4-2-1985

Simulation of a data flow computer

Carol Torsone

Rochester Institute of Technology
School of Computer Science and Technology

SIMULATION OF A DATA FLOW COMPUTER

by
Carol M. Torsone

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by: John L. Ellis
John L. Ellis, Ph.D.

Lawrence A. Coon
Lawrence A. Coon, Ph.D.

Peter H. Lutz
Peter H. Lutz, Ph.D.

April 2, 1985

PERMISSION TO REPRODUCE

Title of Thesis: Simulation of a Data Flow Computer

I, Carol Torsone, hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Carol M. Tarsone

April 2, 1985

ABSTRACT

A data flow computer is a highly concurrent and asynchronous multiprocessor due to its fundamentally new architecture. It has no program counter and is not sequential. Instructions execute whenever their operands are available to them. Because of this data-activated instruction execution, multiple instructions can execute concurrently.

The project for this thesis was the simulation of a data flow computer. A graph language and machine language were defined; then a simulator was written which reads and executes a machine language program in the asynchronous and concurrent manner of a data flow computer.

KEYWORDS: data flow, concurrent, asynchronous, multiprocessor, data-activated.

Table of Contents

FIGURES

TABLES

ACKNOWLEDGMENTS

I would like to recognize the members of my committee for their assistance in the completion of my thesis. Their direction, help and ideas were invaluable throughout the project.

I wish to thank Dr. Lawrence Coon for sharing his interest in data flow computers, pointing me in the direction of the research in this area, and lending his encouragement, interest and ideas.

I also owe a debt of gratitude to Dr. Peter Lutz for his ideas and assistance in the area of concurrent programming in the Euclid language.

My chairman, Dr. John Ellis, deserves special thanks for the suggestions he has made, and also for his time and efforts in guiding me from beginning to end through my thesis project.

CHAPTER 1

Introduction

The computers of today still share fundamental properties of the von Neumann design, which are:

(1) A central processing unit

(2) A global, updatable memory

(3) A single instruction counter which causes a sequential centralized control of computation.

(4) A connecting tube that can transmit a single word or address between the CPU and memory (the "von Neumann bottleneck") [Backus 1978].

Computing applications of the future will require billions of operations a second, and the von Newmann concept has caused problems in attaining these speeds for the following reasons.

The first is the bottleneck in the physical configuration of the machines, as described above.

The second is the fact that conventional programming languages used today reflect the same architectural principles as the computers:

(1) A variable in a programming language mirrors the concept of memory storage cells.

(2) An assignment statement is similar to the idea of a processing unit which performs state changes in the storage through the "word-at-a-time" tube.

(3) The statements used for control flow (go to, if, do, call) follow the concept of the instruction counter.

These "von Newman" languages are complex and weak, due to a defect at their most basic level: They were designed from the inside out (they closely reflect the behavior of the underlying architecture) instead of from the outside in (from the programmer's point of view, permitting the natural expression of the problem) [Backus 1978] [Myers 1982].

The third is that, in an effort to achieve greater speed and computing power, the concept of parallelism has evolved. However, it has been largely restricted to the switching of a processor among separate processes, and the use of multiple processors (programmer-specified decomposition of a program into parallel instruction or data streams to be processed by separate processors). The first approach doesn't buy much in terms of overall speed, and the latter presents a non-trivial task to the programmer for which he is given few tools, and still does not offer relief to the problem of memory interference on a system which has multiple processors and shared memory

[Myers 1982].

Over the past few years, a number of novel computer architectures have been proposed based on "naturally" parallel organizations. They arose from a desire to utilize asynchronous concurrency to increase computer performance, to exploit VLSI in the design of the computer, and from the need for new "very high level" programming languages, which are based on principles which conflict with those of the von Newmann architecture [Treleaven, Brownbridge and Hopkins, 1982]. They also arose from the feeling that necessary improvements would only come about through new and radical approaches to the basic design of computers.

One of these areas of research is in data flow (or data driven) computers, which was pioneered by Jack Dennis at MIT. Data flow does away with the basic properties of the von Newmann design by:

(1) Eliminating the idea of the instruction counter, sequential instruction execution, and control flow. Data flow computers are data driven; when the operands to an instruction are available, the operation is executed. As a consequence, many instructions may be available for execution at once.

(2) Eliminating the concept of memory for storing variables. Data values move from one instruction to the next as the

program executes.

(3) Taking advantage of parallelism within a program without explicit directions from the programmer. Highly concurrent computation is a natural consequence of the data flow concept.

Data flow programs are represented by connecting instructions (nodes) in a directed graph; i.e. one instruction's output is another instruction's input. The order of execution is controlled by the availability and flow of data among the instructions, not an instruction counter. A data flow graph is, in effect, the base language or machine code for a data flow computer.

Data flow processors are stored-program computers in which the stored program is a representation of data flow graphs. The machine itself is designed to recognize which of the instructions in its program memory are enabled, and all such instructions are dispatched to execution units as soon as resources are available. Any instructions in the program which have their data available can execute concurrently [Agerwala and Arvind 1982].

Language-based computer design can insure the programmability of a radical architecture, where the computer is a hardware interpreter for a specific base language. Since data flow languages allow the expression of concurrency of program

execution on a large scale, a data flow computer will be able to support great concurrency and achieve a significant increase in performance.

In this thesis, the various architectures which have been designed to support data flow graphs are investigated, as well as the instruction-handling mechanisms used to build prototype data flow systems. Further, a data flow base language will be defined for a "generic" data flow machine architecture and this machine will be simulated in its execution of programs. Specific architectural features such as interconnections between processors, speed of communications lines, etc., will not be simulated as they are beyond the scope of this thesis.

Chapter two defines what a data flow graph or program is and describes basic operators commonly used. Also described are approaches and problems in handling data structures in a data flow program, and machine representation of data flow graph programs.

Chapter three describes the configuration of a data flow computer's resources and how these resources can be allocated to support execution of a program.

Chapter four describes the implementations of the major data flow models and prototypes described in the literature.

Chapter five contains a detailed description of the data flow model simulated for this thesis, as well as a definition of the graph language used, a mnemonic form of the graph language to be used as input into an assembler, and the machine language executed by the simulator.

Conclusions drawn after the project was completed are contained in Chapter six, i.e. an appraisal of the simulator. Also described are the next steps which could be taken and related topics which may be or are currently being investigated.

CHAPTER 2

Data Flow Program Organization

2.1. Data Flow Graphs

In this chapter, data flow graphs or programs will be described as they are generally viewed by those doing research in this area [Dennis 1975] [Davis and Keller 1982]. There are certain operators that are included in almost all graph representations. Even though they may have variations in name, pictorial representation, or slight variations in their use (e.g. number of input and output arcs), they are basically the same operator.

Elementary data flow programs are represented as directed graphs in which the nodes are operators (i.e. instructions). Nodes are connected by arcs along which elementary values or tokens can travel (an elementary data flow value is of type integer, real or string). An operator is enabled when tokens are present on all its input arcs. An enabled operator can then execute or fire at any time, provided no tokens are present on any output arc. The enabled operator removes the tokens from its input arcs, computes a value based on the input tokens or operands, and places this result token on its output arc.

A result may be sent to more than one destination by means of an operator which duplicates tokens: It removes a token from

its input arc and places copies of the input token on its output arcs.

A node marked with a constant value is assumed to regenerate that value as often as it is needed.

An example of a data flow graph is shown in Figure 2.1. This graph repeatedly computes the polynomial function f(x) = x ** 2 - 2 * x + 3 for a sequence of input tokens. Note that the two multiply operators are not connected by an arc, which implies there does not exist a data dependency between them. Therefore, these operators could execute concurrently. This type of concurrency is

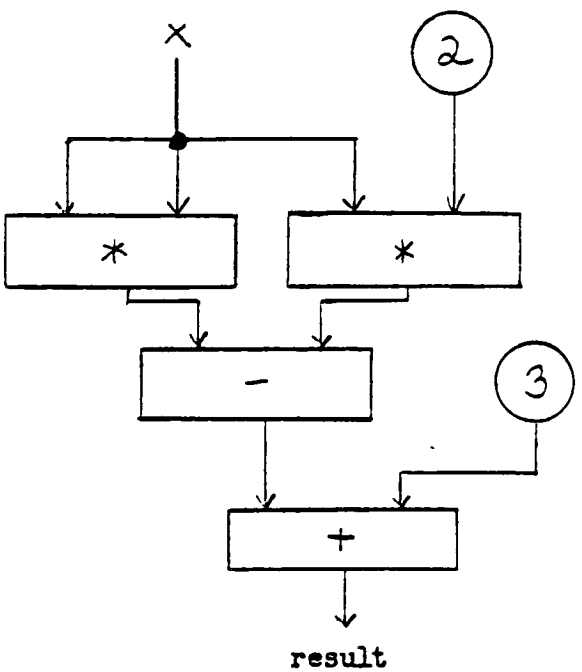


Figure 2.1 f(x) = x**2 - 2*x + 3

called horizontal or spacial. Temporal concurrency or pipelining corresponds to several generations of tokens moving through a graph, and is illustrated in Figure 2.2 [Davis and Keller 1982].

An operator is drawn as a box containing the name of the operation. Certain operators are distinguished by shape. The operator which duplicates tokens, as described above, is a tiny circle.

The merge, switch, and gate operators have both data and control inputs. The merge and switch operators (Figure 2.3) are used in conditionals and iterations. With a merge, a control token with the value true or false must first arrive on the horizontal input. The value of that token determines from which of the vertical data inputs the next token will be taken. Any token on the other input arc remains there until selected by a subsequent true or false token. A switch also waits for a token on the control input; its value of true or false determines the output arc to which the vertical token is passed.

