the begining and the statements within the block can be executed with out halt or transfer of control, except at the end of block. The block containing the first statement of the source program is termed the initial node. The block containing the last statement in the program is termed as exit node. The first statement within a block is known as the leader of the block and each block represents the node of the flow graph.

<u>Definition:</u> A block B is a triple (P, I, U) where, P is a set of statements S1, S2, S3, . . . ,Sn ;n 0

I is a set of input variables, and

U is a set of output variables.

Definition: A statement is a string of the form

$$A - p\ I1, . . ., Ir$$

where, A,I1, . . ., Ir are variables and r is a set of operators.

If a statement Sj in a block references a variable A then A is either an input variable or defined in a statement before Sj. If a variable is referenced in a statement, then it is said to be in use in the block containing the statement. If the data value of a variable is modified in a statement either by assignment or reading a value for a variable, then it is said to defined. The definition of a variable may not be unique in a program, since it may be redefined at any other point the program. Each different definition of the same variable has a scope of its own, created by the redefinition. With in the scope of a definition there may be several uses or none. Therefore, it is necessary to collect all the definitions of a variable and its use. In general, a node of a flow graph can have more than one predecessors or may lie with in a loop formed in the flow graph. In such a situation more elaborate analysis regarding the definitions and its use is essential. It is necessary travel along all different arcs connecting a node to its predecessors and successors to analyze the behaviour of the data value of a data item or variable. This analysis is known as data flow analysis and the result of the analysis gives complete information about the definition and use of the variables. From these details it is possible to create a successor list for each definitions representing its corrsponding use. This representation essentialy results in a data flow graph.

The information obtained during data flow analysis can be used to perform certain optimizing transformations. The optimizing tranformations can be viewed in rwo ways, the first one being the tranformations before the data flow analysis, and the other being the transformations after data flow analysis. These transformations are usually termed as block transformations and loop transformations respectively.

Block Transformation

Block transformation plays the dual part in optimization and program level transformation. If a block B1 is transformed to block B2 then they are said to be equivalent, if and only if their values are equal. The value of block is a set of all the values of the definitions with in the block. The value of a variable is based on the value of the expression that defines or assigns the variable. As a result each definiton has a unique value representation. There are four different block transformations (3) as listed below.

1. Elimination of useless computations : If a variable is assigned a statement and it is not active with in its scope then the assignment statement is redundant.

2. Elimination of redundant computations : This transformation involves in detecting common sub-expressions or expressions with in the block and avoids unnecessary repetation of computations.

3. Renaming : If an assigned variable A in a statement of the form

   A - o I1,I2, . . . ,Ir

is replaced by an another variable C that is not active with in the scope of the statement is called renaming. Then all the references of A with in its scope are replaced by C.

4. Flipping : Flipping is the transformation by which two statements or the entries Si and Si+1 of the block can be interchanged, so that the resulting block is equivalent to the original block.

The last two transformations signifies the construction of directed graph for the block.

Loop Transformations

Execution time can be improved by means of several loop transformations. A loop can be defined as a cycle in a flow graph characterizing the following properties:

1. A loop has a single entry node known as header node

2. Loop is a strongly connected region of the flow graph.

Definition: A strongly connected region of a graph G is a sub-graph of G in which there is a path between all pairs of nodes.

Loop Invarient Computations

Some of the computations with in the loop are invarient as long as they are with in the loop and such computations can be moved above the loop header. As a result these computations will not be repeated unnecessarily through out all the iterations of the loop. In order to perform this transformation information regarding the live definitions are necessary. This information can be obtained during the data flow analysis.

Loop Unrolling

The main advantage of loop unrolling transformation is that it is possible to expose more insructions for parallel computation. A loop can be unrolled so that all the iterations can computed in parallel, assuming a number processors, as many as the number of iterations are available. It is also possible to unroll the loop partially and these transformations are illustrated in Figure 30.

Loop Jamming

Another important loop tranformation that could expose more instructions for parallel computation is loop jamming. Merging of two loops together is called loop jamming and it is illustrated in Figure 31. The following criteria to be met in order that two loops can be merged:

1. The loops that are to be jammed should have the same execution conditions. If the number of iterations are different two seperate test conditions has to be used.

2. The computations of the two loops should not be interdependent. There certain situations where this condition can be relaxed.

Loop Unswitching

This transformation is opposite to loop jamming and one of the situation under which this can be used is illustrated in Figure 32. The above mentioned transformations can also be applicable under nested loop situations.

```
for i = 1 to n step 1
ARRAY1(i) = ARRAY1(i) + ARRAY2(i)
end
```

After unrolling by 2:

```
for i = 1 to n step 2
ARRAY1(i) = ARRAY1(i) + ARRAY2(i)
ARRAY1(i+1) = ARRAY1(i+1) + ARRAY2(i+1)
end
```

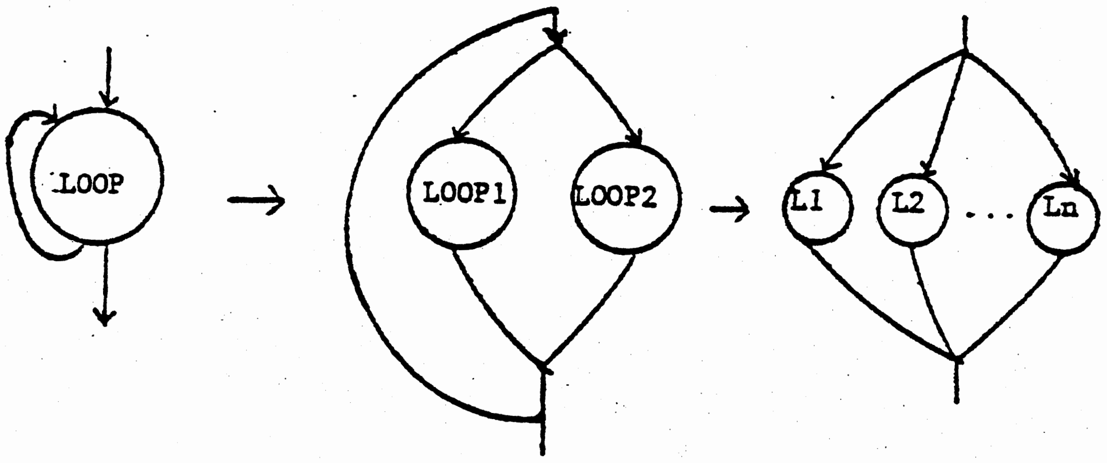


Figure 30. Loop Unrolling Illustration

```
for i = 1 to n step 1
ARRAY1(i) = 0
end
for i = 1 to n step 1
ARRAY2(i) = ARRAY3(i) + CONST
end
```

After loop jamming:

```
for i = i to n step 1
ARRAY1(i) = 0
ARRAY2(i) = ARRAY3(i) + CONST
end
```

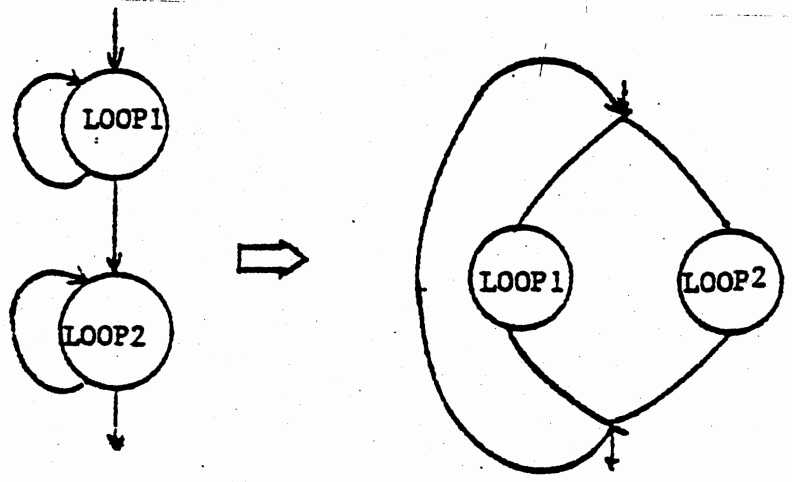


Figure 31. Loop Jamming Illustration

```
for i = 1 to n step 1
if (cond) then
          ARRAY1(i) = ARRAY1(i) * ARRAY2(i)
          else
          ARRAY1(i) = ARRAY1(i) / ARRAY2(i)
end
```