A gate is an operator that either passes on or absorbs the input token depending on the value of a boolean control value (Figure 2.4). In the case of a "gate if true," the vertical data token is absorbed; then if the horizontal token had the value true the data value is put on the output arc, but if the control value were false the data token is absorbed and there is no output value. A "gate if false" operator works in a similar way,

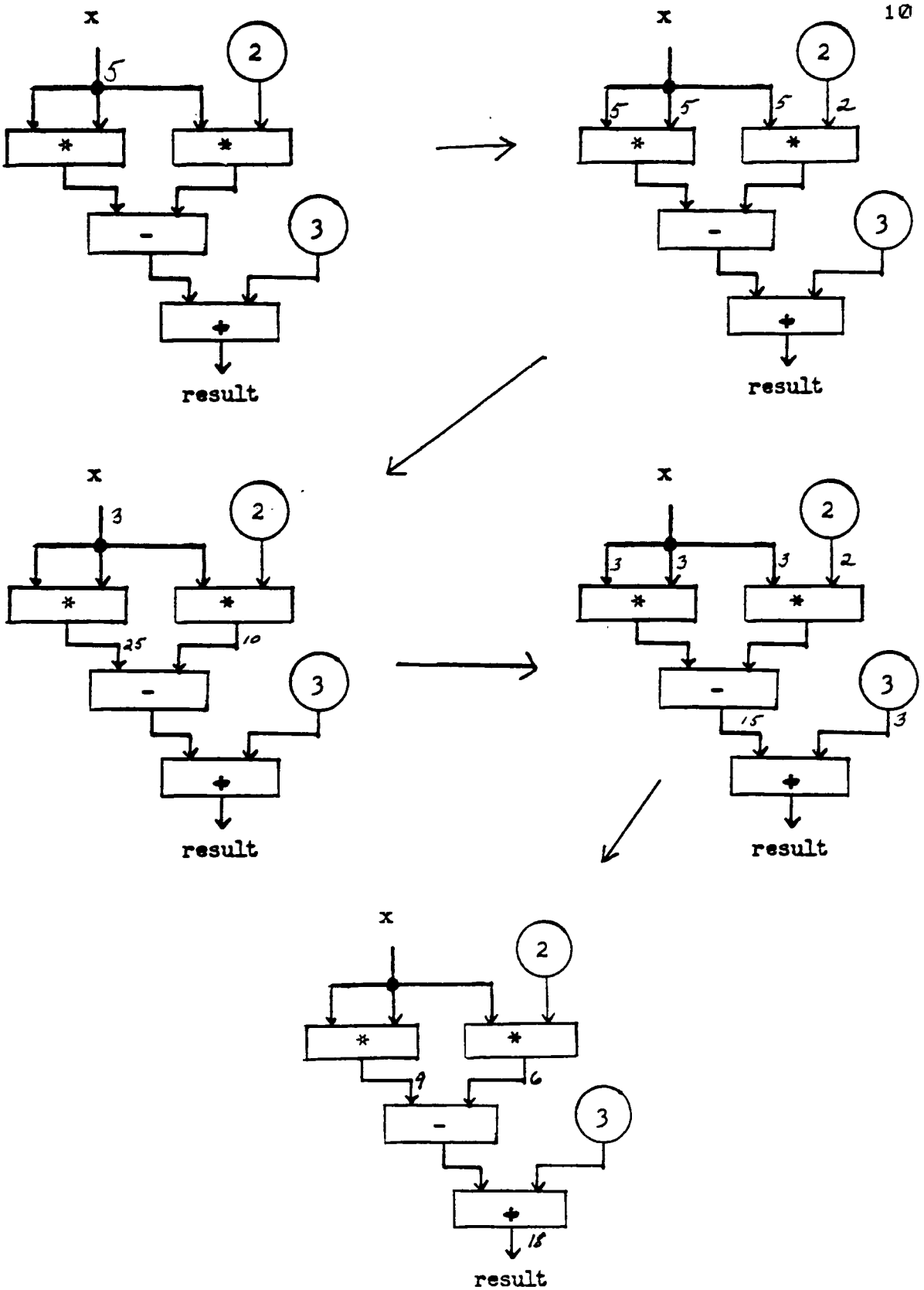


Figure 2.2 f(x) = x**2 - 2*x + 3

Figure 2.3 Merge and Switch Operators

Figure 2.4 Gate Operator

except that a false control value causes the data token to be put on the output arc.

A decider operator produces a true or false control result by applying its associated predicate to the data inputs (Figure 2.5). Typical predicates are equality, inequality, less than, etc.

A constant-producing node can contain a graph representing a function, and its use is similar to the way in which conventional programs use subroutines and procedures. The value of the token

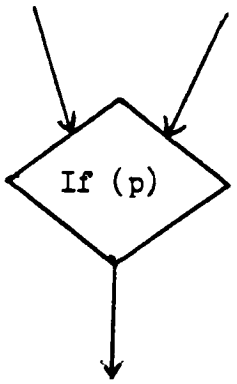


Figure 2.5 Decider Operator

produced by the node is the definition of the function. It is used as an input token to an apply operator along with other input tokens which carry parameter values for the function. This is illustrated in Figure 2.6, where inputs to the apply operator are the data value of r (radius) and the graph for the function to compute volume of a sphere.

2.2. Data Structures

The token model of an data flow program shows an elementary data value being completely swallowed up as input to an operation, and a new value being produced as output. To be consistent with this and to maintain the clear semantics of data flow, a data structure such as an array must be treated as a single object or token rather than a collection of elements. The entire structure must be moved through the system in the same way as an

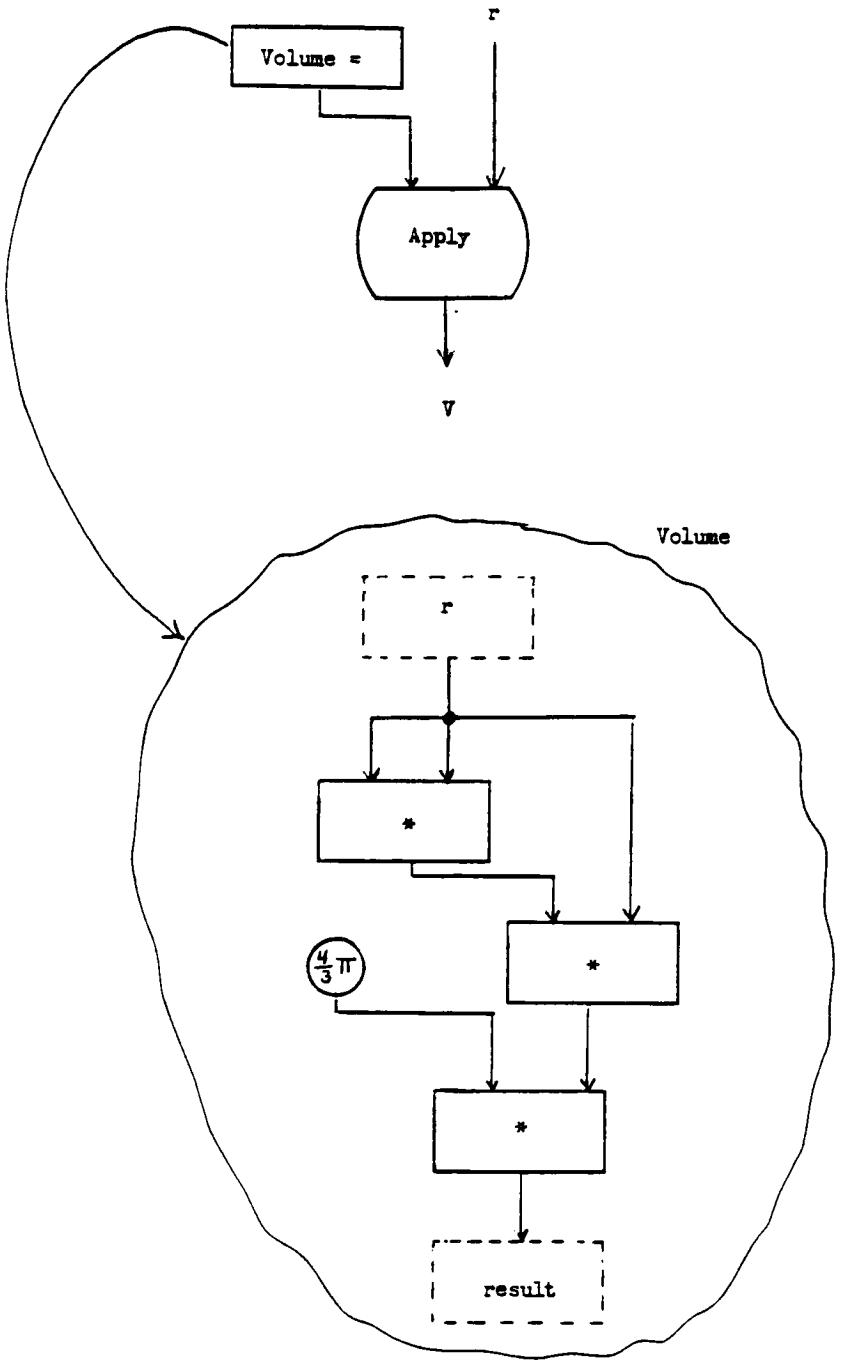


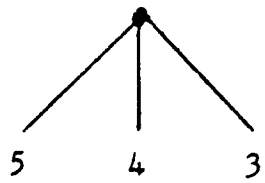
Figure 2.6 Apply Operator

elementary token and must be copied whenever a change is made to the structure, even if only one element will be altered.

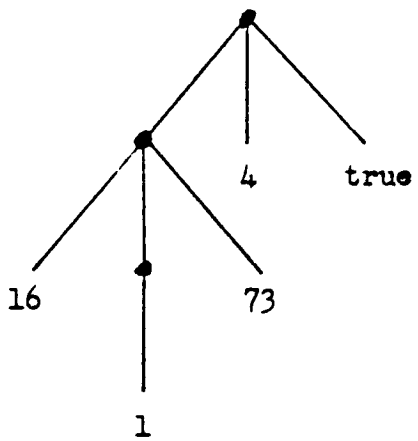
This is obviously impractical as far as overhead is concerned, and one solution is to add a memory to a data flow machine that will store only data structures [Myers 1982] [Rumbaugh 1975] [Dennis 1975]. Structures are then represented as trees whose nodes are substructures (the branches are ordered), and whose leaves are non-structure values. Tokens can then carry pointers to these structures, rather than the structures themselves. Examples of structures are shown in Figure 2.7.