After loop unswitching:

```
if (cond) then
          for i = 1 to n step 1
          ARRAY1(i) = ARRAY1(i) * ARRAY2(i)
          end
               else
               for i = 1 to n step 1
               ARRAY1(i) = ARRAY2(i) / ARRAY3(i)
               end
```

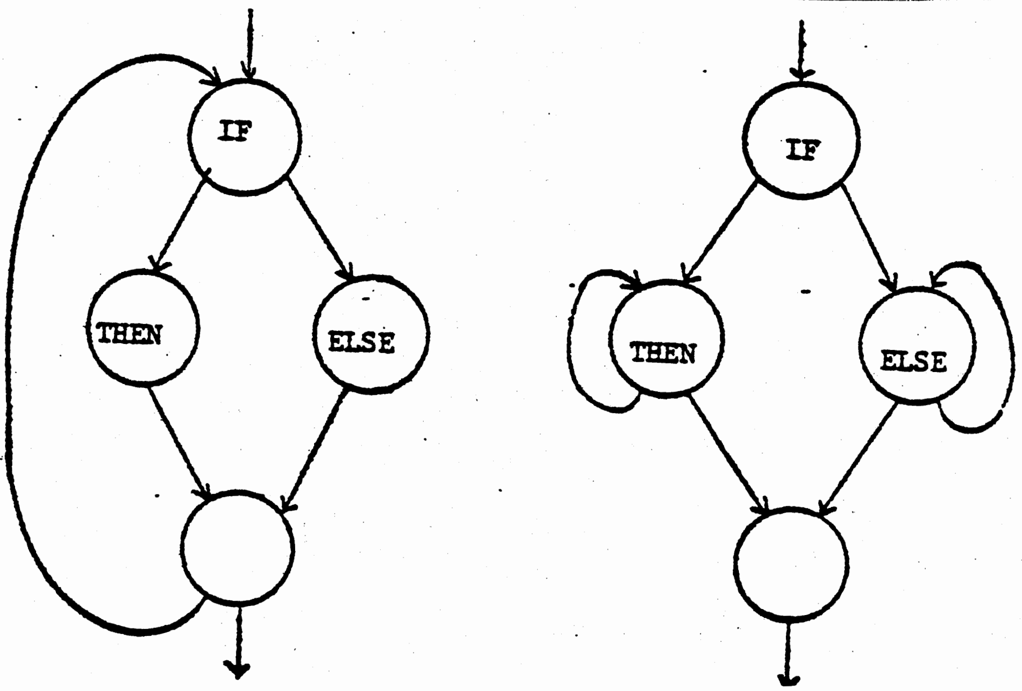


Figure 32. Loop Unswitching Illustration

Restructuring

The intermediate form of the program is partitioned to basic blocks and a flow graph is constructed with basic blocks as nodes and their successor relationships as arcs. The construction of flow graph could be a pre-pass or as a part the previous phase. This is an essential requirment during the first level of restructuring. The details of flow graph construction can be found in references (29, 60).

The second level of the restructuring process is data flow analysis. The data flow analysis is used to obtain the necessary data flow information utilized during the final restructuring stage and optimization of data flow graph. The motivation of this analysis is to determine reaching definitions to establish the relationship between the data dependencies amoung the various blocks. There are two methods for data flow analysis:

1. Iterative data flow analysis.
2. Interval data flow analysis.

A three phase method for interval data flow analysis is implemented in PL/1 by Allen and Cocke (6). A comparative study of the two methods of data flow analysis is given in the report of S. Y. Lee (60).

Final Restructuring

This phase is the final restructuring in the sense that a complete data flow graph will be available at the end of this phase. During this phase the use and definitions amoung the various blocks are linked by means of a successor list at the generated node. The generated node at each block is considered individually and all the other blocks are scaned for their use. For each definition of the block a list is created with a header containing the information about the definition. The live definitions of the other blocks give the information about the use of a definition. If there is a use of the definition the use node is inserted in the corresponding list. At the end of scan, the lists are transfered as a successor lists of the respective definition nodes in the block.

Optimization

The data flow graph present at this stage is sufficient to generate desired data flow machine code. In order to improve the run time and expose more parallelism certain optimizing transformatins can be performed. The notation of the variables are completely lost at this situation and they do not have any significance. The data flow graph is completly based on the flow of data between the various nodes. Each block now represents a data flow graph, consisting of one or more nodes where the computation is involved. Thus a block forms a cluster of nodes and it can be termed as cluster node. The nodes involving the computation can be termed as atom node. An interval therfore consists of one or more cluster nodes and the interval header is a cluster node. For each cluster node there is an information about the the reaching definitions and live definitions. A loop may consists of several cluster nodes or a cluster node itself form a self loop. The optimization transformations are simplified, by applying only to loops with in the intervals. The effect of optimization plays a dual role under the following situations:

1. Move a cluster node out of an interval to expose more parallelism or to avoid redundant computations with in a loop.
2. Split a cluster node into one or more cluster nodes to expose more parallelism or to avoid redundant computation with in the loop.

In order that a cluster node with in an interval to be exposed for parallelism the uses of the cluster nodes should have live definitions at the header of the interval and the generated definitions should not have live definitions at the header. The reason for this is due to the fact that use of a definition at any cluster will be live at the interval head if and only if it is not redefined in any of the path between the head to the point under consideration. At the same time, a cluster node may use a definition and redefine it. If the redefined definition has an use at any point between the header and the definition then it will be live at the header. It means that a definition being used at every iteration and redefined with in a loop. Another way of interpreting this transformation is that a cluster node can be moved out of a interval if and only if the successor list of all the atom nodes with in the cluster does not contain any atom nodes with in the interval.

The computations with in a cluster node may contain more than one independent computation. In such cases a cluster node can be split into two or more clusters, all of which can be computed in parallel. When a cluster node itself represents a loop then the same rules as that of loops in intervals holds good. The different situations exposing the parallelism and loops are illustrated in Figure 33. A procedure implementation of opimization phase is shown in Figure 34. Different types of loop transformations are already illustrated. The loop unrolling is essentially same as spliting of the cluster node. Here the loop representing a cluster node is split into a number of cluster nodes as many as the number of iterations in the loop. The loop jamming is essentially merging two different cluster nodes, which are identical but different in the sense of computation. the loop switching involves in duplicating a cluster node under a special situation; for e.g. in an if construct as illustrated earlier.

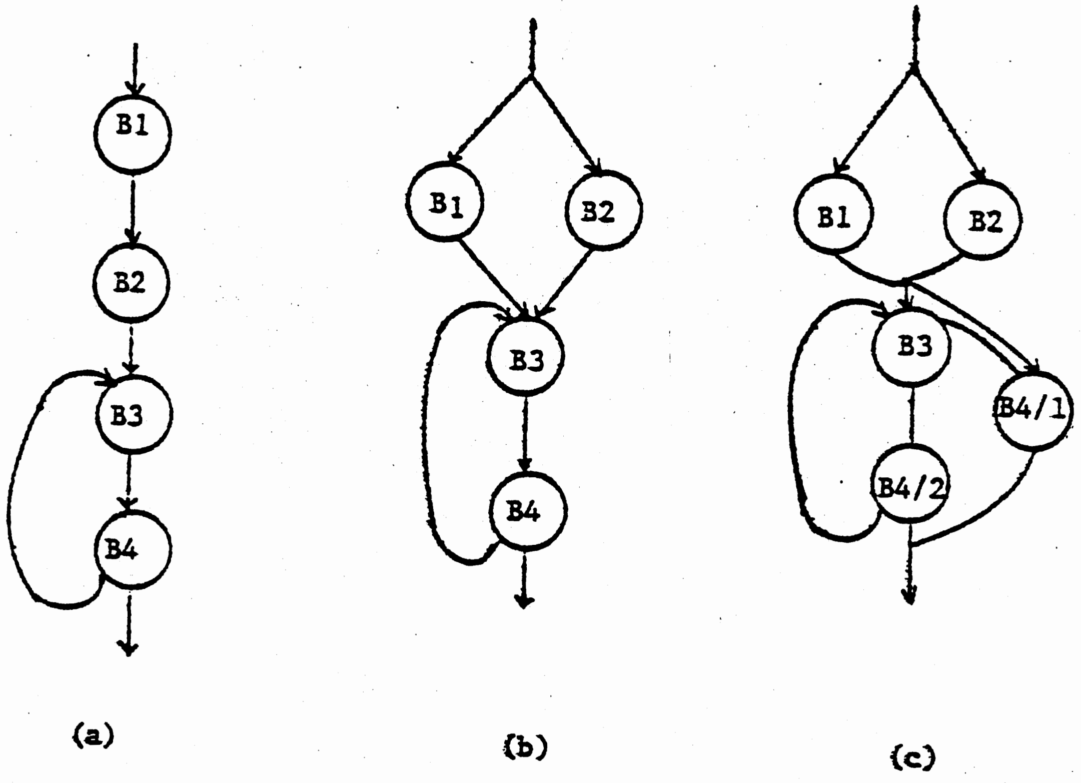
Data Flow Code Generation

The main task of this phase is to interface the generated data flow graph to the data flow machine under consideration. It depends on the machine organization. One of the method of generating the data flow code is to interpret the genarated data flow graph to a data flow notation. The data flow notation or the base language is mapped, either by hardware or software translator. This transformation is essential, in order to implement the conditional and loops in the data flow machine. These interpretations are already discussed in the earlier chapter. Each cluster node of the data flow graph has a successor list. All the successors of a cluster node can be computed concurrently. The situation is different in the case of a atom node. The successors of a atom node specifies the destination nodes for the computed result. Although the splitting of the cluster nodes to expose paralelism, it is not absolutly essential because of the properties of the data flow. But it could be helpful under the situations where it is possible to allocate different sets of computations to different sets of processors.

Structured Languages

A program is said to be well structured if the program segments are easy to understand and the control paths between the segments of the program are minimal. The conventional languages can be classified either under absolutely unstructured or structured. By structured, it means that it is possible to develope well structured programs using such languages. The main objective is to find a suitable language or a subset of a language which will always result to structured programs.

In conventional language the flow of control i.e., the order in which the statements are to be executed in sequence, is sequential. The sequential flow can be altered by means of conditional or unconditional transfer statements. As a result programs can be developed with out any control flow restrictions. This is the cause for the complexity of the data flow analysis. The analysis could be simplified by restricting the use of control statements, which ultimatly leads to structured programs.



(a) A Flow Graph with Interval = (B1, B2)
  and Interval 2 = (B3, B4)
  Assume B1 and B2 are Independent,
  Loop Invarient Computation exists in B3 and B4

(b) Exposing Parallelism in B1 and B2

(c) Exposing Parallelism by removing Loop
  Invarient Computations in B4

Figure 33. Exposing Parallelism

```
OPTIMIZER: procedure
   for each interval of the flow graph do
        for each block j in the interval do
        FLAG = 0
        call MOV_BLK_OUT
              for each entry e in the block j do
                  if ENT_IN_USE(e) = LIV_DEF(j) then do
                  scan the tree for definitions
                        if definition = LIV_DEF(j) then do
                        create a new block for the entry
                        include the new block in SUCC_LIST of
                                the predecessor of j
                        if FLAG = 0 then call MOV_BLK_OUT
                        end if
                  end if
              end do
     end do
  end do
end OPTIMIZER
```

```
MOV_BLK_OUT: procedure
    if LOC_USE(j) = LIV_DEF(h) and GEN_DEF(j) = LIV_DEF(h)
    then do
    include j in SUCC_LIST of predecessor of h
    delete j from the current SUCC_LIST
    FLAG = 1
    end if
   return
end MOV_BLK_OUT
```

Figure 34. Optimization Procedure Implementation

Language Representation

There is no need to be alarmed as a result of the above mentioned restriction in the language representation. The same effect of the control statements can be well established by means of conditional iterative constructs. A typical grammer of such a language is shown below in BNF.

```
<statement> ::=< simple statement >|< compound statement>
<simple statement> ::=< assign stmt >|< proc stmt >|
                               < call stmt >
<compond statement> ::=< set of stmts >|< cond stmt >|
                                 < iter stmt >
<assign stmt> ::=< simpl asgn >|< compute asgn >
<simpl asgn> ::=< identifier > := < identifier >|< constant >
<compute asgn> ::=< identifier > := < expression >
<proc stmt> ::=< proc keyword >< identifier >< set of parms >
<call stmt > ::= CALL <identifier> <set of parms>
<set of stmts> ::= BEGIN (<stmtatement> |<set of stmts>) END
<cond stmt> ::= IF <cond> THEN <set of stmts>
            IF <cond> THEN <set of stmts> ELSE <set of stmts>
<iter stmt> ::= <steped iter> |<cond iter>
<steped iter> ::= ITER <identifier> := <limits> <set of stmts>
<cond iter> ::= ITER <cond> <set of stmts>
```

Conventional compilation techniques employ a source to source conversion to develope an intermediate representation of the program in the form of triples or quadraples. Such intermediate form is essentially based to suit the conventional machine architecture. It is necessary to consider a suitable intermediate representation that will simplify the construction of data flow graph construction. In an earlier discussion it has been emphasized that a control flow analysis is a prerequisite for the data flow analysis. This type of data flow analysis is known as low level data flow analysis A high level data flow analysis (5, 8, 30, 56) eliminates the control flow analysis requirement. The main advantages of this language representation are:

1. High level data flow analysis.
2. Simplification of data flow analysis.
3. Improvement in program structure and readability.

Language Extentions

The main idea behind the use of the structured language is to reduce burden of the data flow analysis. It is worthwhile to focus the attention to reduce the number of passes with out loosing much of effeciency in code generation. Through some language extentions, it is possible to take advantage by exposing the parallel structure of the problem. Live variable analysis and loop optimizations are required to remove loop invarient computations. By including an iteration construct UNROLL, it is possible to inform the compiler explicitly, that the loop is to be unrolled to expose parallelism. Using the locality of effect the burden of detecting the loop invarient computations can be reduced or ignored. The locality of effect means that the data dependencies among a set of instructions is fixed. Thus the use of data item at different blocks of program are independent. By defining a scope to a data item, the data flow analysis of the data item can be restricted with in the scope, rather than scanning through out the program whereever it is used. Scope is a block or set of instructions over which the definition and use of a data item is effective. The problem of side effect is to be tackled while exposing more computations for parallelsim. Particularly, the side effect is more pronounced when a loop or a function or a procedure is unrolled. Under these circumstances it is important that different activities of the same computation are independent. The definition of a data item in one activity should not effect the computation in the other parallel activity.

Consider two concurrent activites of a function as an example. Let the input to the function be passed by means of a set of parameters. If any change in the input parameter during the activity one will affect the use of the particular data item in activity two, assuming that both the activities are sharing a common set of input parameters. This hazardous effect can be observed easily, if the activities are based on the conventional call by reference. Call by reference passes an address pointer to the function. If several activties of the function appear in parallel, then all of them share the same memory location. This common reference location for different activities may result in side effects. In order to prevent side effects, each activity of the function should have an independent copy of the parameters. This essentially means that each activtiy has to be provided with the actual values of of the input parameters rather then sharing a common memory location. This implementation is similar to the call by value in conventional languages.

Use of conventional language on data flow machine is possible. The techniques of data flow analysis can be used to construct the data flow graph of the high level language program. The data flow machine code is generated from the data flow graph. In addition to the construction Considerable importance has to be given to optimization, in addition to construction of data flow graph during language translation. The main motivation of optimization is to expose parallelism in the program. The optimization process follows the data flow analysis to perform the following:

1. The invarient computations in each interval is moved to expose parallelism.

2. The parallelism in loops are exposed by performing the transformations: loop unrolling, loop jamming, and loop unswitching.

The data flow analysis is complex because of unrestricted control flow. The analysis can be simplified in structured language where control flow is absolutely restricted. Special constructs can be used to expose parallelism explicitly.

CHAPTER V

APPLICATIVE LANGUAGES

A program can be considered as a representation of a function (f) applied to a set of data elements (x) which results a set of values y [y=f(x)]. The representation of function depends on the language under consideration. Depending upon the representation, the function must undergo many further transformations before it can be applied to its arguments. Procedural languages exhibiting explicit sequencing constraints, based on the conventional architectures. In procedural languages a procedure can be used to represent a function, and use of the function at different parts of the program is implememted by means of call to the procedure.