Based upon pure LISP, each node of a structure must keep a reference count; i.e. the number of existing pointers to that node. No changes can ever be made to a shared structure; i.e. one with a reference count greater than 1 (single assignment rule). Instead, if a node's values must change and the reference count for that node is greater than one, then the contents of the node must be copied before any changes can be made to it in order to prevent side effects for other pointers to that same node. If a node's reference count becomes Ø, that node's memory space can be deallocated as it is now inaccessible. A garbage collection scheme may be used for this purpose.

Another problem with treating structures as a single object is that it limits runtime parallelism since a computation requiring a structure would not be able to run until the entire struc-

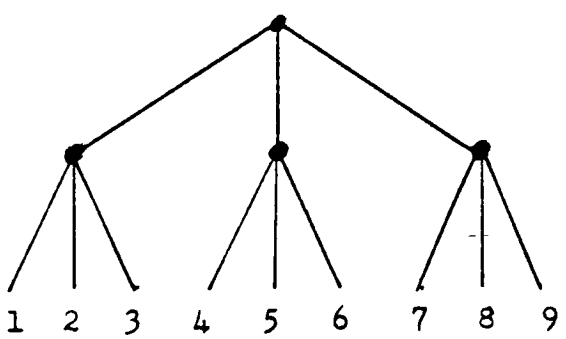


vector (5, 4, 3)



The LISP-like list

((16, (1), 73), 4, true)



A 3 x 3 matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Figure 2.7 Structures

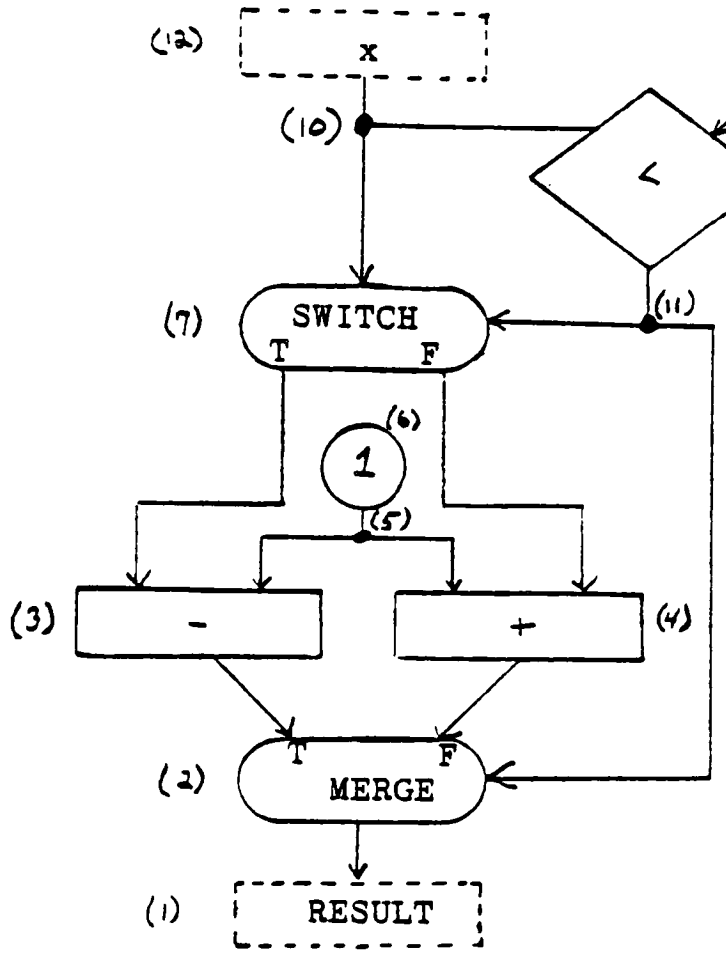
ture is complete, even though many of its elements may be available and could be used to start the computation. A proposed solution to this problem is the use of I-structures as a data type [Arvind and Thomas 1980]. An I-structure is an asynchronous, array-like data structure which allows random access to individual elements, and is useful in reducing data dependencies from the entire structure to individual elements of the structure. They also use a reference count for reclamation of structure space.

2.3. Machine Representation of Data Flow Graphs

Graph programs can be directly encoded to represent the machine code for a specially-constructed processor (or can be used as virtual machine code and interpreted on a conventional processor). There are many ways to represent the coding of a program, depending on which specific prototype of a data flow computer the program is intended to run. One method will be demonstrated here. Data tokens will apply only to single values, not structures.

The graphical program is represented as a set of contiguous memory locations, one node or instruction per location, called a code block (Figure 2.8) [Davis and Keller 1982]. Each instruction has a location, a function or op code, an ordered listing of the node's input arcs identified by the location of the nodes from which the tokens originate, and a location (instruction) to

Graph Program



Equivalent Pascal Program:

```
if x < 0 then
    x := x - 1
else
    x := x + 1;
```

Machine Representation

Instruction Location	Opcode	Input Path One	Two	Three	Destination of Results
1	result	2			
2	merge	3	4	11	1.1
3	-	7	5		2.1
4	+	5	7		2.2
5	duplicate	6			3.2, 4.1
6	1				5.1
7	switch	10	11		3.1 or 4.2
8	if (<)	10	9		11.1
9	Ø				8.2
10	duplicate	12			7.1, 8.1
11	duplicate	8			7.2, 2.3
12	input				10.1

Figure 2.8 Machine Representation of Data Flow Graphs

which the output token is sent. The destination of the result is in the form i.j where i is the location of the node and j is the input path [Treleaven 1979].

Each instruction also needs to keep an enabling count of the number of input tokens it is waiting for before the instruction can fire. Initially this count is set to its maximum number of input arcs. When the count reaches zero, the node can fire.

Corresponding to the instructions in memory is the data block, containing data tokens in contiguous locations which parallel the instruction nodes from which they emanate. That is, if location i in the code block represents a node of the graph, location i in the data block represents the token value on the arc leaving that node.

When a node becomes firable, i.e. its enabling count is zero indicating all tokens are available, the processor can execute the operation indicated by the instruction by fetching the values on the node's arcs. The result of the calculation is then stored on the node's output arc. The count of the receiving node is decremented; if it becomes zero, implying it is now firable, the cycle begins again. Any number of firable nodes can be processed concurrently. A list of enabled instructions is kept, and instruction execution is distributed over many processing units.

Firing of enabled nodes continues until no firable nodes are left.

CHAPTER 3

Machine Organization

This section will describe how a data flow computer's resources are configured and allocated to support execution of a program.

3.1. Packet Communication

Data flow computers are most often based on a packet communication machine organization, which consists of a circular instruction execution pipeline in which processors, communications and memories are interspersed with pools of work (Figure 3.1) [Treleaven 1982]. A program's instructions are stored in memory, and execution of the program consists of independent information packets, which may split and merge, traveling around the pipeline. In this way, packets of work are allocated to

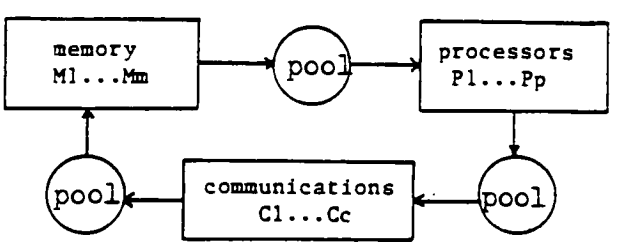


Figure 3.1 Packet Communication Organization

resources. Each packet to be processed is placed with similar packets in one of the pools of work (e.g. all the packets of enabled instructions are contained in the pool waiting for a free processor). When a resource becomes idle, it takes a packet from its input pool, processes it, places a modified packet in its output pool, and returns to an idle state. A system could be configured to have many identical resources between the pools, or have many individual pipelines connected by a communications network.

3.2. Synchronization of Instruction Execution

There are two schemes employed in synchronizing instruction execution [Dennis 1979b] [Treleaven, Brownbridge and Hopkins 1982]: token storage and token matching.

Token storage is illustrated in Figure 3.2 [Treleaven, Brownbridge and Hopkins 1982]. Data token packets arrive at the input pool of the Update unit. The Update unit stores the input tokens in their destination instructions in the Memory unit. At this time, the Update unit determines if all the tokens for that instruction have arrived, enabling it for execution. If so, the address of that instruction is placed in its output pool. The Fetch unit then uses these addresses to retrieve the instructions from memory and places them in its output pool to wait for a free processor. This basic scheme is essentially that implemented in the MIT data flow computer [Dennis, Misunas and Leung 1977] and a

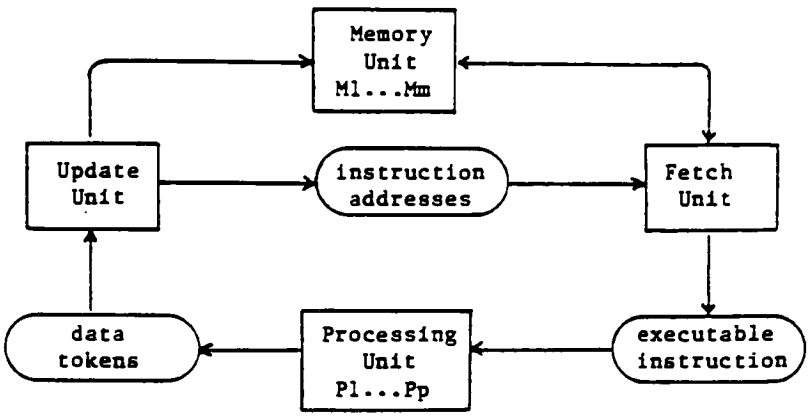


Figure 3.2 Token Storage

prototype data flow computer built by the Texas Instruments Company [Cornish 1979].