The properties of the programming language are characterized by its syntactic and semantic specifications. Syntax specification specifies the class of strings representing a program. Semantics concerns with the various objects used in the program. The objects are of two types; viz., objects internal to the language and objects external to the language (68). The term object refers to the external objects unless otherwise specified. From the point of view of conventional architectures, an object defines a storage location. The object can be a value rather than a storage location, since a value oriented objects are suitable for data flow architectures.

If a function can be evaluated by means of physical substitution of its arguments, then it is called substitutive function. A macro call, function subprogram in FORTRAN are close examples of a practical implementation of substitutive functions. The function evaluation in a procedural language are not 'pure' since they are not substitutive. This section is concerned about the languages which are value oriented and employs substitutive function evaluation technique.

Languages based on functional application on the values are called applicative languages and the operators represent the function (24, 43, 44). By means of applicative language one can specify what results are required, instead of specifying how to compute the result. A definitional language is a value oriented language and based on definitions. The function evaluation in a definitional language may or may not be substitutive. Best known pure function evaluation techniques are based on the lambda calculus and combinatory logic, an extension of lambda calculus. In genaral, a functional evaluation in a functional programming language can involve an unlimited number of arguments. For all practical purposes, a subset of functional language can be choosen involving functional application not more than two arguments. Such a language is termed as reduction language (63).

Definitional Language

As mentioned earlier, a definitional language is a value oriented language. An identifier holding a value is defined by means of definition statement. It is obvious that value held by an identifier is unique. Therefore, it is necessary to define a scope to the identifier. The above mentioned facts can be condensed as a rule, known as single assignment rule, which characterizes the definitional language. The single assignment rule can be stated as follows:

An identifier used on the right hand side of an assignment statement can not appear on the left hand side with in its scope.

The program representation is a procedure, which includes definitions and problem statements. There is a procedure statement to mark the beginning of the program and a procedure end statement to mark the end of program. A definition is assumed to appear prior to its use with in its scope. This is not very critical, but enables the programs to be well structured, free from definition ambiguties and fewer passes during compilation stage. The definition statement are of three types:

1. Function definition: Defines a function of given name representing a procedure. The values can be passed in or out of the function definition only through a set of parameters.

2. Expresion definition: A direct value assignment to an identifier. It is a single statement by means of which an identifier is assigened by evaluation of expression.

3. Count definition: A count definition is used to define an identifier as a counter. The operation on the count identifier can be either incremental or decremental.

Within a procedure a scope can be defined by means of scope begin and scope end statements. The conditional and iteration constructs can be implemented using the conventional syntax. An acyclic directed graph can be used to implement structures as discussed previously. The value of a structure, A, and selector, s, can be denoted by A(s), and can be used in the program like any other identifier. During the code generation this is implemented by means of a select structure operation (or select actor). Other structure operations can be implemented by using the following syntax.

1. mod(structure,selector,expr): To modify the structure definition of a given structure and selector.
2. create(structure,selector,expr): To create a structure definition in a given structure and selector.
3. append(structure,selector,expr): To append a structure definition to a given structure and selector.

Structure operations always result in a new structure, retaining the old structure on which the operation performed intact. This facilitates the use of same structure for various concurrent activities with out any side effect. It can be noted that all structure operations are implemented in a form of simple functions. A grammar for a definitional language is given below in BNF.

```
< Procedure >   ::= < proc stmt >< stmt > . . . < proc end >
< proc stmt >   ::= Procedure < parameters >
< stmt >        ::= < def stmt >|< cond stmt >|< iter stmt>
< def stmt >    ::= < assignmrnt >|< func def >
< assignment >  ::= < identifier > := < identifier >|< expr >
< expr >        ::= < value >< operand > . . . < value >
< value >       ::= < identifier >|< func name >|< struc func >
< struc func >  ::= < opr >< struc id >< selector >< expr >
< opr >         ::= create | append | modify
< struc id >    ::= < identifier >
< cond stmt >   ::= if < cond > then < expr > else < expr >
< iter stmt >   ::= iter < cond >< expr >
< func def >    ::= Function < func name >< procedure >
```

Compilation Techniques

The intermediate representation of the program is created during initial phases of compilation. The intermediate form is a simple structure consisting of two information fields and a link field to link all the entries of the intermediate form. The two information fields represent the type field and tree pointer field respectively. The type denotes the type of statement and the tree pointer points to the tree structure of the statement if available, otherwise it is set to zero (i.e., to indicate that the tree is not available). The data flow graph construction is straight forward from the intermediate form with out any complex analysis. A stack of definition lists is maintained and the list on the top of the stack corresponds to the current active scope. A data flow graph is constructed for each of the procedure independently.

Various steps involved in the construction of a data flow graph for a procedure is shown in Figure 35. A list of function defintions is maintained which indicates the location of the corresponding data flow graph. A reference to the function name implies function evaluation with the given set of parameters. The function reference can be implemented by following different methods.

1. A simple method is to substitute the complete data flow graph of the corresponding function reference, instead of the function name. This method may not prove efficient under all circumstances. The following methods using apply actors can be prefered instead of substitution.

2. The same copy of the function used for several concurrent activities. The above method implies all the activities absolutly concurrent. But this method and following method implies a pipelined cocurrency of various activities. The second method is by means of maintaining a structure of various activities of the function as discussed in one of the eariler chapters.

3. The third method is to pass the location of the apply actor that activates the function. This distinguish the various concurrent activities and the results are passed to the respective distributers of the apply actors.

Functional Language

This discussion has progressed from conventional languages to the value oriented languages which are based on both values as objects and on function definitions. This can be further extended to languages that are based on pure

```
RESTRCTURE: PROCEDURE
  DO for each entry in intermediate form
     IF type = procedure or function then do
     Initialize stack
     Push the procedure or function identifier
          and its information
     END IF
     IF type = scope then do
     CASE
         :scope = begin:
         Increment stack pointer
         :scope = end:
         Decrement stack pointer
     END CASE
     END IF
     IF tree is avilable then do
     scan the leaves of tree
         DO for each leaves of tree
         Search for its avilability in defintion list
         IF available = true then adjust the node pointers
            ELSE error
         END DO
     Include the definitions in tree in defintion list
     END IF
  END DO
END RESTRUCTURE
```

Figure 35. Restructuring for Definitional Languages

function evaluation technique. As mentioned earlier a functional language is a lambda calculus language which has simple syntax and simple semantics. It is possible to arrive at different language representation by altering the syntax of the lambda calculus. SASL, LISP are examples of the languages based on lambda calculus. Lambda calculus is a language of pure function evaluation and it composed of lambda expressions. Variables are used as the building blocks of lambda expressions.

Lambda Notation

Church introduced a method of defining functions known as lambda notation, in which the symbol lambda plays a special part (68).

Example: A function $f(x) = ax^2+bx+c$ can be represented in lambda notation as follows:

$$f = \lambda x.\ ax^2+bx+c$$

A function definition generally consists of three parts:

1. The name of the function being defined (f)
2. The variable (x)
3. An assosiated form of the function which is an expression determines the value of the function.

Lambda Expression

A Lambda expression is a function repreesenting an expression using the lambda notation. The genaral form of a lambda expression is denoted by λ x. M. The notation λx. M means that a result of lambda expression M is composed by the abstraction of x from M. The variable between lambda and dot (".") is known as the bound variable. M is said to be the body of the lambda expression. The variables that are not bound in M are said to be free variables. The representations λxM and (λx(M)) give the same meaning as λ x. M.

Basic Rules of Lambda Calculus

Let F be a lambda expression of the form λ x. M. If there exisist a definition D for the bound variable then the function can be evaluated by substituting the definition to all the bound variables that occur in the body of the lambda expression. The representation of a lambda expression along with the definition of the bound variable can be denoted by (F D), termed as operator-operand combination. The definition D is known as auxilary definition for the lambda expression F. The function evaluation in lambda calculus (13, 68) is based on the following two basic rules:

1. Reduction rule.
2. Renaming rule.

Reduction Rule

The expression $\lambda x M\ D$ can be reduced by applying the operator $\lambda x M$ to the operand D. The definition D is substituted in all occurance of the bound variable x in M. The substitution is valid if and only if,

1. D does not contain a free variable that is bound in M.
2. M should not contain an inner lambda expression with the same bound variable that is bound in outer lambda expression. In order to overcome the naming conflicts the following renaming rule can be applied to simplify the reduction rule application.