Token matching is illustrated in Figure 3.3 [Treleaven, Brownbridge and Hopkins 1982]. The Matching unit takes data tokens from its input pool and forms them into sets, using their destination instruction address to determine set membership. When a token arrives, the token store in the Matching unit is searched for a token with the same destination address. If no tokens are found with the same address, the enabling count required by the destination instruction is decremented by one. If the enabling count becomes zero, the instruction is now enabled and the token is sent directly to the Fetch/Update unit. Otherwise, the token is stored. If, however, tokens with the same address are found in the token store, the count is

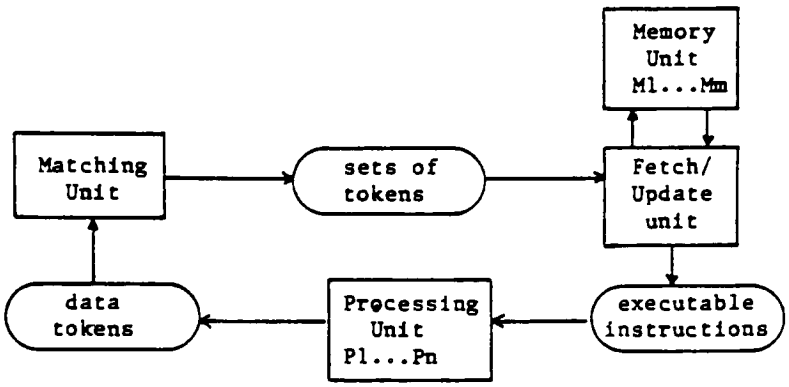


Figure 3.3 Token Matching

decremented by one, and if the count is still greater than zero, the token is stored with the rest of its set; if the count becomes zero, implying the instruction is now enabled, the whole set of tokens is released to the Fetch/Update unit. This unit forms a packet consisting of the instruction and its tokens, and places it in its output pool to wait for a processor. Examples of this scheme include Irvine Data Flow [Arvind, Kathail and Pingali 1980], the Manchester Data Flow System [Watson and Gurd 1979], and the Newcastle Data-Control Flow Computer [Hopkins, Rautenback and Treleaven 1979].

The main advantage of token matching over token storage is that it allows the removal of the restriction that only one token can be on an output arc at any one time. The arcs then become FIFO queues, allowing tokens to be matched into sets, and allow-

ing a program's instructions to be used reentrantly.

CHAPTER 4

Implementations

There are many data flow models and implementations in the literature. This chapter will describe the major ones, starting with Jack Dennis's data flow computer at MIT, which has formed the basis for most other projects.

4.1. MIT Data Flow Computer

The organization of the M.I.T. machine uses token storage, feedback and a cell block architecture. The structure of the machine is shown in Figure 4.1 [Dennis 1980].

Only one token is allowed to exist on an arc at any given time; i.e. the firing rule is that an instruction is enabled when all of its operands are present and there is no token existing on its output arc.

An enabled instruction packet enters the arbitration (routing) network, which passes the packet on to the appropriate processing unit according to its opcode. The processing unit performs the necessary operation and sends a result packet to the distribution network, which directs it to the cell block containing the destination instruction.

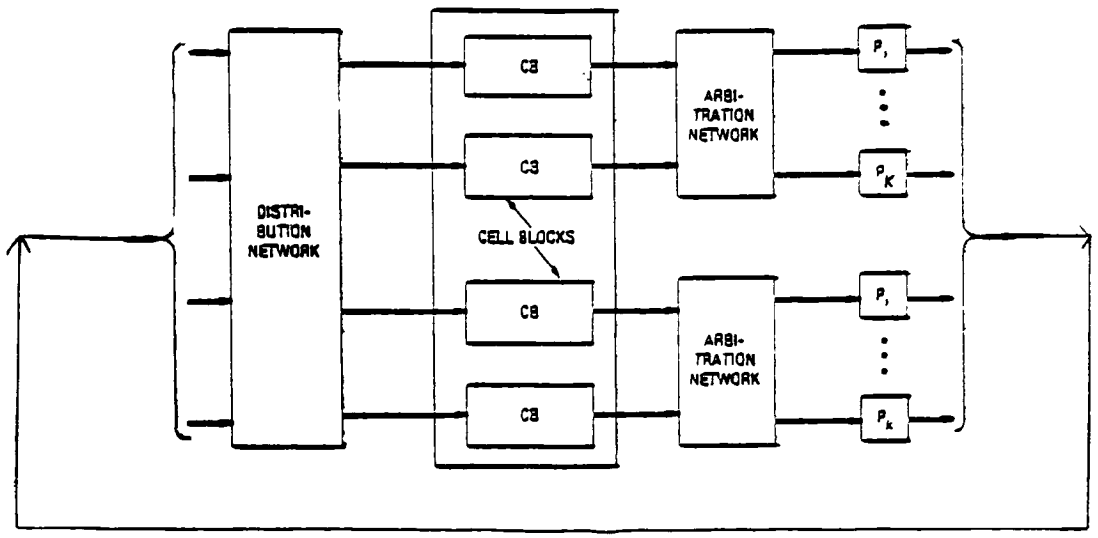


Figure 4.1 MIT Data Flow Computer

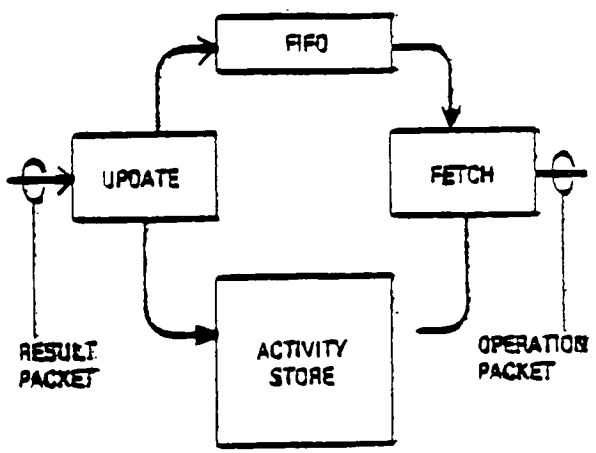


Figure 4.2 Cell Block

Each cell block is structured as in Figure 4.2. The cell

block functions in the way described for token storage in section 3.2, with the program instructions stored in the Activity Store.

4.2. Manchester Data-Flow Computer

The Manchester Data-Flow Computer, shown in Figure 4.3 [Treleaven, Brownbridge and Hopkins 1982], is very similar to the MIT design with two major exceptions.

The first is that it uses token matching. The token queue is a FIFO buffer and leads to the Matching Store that is associative in nature, implemented using RAM with hardware hashing techniques. A token may be one of a pair or a single input to an instruction. Token pairs from the matching store, or single tokens for single input instructions that have bypassed the Matching Store, are routed to the Instruction Store. Here the

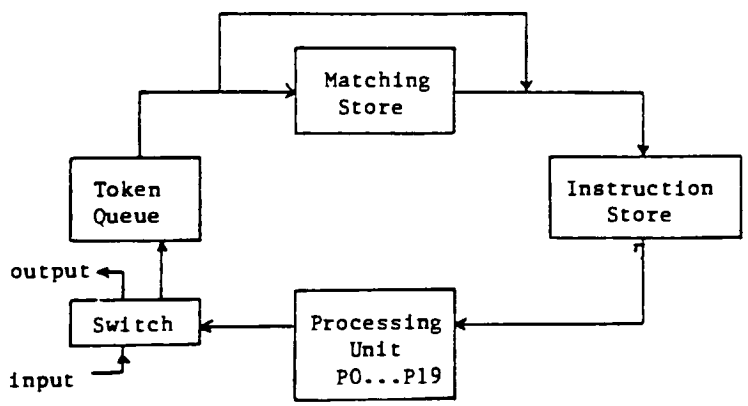


Figure 4.3 Manchester Data-Flow Computer

tokens are combined with a copy of their destination instructions to form an executable instruction packet that is passed to the Processing Unit. The Processing Unit consists of an arbitration and distribution system and microprocessors, any of which can be assigned to an executable instruction.

The second major difference is the label field carried by each token. Since the arcs of the program graph are viewed as FIFO queues and consequently more than one token can be on an output arc of an instruction, each data token must carry a label field which identifies the process to which the token belongs, the destination instruction address, and an iteration number specifying which token on an arc it is. This allows a program's instructions to be used as reentrant code, and greatly increases concurrent execution of a program.

In addition, the Manchester design has a built-in switch to provide I/O for the system.

A complete description of this machine is found in [Watson and Gurd 1982] and [Watson and Gurd 1979].

This project has included the design of LAPSE, a high level, single assignment language, and the implementation of a translator for LAPSE into data flow program graph language. It also included translating conventional languages into data flow graphs, which resulted in the development of an experimental compiler for a subset of Pascal by P. J. Whitelock.

4.3. Irvine Data Flow Machine

The Irvine Data Flow Machine originated from research at the University of California at Irvine, and is now located at MIT, where research is continuing. It supports the ID high level data flow language, makes use of VLSI with a multiprocessor design, token matching, supports I-structures, and uses a sophisticated token identification scheme [Arvind, Gostelow and Plouffe 1978], [Treleaven, Brownbridge and Hopkins 1982].