Renaming Rule

In lambda expression $\lambda x M$ the bound variable x can be replaced by any any other variable i.e., the expression can be renamed from x to any other variable name. The renaming is legitimate if and only if the new variable does not occur in M either as a free variable. If the inner bound variable of the lambda expression is the same as that of the outer bound variable then the renaming of outer variable should not consider the inner bound variables.

Operations in Lambda Calculus

A lambda expression of the form $\lambda x A$ represents a function of one argument. If the body A represents a lambda expression of $\lambda y B$ then the lambda expression represents a function of two arguments. Extending further lambda expression can be genaralized to represent a function of n arguments. If D represents an auxilary definition of the lambda expression $\lambda x A$ then D is the domain of the function and the set of all possible values of ($\lambda x A$ D) represents the range of the function. In arithmetic operations the domain and range are restricted to a defined class of numbers.

In logical operations the domain and range are restricted to a set of two elemants representing values of true and false, respectively. A conditional statement which is most common in all programming languages, selects one of the two specified alternatives depending on the logical value of the condition. In lambda calculus such a conditional statement can be regarded as a function of two arguments, which selects a list of two elements. The list of two elements specifies the required alternatives for the condition. A true value from the function selects the first element of the list and the false selects the second element of the list. The logical true and false can be implemented by a two argument functions $\lambda t \lambda f t$ and $\lambda t \lambda f f$ respectively. Consider a list (F1,F2), where the elements F1 and F2 are two functions. Applying true function to the list:

$\lambda t \lambda f t$(F1 F2) = λf F1(F2) = F1

Applying false function to the list:

$\lambda t \lambda f f$(F1 F2) = $\lambda f f$(F2) = F2

Another important use of logical functions is that they can serve as a list selectors. This aspect is already illustrated in representing the conditional construct in lambda calculus. This can be extended for a general list representation, which can be used for defining data structures. By operating a truth function on a list it is possible to select the the first element of the list. A false function on a list selects all the the elements except the first. Therefore, a successive false and true on a list selects the second element of the list. This can be generalised as i successive false followed by a single true function operation selects the (i+1)th element of the list.

A list can be represented as a function of one argument. The list function can be operated on the list operators, represented by true and/or false functions. This operation is much simpler than the list operations on the list as discussed earlier. The notation for the list representation is illustrated in the following example:

a0 = λl.(la0)ϕ where ϕ is a null list.

a0,a1 = λl.((la0) λl.((la1)ϕ))

= λl. la0 a1

The list operations on a list of two elements are illustrated as follows. The first example indicates a list operation to select the first element of the list and the second example illustrates the operation to select the second element of the list.

Example 1: (a0, a1) T

list operated on a true function operator T.

((λx (xa0) λx (xa1) ϕ) λtλ f t)

((λt (λf t)a0) λ x x a1) ϕ

a0

Example 2: (a0, a1) FT

a false and a true operator FT.

$(\lambda x\ (xa0)\ \lambda x\ (xa1)\ \phi\ \lambda t\ \lambda f\ (fT))$

$(((\lambda t\ \lambda f\ fT)\ a0)\ \lambda x\ xa1\ \phi)$

$(\lambda t\ \ (\lambda f\ f\ t\ \lambda f\ t)\ a0)\ \lambda x\ \ xa1\ \phi$

$(\lambda x\ \ xa1\ \phi\ )\ \lambda t\ \ \ f\ \ t$

$(\lambda t\ \ ft\ a1\ \phi)$

a1

Practical Application

In order to use the lambda calculus language conveniently, it is necessary to define a syntax that corresponds to lambda expressions. A lambda expression of the form $\lambda x.M$ represents a function and if D defines the arguments of the function, then the lambda expression is represented by using the auxilary definition i.e λ x.MD . It is possible to arrive at different types of syntax for this representation and some of them are illustrated as follows:

1. M where x = D.
2. let x = D in M.
3. substitute (x, D) M.

The interpretation technique varies for different notation. In the first type of notation shown above, the arguments are specified after the expression, where as in others it is viceversa.

The conditional statement can be formed by applying true or false functions on a list of two elements and the following syntax can be used:

```
if C then T
     else F
```

This is equivalent to (if C) (λx.T, λx.F), where 'if' is a functional operation results in a true or false function. A true operation on the list yields the first element T of the list and a false operation yields the second element of the list. Similar to 'if' function, functions such as case, head, tail, first, rest, . . . functions,, and recursive functions can be defined. It is possible to come up with unlimited number of such functions to perform various operations. A complete discussion of varying types of syntax for such functions are beyond the scope of this thesis.

Mego (41, 42) has proposed an architecture, which executes the functional language. The computational model is based on the reduction rule of lambda calculus. The machine has two types of cells, interconnected in a regular manner. One type of cell is known as leaf cell, holds the program. The other type of cells are interconnected in the form of binary tree. The reducible expressions are distributed to the tree cells and computation is performed on the basis of reduction rule. The result obtained at the root node will be placed at the appropriate leaf cell, where the corresponding reducible expression is present. Mego (42) has proved the following propositions:

1. Every tree cell other than the root cell is connected to its parent cell either by two branchs of two different areas or by a single branch of one area.

2. Each tree cell holds one, two, three or four nodes each belonging to a different area.

In addition, the following features of this machine illustrates that the architecture with functional programming cababilty can exploit the advantages of VLSI implementation.

1. Since the computation is based on the reduction rule, the result due to application is placed at the leaf cell where the reducible expression was held. Therefore there is no need for additional cells to save the results.

2. The reducible expressions can be reduced independently in the tree cells. Therefore there is no need for interconnection between the tree cells of the same level.

3. Since there is no interconnection between the tree cells of the same level, the tree cell interconnections are simple and regular - an important requirement for the effecient VLSI implementation.

Definitional language is a value oriented language based on single assignment rule. Every program can be represented as a procedure. There are three different types of defintion statements: function definition, expression definition, and count definition. In addition, statements such as mod, create, and select are used to perform the structure operations. The data flow graph can be constructed directly without much complex analysis. Definitional language can be effectively used as a high level language on data driven machines.

Functional language is based on the functional application properties of lambda calculus. The importance of the language on demand driven data flow machines is due to the reduction rule of lambda calculus. These languages are not best suitable for sequential nature of the conventional architecture. Attempts have been made (13, 68) to come up with lambda calculus based machines, which served as a model for the development of compilers of such languages for conventional architecure (for eg. LISP). In the case of data flow machines, the execution of an instruction based on the availability of the operands with out any sequencing constraint. This property enables the data flow machines best suitable for languages based on lambda calculus. A typical practical example is the Treleaven's reduction machine(63). The data flow architectures based on the functional languages can be regarded as high level language architectures. The main reason being that the the functional language could be the direct meachine representation, unlike the other languages. The cellular architecture proposed by Mego can demonstrate that the functional language based models can be implemented by VLSI. As a result of VLSI implementation, it is possible to develop highly parallel architectures at low cost, without much complexity in hardware.

CHAPTER VI

SUMMARY, CONCLUSIONS, AND SUGGESTED FUTURE WORK

A survey of data flow architecture was presented. Main attention was focused on the following architecturs: Dennis data flow architecture, the TI distributed data flow machine, Arvind data flow machine, Manchester data flow machine and Treleven's reduction machine. Aspects of data flow program representation and notations were presented. Starting from the data flow architecture, use of conventional langages on these machines were considered. It was found that the compilation techniques can be simplified by using structured languages. The importance lies in the restriction of the control flow, which leads to simplification of data flow analysis. Possibilites of language extentions were considered to expose the parallelism of the problem.

The next consideration was given to the applicative languages. Definitional languages are a class of applicative languages, known as value oriented language. The aspects of definitional language and language specifications were presented. The characterstics of the definitional language makes it better suitable for data flow machines than the procedural languages. Functional language is another class of applicative language, based on the functional evaluation techniques of lamdbda calculus. Following the basic aspects of lambda calculus, the correspondence between functional and lambda calculus were presented.

The current developments in VLSI necessiates the consideration to be given tolanguage first rather than considering the language design from the architecture point view. The development in microelectronics technology enables to take advantage of the inherent parallel specification of the functional language. The main advantages of VLSI are (45):

1. Simple and regular interconnection leading to cheap implementation.
2. High density leading to high performence and there is no over head for support components.