It consists of N processing elements and a packet communications network for routing a token from one physical element to

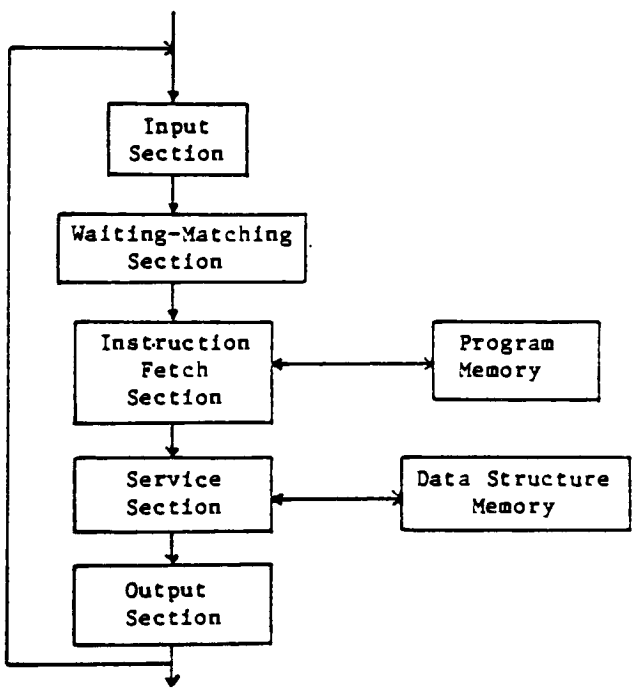


Figure 4.4 Irvine Data-Flow Processing Element

another. One processing element is shown in Figure 4.4. Tokens are routed to the element that holds its destination instruction in its Program Memory. If a token is destined for the same physical element that generated it, it can bypass the N x N network and use a short-circuit path back to itself.

4.4. Texas Instruments Distributed Data Processor (DDP)

The DDP was designed by Texas Instruments to investigate the potential of data flow as the basis of a high-performance computer. It was constructed using off-the-shelf technology. The project began in 1976 and has been operational since 1978 [Cornish 1979].

It is not connected with any high-level data flow language. Instead, a cross compiler, based on the Texas Instruments Advanced Scientific Computer's optimizing FORTRAN compiler, translates FORTRAN subprograms separately into directed graph representations and a linkage editor combines them into a single program.

The DDP has many similarities to the MIT machine, in that an instruction is enabled when a token is present on all its input arcs and no token is present on any of its output arcs. In addition, it uses token storage and control tokens for feedback signals. The machine organization, shown in Figure 4.5 [Treleaven, Brownbridge and Hopkins 1982], is significantly different from the MIT computer, however. It has four identical data flow

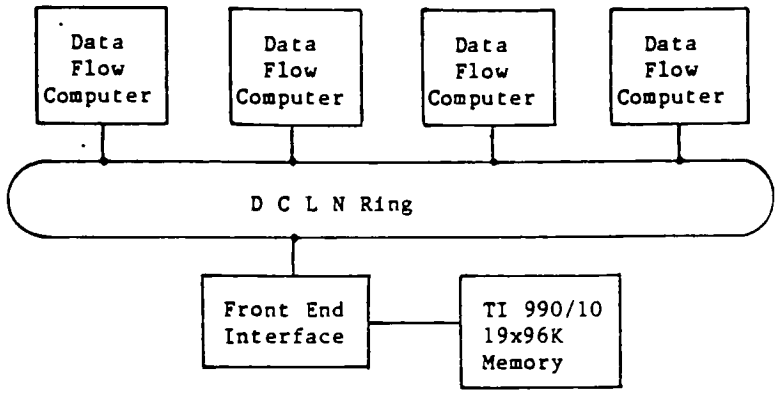


Figure 4.5 Texas Instruments Distributed Data Processor

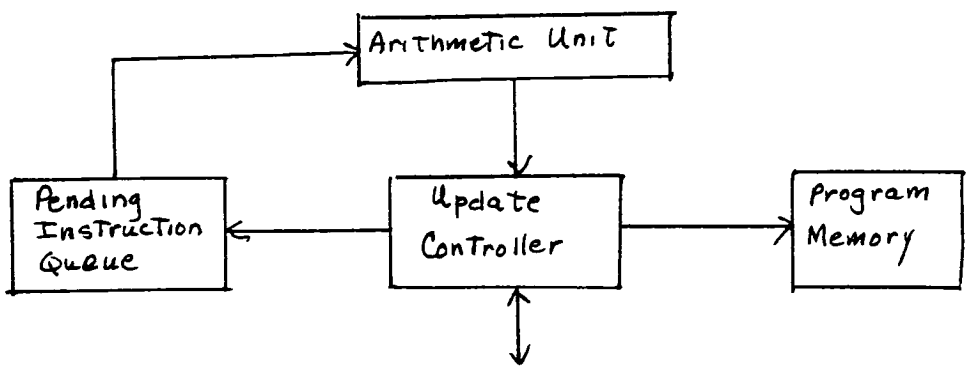


Figure 4.6 Individual Data Flow Computer

computers over which a program is distributed, and a Texas Instruments 990/10 minicomputer acting as a front-end processor for I/O, providing operating systems support and handling collection of performance data. These five units are connected by the DCLN ring, which is a variable-length, word-wide, circular shift

register. The ring may carry up to 5 variable-length packets in parallel.

Each data flow computer consists of four principle units, as shown in Figure 4.6. Executable instructions are removed from the Pending Instruction Queue by the Arithmetic Unit and processed. Output token packets are released to the Update Controller which stores the token in the instruction in Program Memory and decrements the instruction's Predecessor Count. If that count becomes zero, the instruction is ready to execute and a copy is placed in the Pending Instruction queue.

4.5. Utah Data-Driven Machine

The Utah Data-Driven Machine #1 (DDM1) was designed by Al Davis and his colleagues while working at Burroughs Interactive Research Center in La Jolla, California, and completed in 1976. An improved version of DDM1 now resides at the University of Utah, where the project is continuing under support from Burroughs Corporation.

The recursive architecture of this machine is much different from the ones outlined in the previous sections. Rather than a packet communication organization, it has an expression manipulation machine organization (identical resources organized into a tree, where each resource contains a processor, communication and memory capability). It has a VLSI implementation, and exploits physical locality to decrease message frequency and increase

speed.

Its tree structure has a single root and a possibility for up to eight sons at any node. A node is a processor-store element (PSE) which consists of a processor module and its associated local storage module. A block diagram of a node is shown in Figure 4.7 [Davis 1979], [Treleaven, Brownbridge and Hopkins 1982].

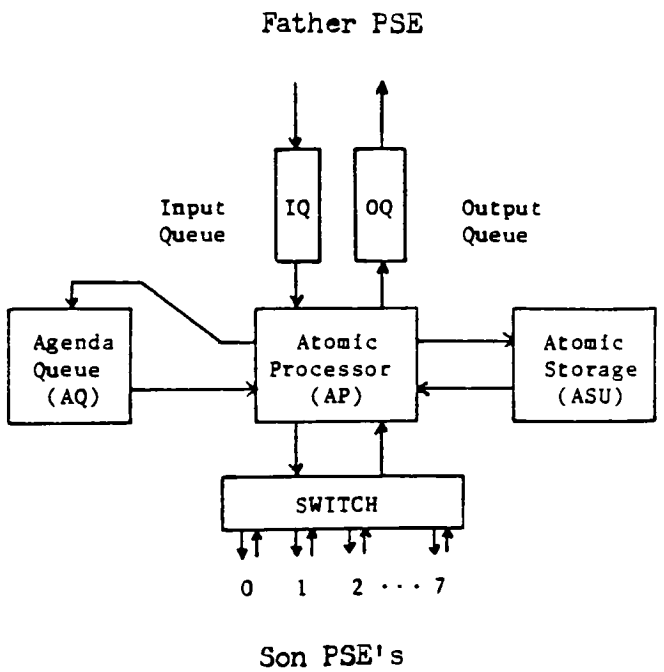


Figure 4.7 Utah Data Driven Machine

4.6. LAU System

The LAU System is based at the CERT Laboratory at the University of Toulouse, France. The project began with the design of a high-level single assignment language, and a compiler and simulator for that language. This led to a the design and construction of a powerful 32-processor data-driven computer.

Program representation is based on three logical types of memory, one each for instructions, data, and control information. The machine language has a three-address format which consists of an operation code, two data memory addresses for input operands, and a data memory address for the result operand.

The LAU machine has a packet communication organization with token storage. Figure 4.8 [Treleaven, Brownbridge and Hopkins 1982] shows the structure of the processor. It consists basically of three units. The Memory unit provides storage for instructions and data. The Control Unit, the truly original and unique part of the processor, contains the Instruction Control Memory and the Data Control Memory. The Processing Unit consists of 32 identical processing elements, each element being a 16-bit micro processor [Syre, Comte and Hifdi 1977], [Treleaven, Brownbridge and Hopkins 1982].

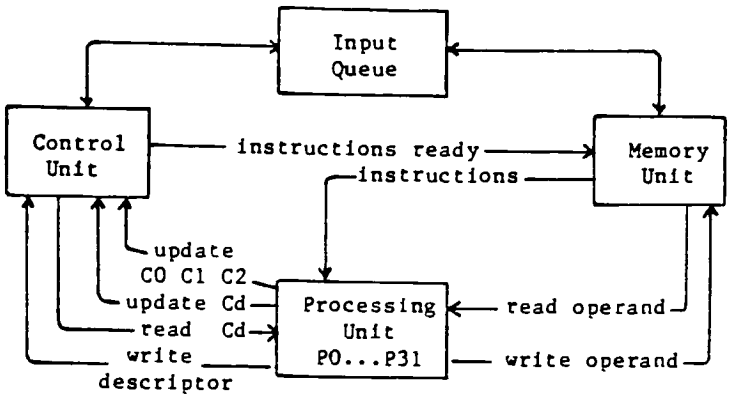


Figure 4.8 LAU System

4.7. Newcastle Data-Control Flow Computer

The group at the University of Newcastle upon Tyne were interested in the data flow program organization only (not the resulting machine architecture), the suitability of these programs for a general-purpose decentralized computer, and the possibility for combining them. The JUMBO computer which they developed will be described here, which was built to study the integration of data-flow and control-flow computation.

The JUMBO computer has a packet communication organization with token matching. However, data can also be embedded in the instruction. When an instruction is enabled, the token inputs and the embedded inputs are merged to produce a set of up to eight data values and addresses. The execution of an instruction can produce data tokens, data to store in memory, and control tokens.