These advantages enables VLSI highly suitable for hardware implementation of parallel algorithms. Discussions on implementing matrix algorithms and hierarchical organization can be found in (45). Current research is to develop data flow machines having hundreds of processors with a functional language as its machine language. This results in the data flow machines being high level machines. Turner has suggested a reduction machine, which has some features of hierarchical organization. The reduction machine proposed by Treleven seems to be suitable for VLSI implementation (63). Mego has proposed a tree structured architecture that can execute reduction language and it can be implemented by VLSI (41, 42). The future studies of functional languages must go in parallel with the study of VLSI. The aspects of functional langages can be elaborated to a powerful language specification and their hardware implemetation is to be considered.

The material presented here can be extended for future studies on simulation of data flow language and architechture, constructional aspects of data flow interpreters and compilers. The material can serve as background for the study of applications of data flow processors in the fields of fault tolerant computer architecture and use of data flow processors as special purpose machines in the fields such as signal processing, artificial intelligence (18).

BIBILOGRAPHY

(1) Ackerman, W.B. "A Structure Processing Facility for Data Flow Computers." Proceedings of International Conference on Parrallel Processing, 1978, 166-172.

(2) ____________. "Data Flow Languages." Proceedings of AFIPS Conference, 1979, 1087-1095.

(3) Aho, A. V and Ullman, Jeffrey. D The Theory of Parsing, Translation and Compiling. Vol(1,2). New Jersey: Prentice-Hall, 1973.

(4) ____________. Priciples of Compiler Design. California: Addison-Wesley, 1978.

(5) Allan, Stephen J. and Oldehoeft, Arther E. "A Flow Analysis Procedure for the Translation of High Level Languge to a Data Flow Language." Proceedings of International Conference on Parallel Processing, 1979, 26-34.

(6) Allen, F. E. and Cocke J. "A Program Data Flow Analysis Procedure." Communications of ACM, Vol 19, Mar 1976, 137-149.

(7) Arvind and Gostelow, K. P. "A Computer Capable of Exchanging Processor for Time." Information Processing 77, 1977, 849-853.

(8) Babich, A. and Jazayari, M. "The Method of Attributes for Data Flow Analysis." Acta Informica, 1978, 245-272.

(9) Backus, J.W. "Can Programming be Liberated from Von Newann Style? A Functional Style and its Algēbra of Programs." Communications of ACM, Vol 21, Aug 1978, 613-647.

(10) Berkling K. J. "Computing Machines Based on Tree Structures." IEEE Transactions on Computer, Vol 20, Apr 1971, 404-418.

(11) ____________. "Reduction Languages for Reduction Machine." Proceedings of Second International Symposium on Computer

Architecture, 1975, 133-140.

(12) Burge, William H. "Combinatory Programming and Combinatorial Analysis." IBM Research and Development, Vol 16, No 5, Sept 1972.

(13) ______________. Recursive Programming Techniques. California: Addison-Wesley, 1978.

(14) Caswell, H. L. "Superconducting Josephens on Computers." Proceedings of IEEE COMPON 1979, Feb 1979, 154-157.

(15) Cocke, John and Schwartz J. T. Programming Languages and their Compilers. Preliminary notes, New York University, Apr 1970.

(16) Davis, A. L. "The Architecture and System Model of DDM1: A Recursively Structured Data Driven Machine." Annual Symposium on Computer Architecture, 1978, 210-215.

(17) Davis, A. L. "A Data Flow Evaluation System Based on Concepts of Recursive Locality." Proccedings of AFIPS conference, 1979, 1079-1086.

(18) Davis, A. L and Stanek, J. A. "A Computer Music Synthesis Study on Tree Structured Data Driven Machine." Proceedings IEEE Compcon 80, Feb 1980, 188-201.

(19) Dennis, Jack B. "Packet Communication Architecture." Proceedings of Sagamore Computer Conference on Parallel Processing, 1975, 224-229.

(20) ______________. "First Version of Data Flow Procedure Language." Lecture Notes on Computer Science, Vol 19, 1974, 362-376.

(21) Dennis, Jack B. and Misunas David P. "A Preliminary Architecture for a Basic Data Flow Processor." Annual Symposium on Computer Architechture, 1975, 126-132.

(22) Dennis, Jack B. and Weng, Ken K.S. "An Abstract Implementation for Concurrent Computation with Streams." Proceedings of International Conference on Parallel Processing, 1979, 35-45.

(23) Fanell, Edward P., Ghani, Noordin and Treleaven, Philip C. "Concurrent Computer Architecture and Ring Based Implementation, Data Driven Machine." Annual Symposium on Computer Architecture, 1979, 1-12.

(24) Friedman, Daniel P. and Wise, David S. "Aspects of Applicative Programming for Parallel Processing." _IEEE Transactions on Computers,_ Vol 27, Apr 1978, 289-296.

(25) Gelly, O. et.al. "LAU-System Software: A High Level Data Driven Language for Parallel Programming." _Proceedings of International Conference on Parallel Processing,_ 1976, P255.

(26) Gostelow Kim, P. and Thomas Robert "A View of Data Flow." _Proceedings of AFIPS Conference,_ 1979, 629-636.

(27) Gurd, J and Watson, I. "A Multilayered Data Flow Architecture." _Proceedings of International Conference on Parallel Processing,_ 1977, 94.

(28) ______________. "Data Driven System for High Speed Parallel Computing, Part 1,2." _Computer Design,_ July 1980, 97-106.

(29) Hecht, Matthew S. _Flow Analysis of Computer Programs._ Holland: North Holland, 1977.

(30) Hecht, Matthew S. and Ullman, Jeffrey D. "A Simple Algorithm for Global Data Flow Analysis Problems." _S.I.A.M Journal on Computing,_ Vol 4, Dec 1975, 519-532.

(31) Hicking Anne, "Translation of some Sequential Language Constracts into Code for Parallel Interpreter." M.Sc Dissertation, University of Nottinghan, Mathematics Dept., Jan 1980.

(32) Hindley J. R and et. al. _Introduction to Combinatory Logic._ London: Cambridge University Press, 1972.

(33) Ian Watson and John Gurd, "A Prototype Data Flow Computer with Token Labelling." _Proceedings of AFIPS Conference,_ 1979, 623-628.

(34) Jensen, John. C. _"Basic Program Representation in Texas Instruments Data Flow Test Bed Compiler."_ Texas Instruments Incorporated, Jan 1980.

(35) Johnson, D and et.al. "Automatic Partition of Program in Mutiprocessor System." _Proceedings IEEE Compcon 80,_ Feb 1980, 175-178.

(36) Kan J. B. and Ullman J. D. "Global Data Flow Analysis and Iterative Algorithms." _Journal of ACM,_ Vol 23, Jan 1976, 158-171.

(37) Karp R. M. and Miller R. E. "Properties of Model for Parallel Computations: Determinacy, Termination, and Queuing." SIAM J. Applied Maths., Vol 11, Nov 1966, 1390-1411.

(38) Keller, R. M. and et.al. "Data Flow Concepts for Hardware Design." Proceedings IEEE Compcon 80, Feb 1980, 105-111.

(39) Kvcil, David J. "Parallel Processing of Ordinary Programs." Advances in Computer, Vol 15, 1976, 119-179.

(40) Landin P. J. "The Mechanical Evaluation of Expressions." Computer Journal, Vol 6, 1964, 308-320.

(41) Mago' Y. A. "A Cellular Computer Architecture for Functional Programming." IEEE Compcon 80, Feb 1980, 179-187.

(42) ____________. "A Network of Microprocessors to Execute Reduction Languages." International Journal of Computer and Information Sciences, Vol 8, no 5,6,1979,349-385.

(43) Maurer W. D. The Programmer's Introduction to LISP. New York: American Elsevier, 1972.

(44) Mccarthy, J. LISP 1.5 Programmer's Manual. Massachusetts: MIT Press, 1970.

(45) Mead C. A. and Conway L. A. Introduction to VLSI Systems, California: Addison-Wesley, 1979.

(46) Merrill Cornisn. "The TI Data Flow Architecture: The Power of Concurrency for Avionics." Texas Instruments Incorporated, 1979.

(47) Misunas, David P. "Structure Processing in a Data Flow Computer." Proceedings of Sagamore Computer Conference on Parallel Processing, 1975, 230-234.

(48) Oldehoeft, A. E and et.al. "Measurement of Parallelism in Computer Programs Through Analysis of Program Graphs." Presented at First European Conferance on Parallel and Distributed Processing, Feb 1979.

(49) Plas, A., et.al. "LAU-System Architecture: A Parallel Data Driven Processor Based on Single Assignment." Proceedings of International Conference on Parallel Processing, 1976, 293-302.

(50) Ratna Dhas, C. "Performance Evaluation of Feedback Data Flow Processor using Simulation," Dept. of Quantitative and Information Sciences, Western Illinois University, Illinois.

(51) __________. "Estimation of Intrensic Parallelism in High Level Programs using Data Flow Execution." Dept. of Quantitative and Information Sciences, Western Illinois University, Illinois.

(52) Rustin, Randel. "Design and Optimization of Compilers." Courant Computer Science Symposium 5, New Jersey: Prentice-Hall, 1971.

(53) __________. "Formal Semantics of Programming Languages." Courant Computer Science Symposium 2, New Jersey: Prentice-Hall, 1970.

(54) Riccelli, R.F. "The Design of Data Driven Prossing Element." Proceedings of International conference on Parallel Processing, 1978, 173-180.

(55) Rosen, Barry K. "Data Flow Analysis for Procedural Languages." Journal of the ACM, Vol 26, Apr 1979, 322-344.

(56) Rosen, Barry K. "High Level Data Flow Analysis." Communications of the ACM, Vol 20, Oct 1977, 712-724.

(57) Rumbaugh, James. "A Data Flow Multiprocessor." IEEE Transactions on Computers, Vol 26, Feb 1977, 138-146.

(58) Rumbaugh, James. "Data Flow Languages." Proceedings of Sagamore Computer Conference on Parallel Processing, 1975, 217-219.

(59) Rumbaugh, James. "A Data Flow Multiprocessor." Proceedings of Sagamore Computer Conference on Parallel Processing, 1975, 220-223.

(60) Lee, Shich-Yi. "A Survey of Data Flow Analysis using Interval and Iterative Analysis Techniques." M.S. Report, Department of Computing and Information Sciences, Oklahoma State University, July 1980.

(61) Sleep, M. R. "Applicative Languages, Data Flow and Pure Combinatory Code." Proceedings IEEE Compcon 80, Feb 1980, 112-115.

(62) Treleaven, Philip C.T. "Exploiting Program Concurrency in Computing Systems." Computer, Vol 12, Jan 1979, 42-50.

(63) Treleaven, Philip C. and Mole, Geoffrey F. "A Multiprocessor Reduction Machine for user Defined Reduction Language." Computing laboratory, University of Newcastle upon Tyne, Sept 1979.

(64) Treleaven, Philip C. "Program Evaluation Prespective." Computing Laboratory, University of Newcastle upon Tyne, July 1979.

(65) Treleaven, Philip C., et.al. "Data Driven and Demand Driven Computer Architecture." Computing Laboratory, University of Newcastle upon Tyne, July 1980.

(66) Turner, D. A. SASL Language Manual. Computer Laboratory, University of Kent, Dec 1976.

(67) Turner, D. A. "A new Implementation Technique for Applicative Languages." Software Practice and Experience, Vol 9, 1979, 31-49.

(68) Wegner, D. Programming Languages, Information Structure and Machine Organization. New York: Mcgraw-Hill, 1968.

(69) William Sauber. "A Data Flow Architecture Implementation." Texas Instruments Incorporated, 1980.

APPENDIX A

SUMMARY OF DATA FLOW MACHINES

TABLE III

SUMMARY OF DATA FLOW MACHINES

Name	Type	Host	Language	Features
TI DDP	Data driven	TI 990/10	ASC FORTRAN	Simple hardware expansion. No Special Structure processing; No token labelling;
Dennis	Data driven	-	Data flow language	Modular; Structure processing units can be usedfor structure operations; No token labelling
Arvind	Data driven	-	Data flow language	Tokens are labelled; Highly parallel and has LSI implementation.
Manchester	Data driven	PDP 11	Data flow language	Tokens are labelled; Highly parallel and has LSI implementation.
Reduction machine	Demand driven	-	Reduction language	High level architecture; Highly parallel and has LSI implementation.

APPENDIX B

SYMBOLIC REPRESENTATION FOR DATA FLOW PROGRAM

Instruction Examples

field 1	field 2	field 3
DEF	I	0
DEF	@I	false
EVAL	I	K + P * Z
EVAL	@X	(P+Q)=(R+S)
TG	@I	I1 → I2
FG	@I	I1 → I2
MERGE	@I1	I1,I2 → I
SWITCH	@I	I → I1,I2
APPLY	&FUN	P1,P2,P3,@I
TG	@I	RETURN(Z)
FG	@I	STOP
SWITCH	@I	I → I1, RETURN(I)

Example 1

```
INPUT (x,y,n)
   i = 0
   do while(i>n)
   i = i + 1
   z(i) = x(i) + y(i)
   end
OUTPUT (z)
```

A graphical representation for the example program is shown in Figure 36. The set of instruction for the symbolic representation of the program is as follows:

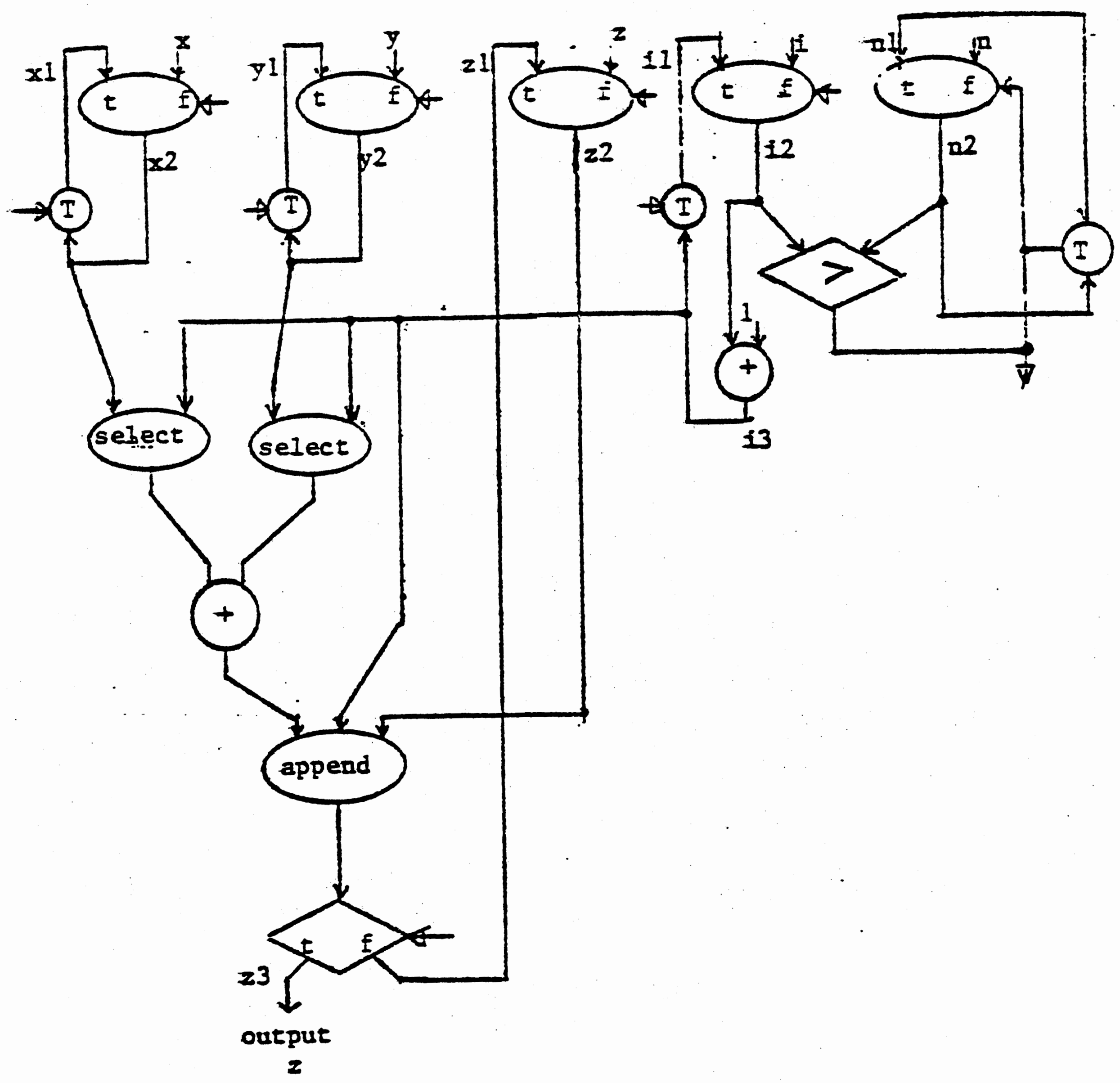


Figure 36. Graphical Representation for Example-1

```
DEF      I          0
DEF      @I         false
MERGE    @i         n,n1  → n2
MERGE    @i         i,i1  → i2
MERGE    @i         y,y1  → y2
MERGE    @i         x,x1  → x2
MERGE    @i         z,z1  → z2
EVAL     @i         i2 → n2
TG       @i         n2 →  n1
EVAL     i3         i2 + 1
TG       @i         i3 →  i1
TG       @i         x2 →  x1
TG       @i         y2 →  y1
EVAL     z2(i3)     x2(i3) + y2(i3)
SWITCH   @i         z2  → z1, RETURN(z2)
```

Example 2

```
if (y > 1) then x = x + 10
            else x = x + 100
```

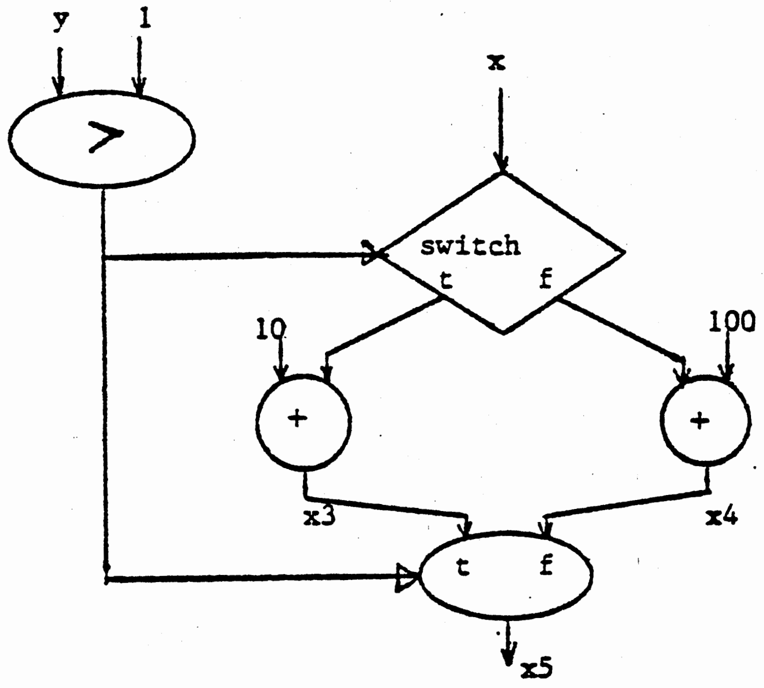


Figure 37. Graphical Representation for Example-2

The instructions for example-2 are as follows:

```
EVAL      @y     y > 1
SWITCH    @y     x  → x1,x2
EVAL      x3     x1 + 10
EVAL      x4     x2 + 100
```

APPENDIX C

GLOSSARY

Activity
: The execution instance of a function is called activity.

Activity name
: Unique identification given to an activity.

Actor
: Data flow operator.

Arbitration network
: Receives operation packets from the instruction cells, present in the memory, and sends them to the appropriate operation unit.

Assignment function
: An activity which maps the logical activity names (program locality) to physical processor address (physical locality).

Associative memory
: This is also known as content addressable memory (CAM). The memory is scaned by means of a key associated with the information (or the information itself) rather by means of a memory address. A hardware or software hashing techniques may be necessary for memory storage and retrievel.

Association table
: This is similar to associative memory.

Asynchrony
: Operaters executed, if and only if the operands are available at the input arcs of the data flow graph.

Balanced tree
: Tree structure balanced with respect to the root node. eg. AVL tree, B_tree, B*_tree

Cache
: Cache is a scrach pad random access memory usually semiconductor type, holds the information that are most often required by the processor. Other information about the program is kept in a slower main memory. The information is passed to the cache, based on certain policies whenever it is required.

Control network
: A network which handles control packets. The network consists of arbitration and distribution units.

Data driven
: The class of data flow in which the instructions are executed when all the operands required by the

instruction are available.

Data flow structure
Structured data residing on conventional memory.

Data packets
Instruction cells containg data values is known as data packets.

Decision unit
It is a hard ware unit performs boolean operations and gives the result in the form of control packets.

Demand driven
The class of data flow, in which the instructions are executed only if the result of the instruction is demanded.

Distribution network
Receives results from the operation unit in the form of data packets and place them in the instruction cells, present in the memory.

Dynamic parallelism
The same function replicated and the resuling functions exhibit the property of inherent parallelism, with out side effects.
e.g. In an iteration the function could be replicated as many times as the number of iterations if the resulting functions are free from side effects.

Fired
When tokens are present at the input arcs of a data flow graph the node is enabled and the operands are removed from the input arcs. i.e. , the operands are fired.

Function
A segment or an instruction which is independent.

ID
Irvine Data flow - data flow language.

Instance
Different stages encountered in data flow machine.

Instruction cell
The memory is organized into instruction cells. Each instruction cell consists of three or more registers to hold the data and operator.

Instruction packet
A packet containing a data flow instruction is called instruction packet. There are three types of

instruction packets viz., operation packet, control packet, and memory command packet.

Link

The program in an elementary data flow language is a directed graph in which the nodes are operators. The nodes are interconnected by means of links.

Locality

Working set or the working area in the memory i.e. , Physical locality or program locality.

Locality of effect

Means that the instruction donot have unnecessary for reaching data dependencies.

Operation packet

Operation packet is one of the types of instruction packet, that are handled by the operation unit. Each operation packet contains one or more data packets (operands) and an operator (instruction).

Operator

Operators are the data flow instructions. Refer link.

Packet

The information, may be either data or operator, sent from one unit to other unit of a data flow machine is termed as a packet.

Scope

A variable can be assumed to have a definite scope, i.e. , a region of a program in which the variable is active.

Selector

Used in the representation of data flow structures - an integer or a string. The structure node is represented as ½selector:value¼.

Selector vector

String or integer used as selectors and ordered values are stored in a contiguous memory words.

Shift register

A register which transfers bit information serially, is called shift register.

Side effect

If the concurrent computations are interdependent then it is called side effect. A defined function always free from side effect.

Single assignment

The single assignment rule is that a name may appear on the left hand side of assignment(definition) only once with in its scope.

Speed
Switching time and delays in the fabricated devices influence the computing time.

State transition table
A table describing the state diagram of a sequential logic (or a finite state machine) is called state transition table.

Static parallelism
Two or more dissimilar functions exhibiting the property of inherent parallelism.

Stream
Stream is a sequence of values of the same type, each value being passed successively.

Token
Refers to the presence of data item (value) in an arc (link) of a data flow graph.

Unfolding interpreter
Data flow interpreter seperates the portions of the program, having the property of the inherent parallelism.

Update controller
Every data flow instruction at the machine level is associated with a counter. The counter is preset with the number of operands that are required by the instruction before it could be executed. The counter is decremented by one whenever the instruction receives a result packet. The hard ware unit which presets and updates the counter is known as update controller.

Value
Data flow value. Refer selector.

VLSI
Very large scale integration.

VITA 2

Krishnan Venkataraman

Candidate for the Degree of

Master of Science

Thesis: EFFECT OF DATA FLOW ARCHITECTURE ON PROGRAMMING LANGUAGE DESIGN.

Major Field: Computing and Information Sciences.

Biographical:

Personal Data: Born Aug 10, 1955, Tamilnadu, India, the son of Mr. and Mrs. S. Krishnan.

Education: Graduated from University College of Engineering, Bangalore, India in July 1978; received Bachelor of Engineering degree from Bangalore University in 1978. Completed requirements for Master of Science degree at Oklahoma State University in July 1981.

Professional Experience: Graduate Research Assistant in Department of Bio-chemistry, Oklahoma State University (1979-81).