A block diagram of the computer is shown in Figure 4.9 [Treleaven, Brownbridge and Hopkins 1982].

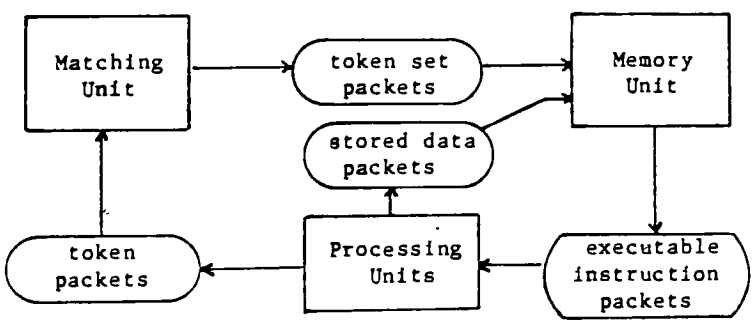


Figure 4.9 Newcastle Data-Control Flow Computer

CHAPTER 5

Project Description

As the main project for my thesis, a simulator has been written which executes data flow program graphs. It was written in Concurrent Euclid, chosen because of its support for concurrent processes and monitors and its portability. Disadvantages of Euclid are that it does not support real numbers and the maximum value of an integer is only 32767; test data must be chosen accordingly.

The overall approach to the simulator is based on the models of Dennis [Dennis 1974], Watson and Gurd [Watson and Gurd 1979], and Arvind [Arvind and Gostelow 1975], [Arvind and Gostelow 1982], where:

1. An instruction can execute whenever its operands become available, and any number of enabled instructions can execute concurrently.

2. All operators are free of side effects; that is, enabled instructions can execute in any order or concurrently and the end result will be the same with no error produced.

3. Tokens carry a tag which identifies not only the instruction to which the token is going, but a code block identification number (which identifies the instantiation of a subprogram or a

loop within a program or subprogram), and an iteration number. In this way, many instantiations of an instruction can occur concurrently.

4. Procedure calls can occur concurrently and loops can be unfolded in the manner of Arvind's U-Interpreter. [Arvind and Gostelow 1982]

5.1. Description of Model

The conceptual design of the model is shown in figure 5.1. It has a packet communication network to carry data tokens from one unit to another. Tokens can be of type integer, boolean or character. Again, reals have not been implemented because Euclid does not support them.

Program instructions are read by the simulator and stored in Program Memory. They can be referenced by the Fetch and Match Units. Any constants which are to be used as input to an instruction are "permanently" entered into the token store in the Match Unit under that particular instruction number, making the constant input always ready and available for use.

When an instruction has been executed, a packet containing a data token is sent to the Match Queue, which is a queue of packets. The Match Unit removes packets from this queue and, by referencing the instruction to which the packet is destined, determines if this one token enables the instruction. If it

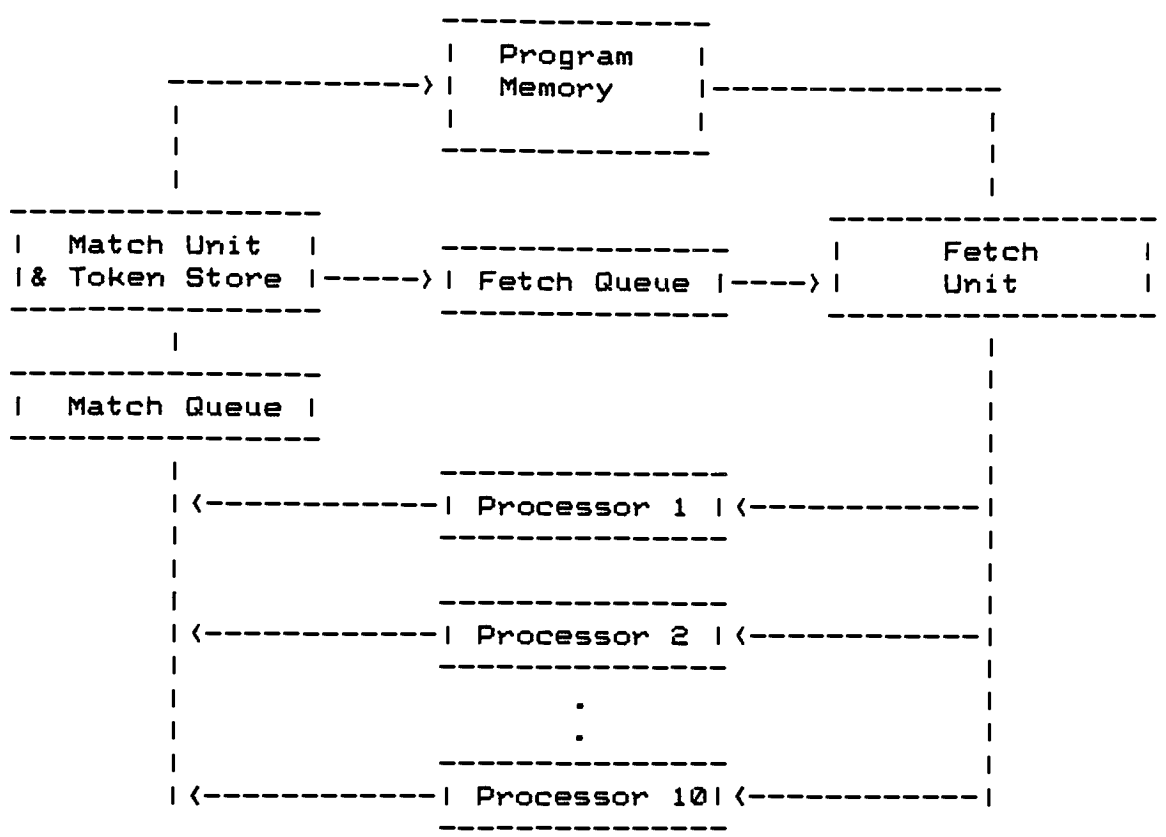


Figure 5.1 Model of Simulator

does, a packet with the data token is immediately sent to the Fetch Queue. Otherwise, the Match Unit must check it's Token Store for the presence of other tokens for that instruction. If the instruction requires two data tokens, i.e. enabling count is two, the Match Unit then checks for the presence of a constant in the Token Store for that instruction. If there is one present, a data token is produced for that constant, added to the packet, then the packet is sent to the Fetch Queue. If there is no constant, or if the enabling count is greater than two, then the

Match Unit must look in the Token Store for other tokens for that instruction. If one or more are found, the enabling count for that instruction is now decremented by one and checked. If it is now zero, then the stored tokens are removed from Token Store, added to the packet and the packet is sent to the Fetch Queue. If the enabling count is not zero, the newly arrived token is stored along with the other tokens for that instruction.

In the case where no other tokens are found in the Token Store for that instruction, and the enabling count has not reached zero, the newly arrived token is entered into the Token Store under its destination instruction number to await the arrival of tokens which will enable the instruction.

The Fetch Unit repeatedly removes a packet from its queue. It gets a copy of the packet's destination instruction from program store and then sends the packet and the copy of the instruction to a processor for execution of the instruction.

There is a pool of 10 processors in the simulator; each processor can execute any opcode. However, in order to make it look as if any one processor is dedicated to only one operation and there are an "unlimited" number of processors for any one operation, the Fetch Unit dynamically allocates the processors as they are needed and assigns the op code at that time. Upon completion of an instruction execution, the processor goes back into a "pool" of available processors. The number of processors is

large enough that the Fetch Unit seldom has to wait for one to become available.

When a processor executes an instruction, the incoming data tokens are "consumed" by the processor and disappear. The resulting data token(s), if any, which are produced by the operation are sent in one or more packets to the Match Queue.

The Match Unit, Fetch Unit, and the processors are all operating concurrently and therefore packets are constantly being pipelined through the system.

There are two types of constants used in the model: program constants and start constants. They will look the same in a graph program but are handled differently by the simulator. Start constants are constants which are created at the start of the program as if they were produced dynamically by the execution of a statement. They are used to execute an instruction only once, and their presence alone enables the instruction. In this way, execution of the program begins; start constants are the "initial" conditions which "get the ball rolling." This is in contrast to a program constant which must be available to an instruction throughout the program and may be used in execution many times. Examples of each are shown in the simple program in figure 5.2 which loops five times and then stops. (The statements identified as L, L1, D, and D1 are used in loops to implement the grouping of tokens into sets and are explained in detail

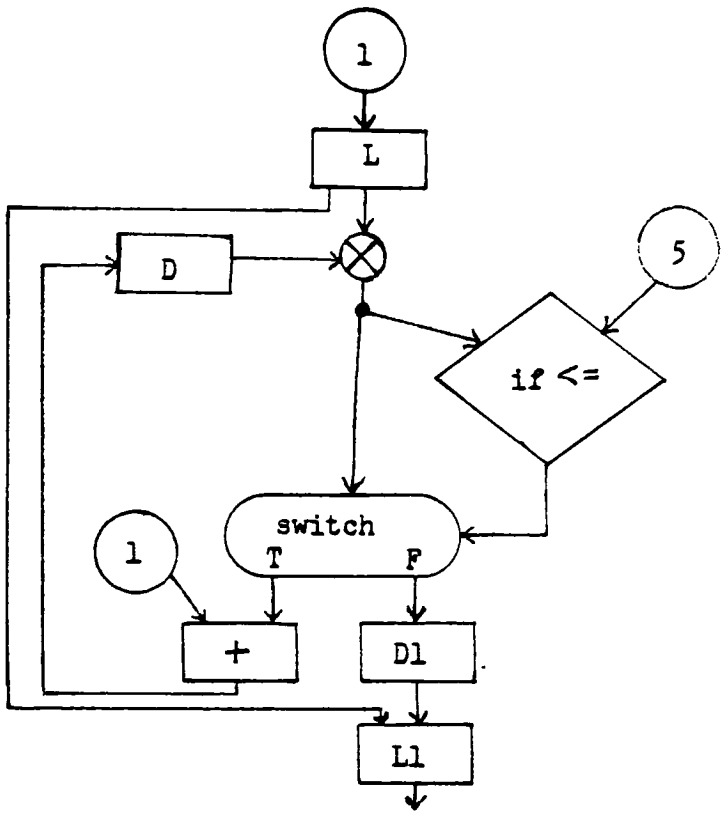


Figure 5.2 Program Constants and Start Constants

in section 5.6.1.9.) The constant 1 entering the L statement is a start constant. It is the only input to that statement; its arrival causes the statement to be enabled and thus executed. This particular L statement will not execute again in the program. In contrast, the constants 5 and 1 entering the "if <=" and "+" statements respectively are only one of two input tokens to their destination statements. Their arrival does not enable the instruction; the instruction still must wait for the arrival of the second data token to be enabled. These program constants

will be required on each execution of their statements, each time the loop is executed.

5.2. Data Flow Program Organization

A data flow program can be broken up into code blocks. A code block is made up of one or more operators. Each loop within the program is a separate code block, and each subprogram is a code block. Groups of instructions not belonging to any loop or subprogram belong to the initial code block number which is 1. Each code block, as the program executes, has its own unique identifier.

As a program begins execution, its code block identifier (CID) is 1. Each time a loop is initially entered, the CID is assigned a new, unique integer, and the old CID is saved for use again on exit from the loop. Also, on entry to each subprogram, a new CID is assigned and the old one saved for use upon return to the calling program. New CID's are assigned sequentially. Each time a new one is required, the last number assigned is incremented by one and then assigned as the new CID.

An iteration count (IID) is also kept. It is initially set to 1; with each succeeding iteration of a loop, the IID is incremented by 1. Upon exit from a loop, the IID becomes whatever it was upon entry to the loop.

Therefore, at any point in a program's execution, the tokens of each instantiation of each instruction have a unique tag with the current CID and IID.

As an example, the outline of a program is presented in figure 5.3 along with the tags of the tokens, represented as (CID, IID).

This method of tagging data tokens enables the execution of loops to be "unfolded" in the manner of Arvind and Gostelow's U-Interpreter [Arvind and Gostelow 1982] and also allows concurrent instantiations of subprograms. This would include calling of a subprogram from different points in a program, or recursive calls to a subprogram. Each instantiation of any instruction within a loop or subprogam will have its own unique tag (CID, IID).

Tagging data tokens in this way greatly increases concurrency, as shown in the graph program in Appendix D which computes factorial 1 to factorial 7. As can be seen, the loop computing the numbers 1 to 7 has very few instructions. This loop will run very quickly compared to the part of the program which actually computes the factorial recursively. It has been shown by actually testing the program on the simulator that the loop from 1 to 7 completes its 6th iteration before the first factorial is produced, and it completes its 7th iteration almost immediately after 1! is computed. This is possible since computation in the loop does not depend on computation of the fac-

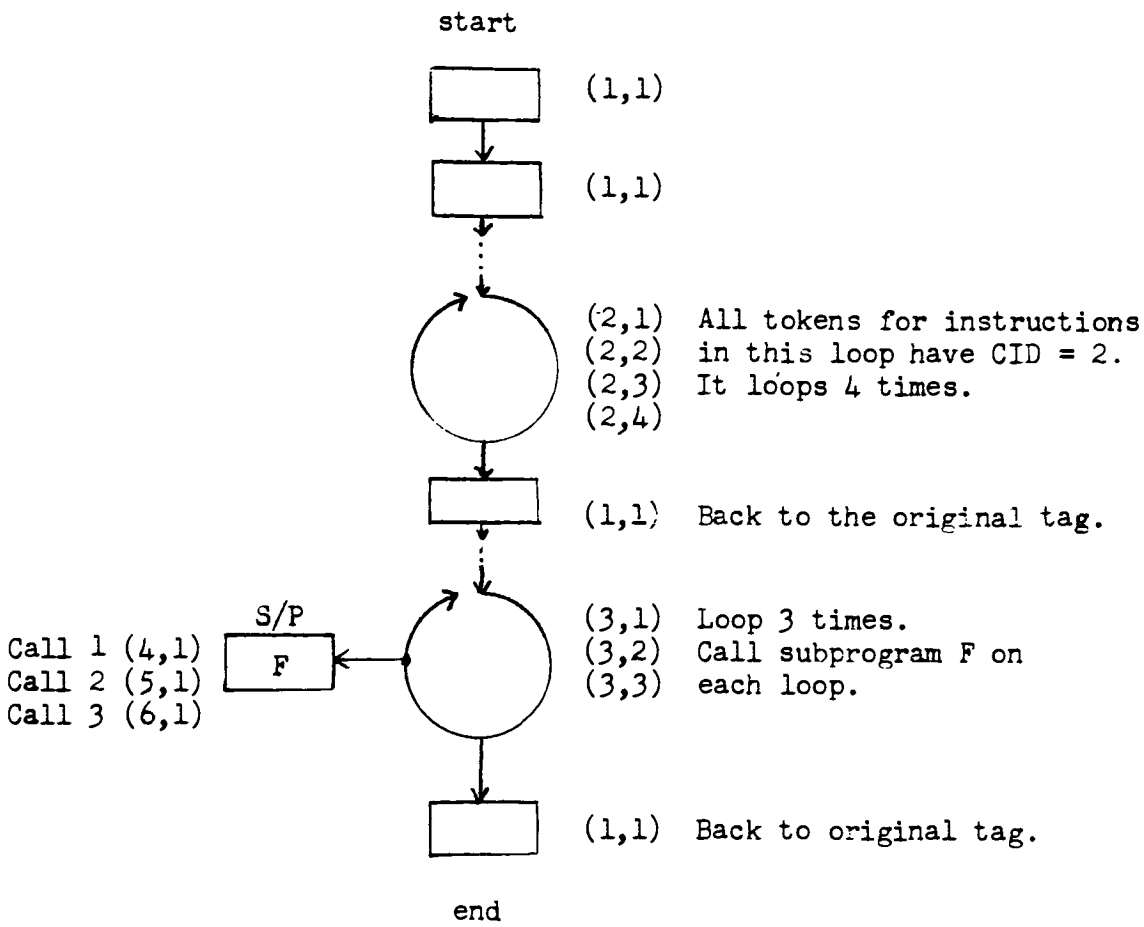


Figure 5.3 Division of Program into Code Blocks

torial. This indicates that the factorial subprogram has greater than six instantiations running at once (greater than six since the factorial subprogram calls itself once for 1!, twice for 2!, etc.). Therefore, six different factorials are being computed at

one time, and there is the probability that there is greater than one data token in the system for many instructions in the factorial subprogram. This can be done since the simulator dynamically assigns a different tag to the data tokens which identifies the context from which the instruction was called.

Since there is no data dependency between successive computations of n!, the loop can unfold and compute all of the factorials concurrently, thereby attaining maximum concurrency. It is conceivable that n! may even complete before (n-1)!.

If the subprogram computations were dependent on one another, that would be the only constraint upon ordering the execution of instructions within an unfolded loop. Consider, for example, the following program written in ID (a high level data flow language) taken from [Arvind and Gostelow 1982] which integrates a function f by the trapezoidal rule.

```
(initial s <-- (f(a) + f(b)) / 2;
     x <-- a + h;
for i from 1 to n-1 do
     new s <-- s + f(x);
     new x <-- x + h;
return s) * h
```

The computation of i from 1 to n-1 could execute independently of the instructions within the loop, and the computation of new x can be done independently of the computation of new s. Each new s and new x, however, depend on the old x, and therefore they must execute in order: new s and new x must always wait for

the x of the previous loop.

5.3. Data Structures

This section will describe in detail the data structures used to implement the various elements of the data flow system; in particular, the instructions, the packet communication network, the Match and Fetch Queues, and the Token Store in the Match Unit.

5.3.1. Instructions

The data flow program machine language instructions are read in and stored in an array called "instruction" of maximum size "maxinstr" which is presently set at 1000. Each element of the array represents one instruction in record form, and contains the following information:

(1) The opcode: an integer representing the mnemonic opcode.

(2) The enabling count of the instruction: the number of context control tokens and the number of data tokens required for execution.

(3) The string to be printed along with output. This is used only for output statements; for other statements, it is undefined.

(4) The file number from which data should be read. This applies only to input statements; for other statements, it

is undefined.

(5) An array of input types. This array is of size 20, which is the maximum number of inputs any instruction can have. In fact, most instructions will have only one or two; only the input and output statements and those statements dealing with subprograms will be allowed to have up to the maximum number of inputs. The type of the data token that will be input to port 1 is the first element of this array, the type of the token input to port 2 is the second element, and so on. Refer to Appendix B for data type codes.

(6) A pointer to the beginning of a linked list of information defining the destination(s) of output data tokens produced by execution of the instruction. Each element of this list is a record and contains:

a. the output port number of the data token to which this destination information applies.

b. the type of the data token being sent.

c. the instruction number to which the token is being sent.

d. the port number of the destination instruction to which the token is being sent.

e. a pointer to the next record in the list.

One output port can have any number of destinations specified; this precludes the necessity of a duplicate instruction. The destination information is read in along with the program instructions. The destination information for each instruction is placed immediately after the instruction itself and is ordered by output port number.

5.3.2. Packet Communication Network

The network described here is the system of packets of information which are "sent" from one unit of the simulated data flow computer to another (see figure 5.4). The packets all look the same, no matter where they are in the system. The difference is only in how each unit handles them.

A packet has a header, which is a record containing the following information:

(1) The instruction number to which the packet applies.

(2) The tag of the data token(s) in the packet; i.e. the code block number and the iteration.

(3) A pointer to a single data token or a linked list of data tokens, depending on what part of the system this packet is headed for and what the instruction is.

The data tokens themselves are records and contain the following information:

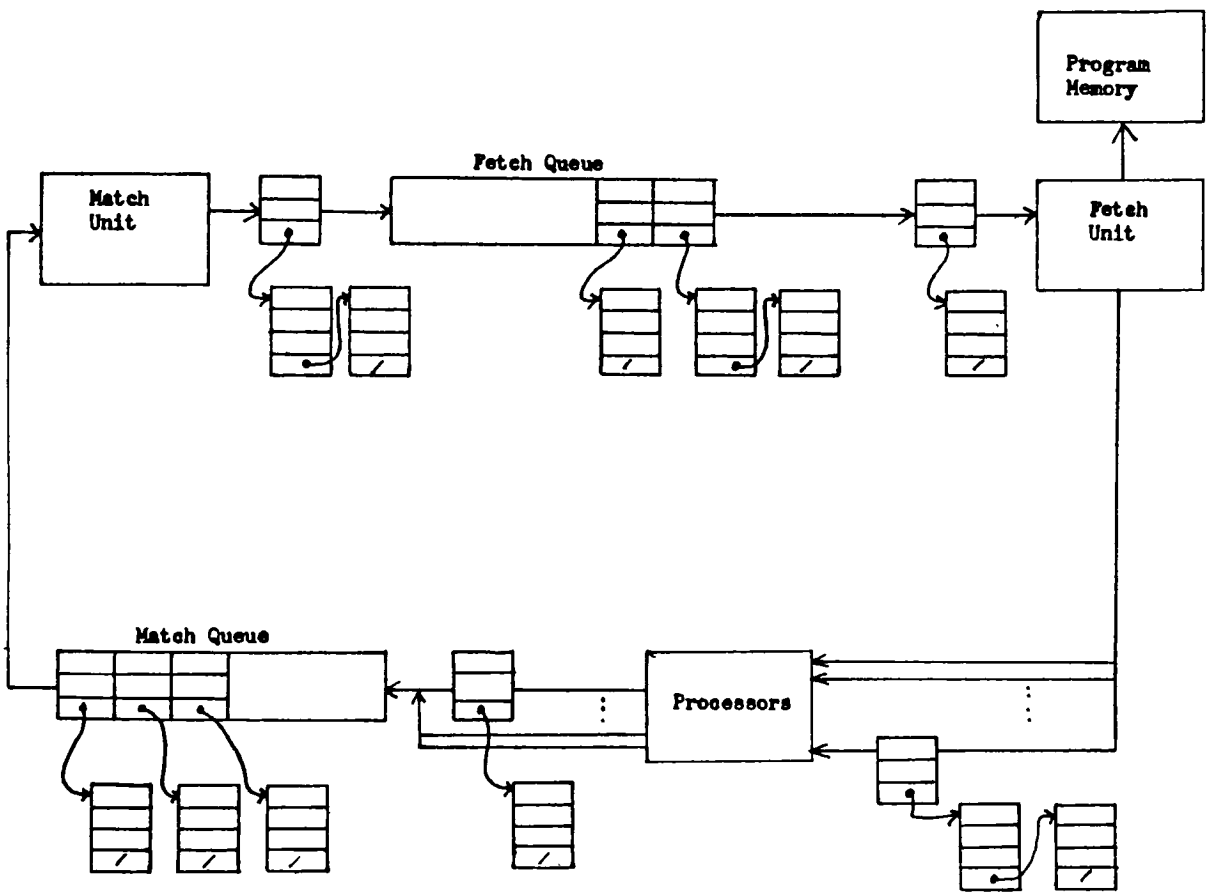


Figure 5.4 Packet Communication System

(1) The type of the data token; i.e. integer, character, boolean, instruction address, or context control.

(2) The actual value of the token. The values are represented as shown below.

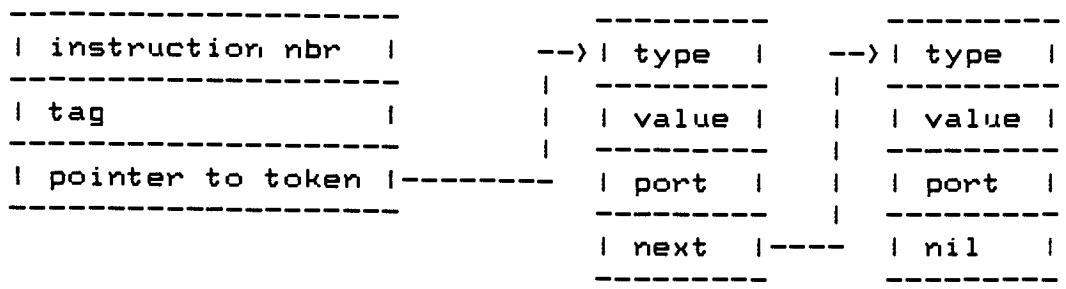
If the type is:	the value is:
integer	integer value
character	ordinal value of the character
boolean	an integer: 1 = true, 0 = false.
instruction address	an integer representing the number of an instruction in the array of instructions.
context control	two integers: the first representing the code block number and the second, the iteration number.

There is also a capability for implementing pointer type tokens; in this case, the value would be a pointer to a structure. However, at present, structures have not been implemented.

(3) The port number through which the data token will enter its destination instruction.

(4) A pointer to the next data token in the packet, or nil if this is the only token or the last token in the list.

A packet with two data tokens looks like this:



Packets which are on their way to a processor would contain the header and a linked list of all the data tokens necessary to execute the instruction.

Packets produced by a processor and going to the Match Queue would in general contain only the header and one data token. The exception to this is a packet produced by execution of the activate or end statements, in which case all data tokens required are sent in one packet to the next instruction to be executed.

The Match Unit removes a packet from its queue and, using the information in the header record, either stores the tokens in its token store to await the arrival of other tokens which will enable the instruction, or sends the packet on immediately to the Fetch queue, sometimes with the addition of other data tokens from the token store.

The Fetch Unit removes a packet from its queue and, by looking at the header, determines which op code is to be executed. It then sends the entire packet to a processor "created" for that

op code.

By using a small packet which points to a list of data tokens, the packet alone can be sent around the system, keeping overhead low, while the tokens remain "stationary". Tokens are easily shifted around from one packet to another by simply redefining the pointer in the packet or a pointer in the list of tokens. They are also easily added to and removed from lists of tokens in token store by simply setting pointers. This ability makes the concept of a packet and tokens a very flexible and efficient data structure.

5.3.3. Match and Fetch Queues The Match and Fetch Queues actually share one pool of buffers, which is an array of records representing headers of packets.

To store a packet in the Match Queue, a processor must acquire a free buffer, i.e. a location in this array of records, store the packet information in that array location, and then enter that location number in the Match queue. The Match Queue itself is just an array of buffer locations (integers), managed as a queue. Access to the Match Queue is shared between the processors, which are the producers of packets, and the Match Unit, which is the consumer. The Match Unit repeatedly removes a buffer location from the Match Queue (as they become available), gets a copy of the contents of the buffer, and then releases the buffer location to the pool of free buffer locations.

Storing information in the Fetch Queue is a similar process, the producer process being the Match Unit and the consumer process, the Fetch Unit.

One can picture a queue to look as shown in figure 5.5, with the packet headers pointing to the data tokens.

The relationship between the pool of buffers and the Match and Fetch Queues is shown in figure 5.6. At any point in time, the buffer numbers in the two queues are mutually exclusive.

5.3.4. The Match Unit

The basic data structure of the Match Unit is what has been called the token store (see figure 5.7). It is an array of size "maxinstr," which is the same size as the array of instructions; this allows a place in the token store for each instruction in the data flow program.

The token store is the structure which will store the data tokens for each instantiation of each instruction while they are waiting to be matched up with other tokens, thus enabling the instruction. In order to keep tokens for each code block number and each initiation separate from one another, and yet be readily and easily accessible, the tokens are stored first by instruction number, then code block number, and then iteration number.

The basic information stored in the array for each instruction, independent of code block number and iteration, is:

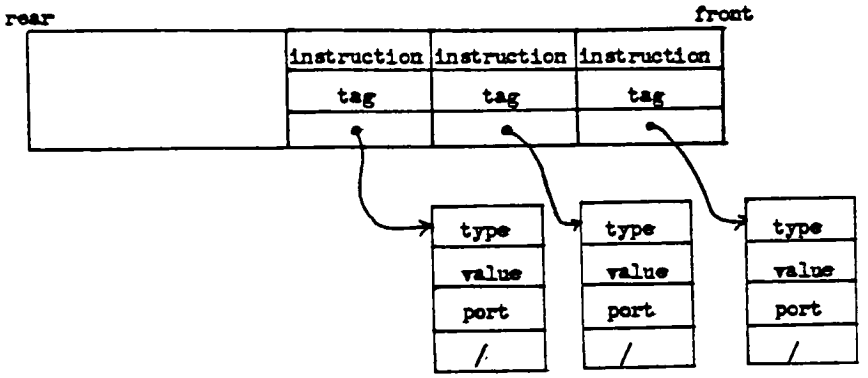


Figure 5.5 Queue

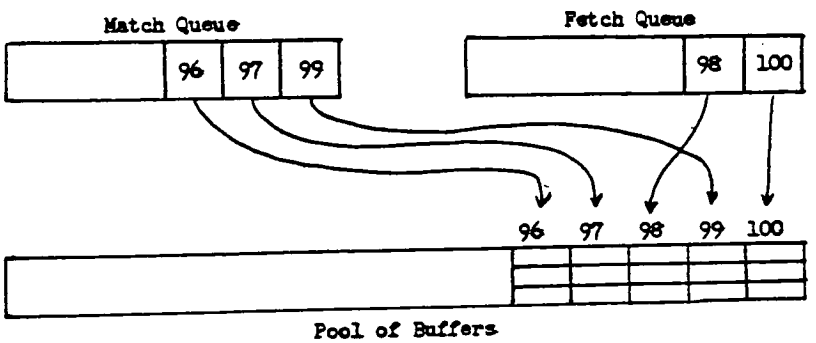


Figure 5.6 Queues and pool of buffers

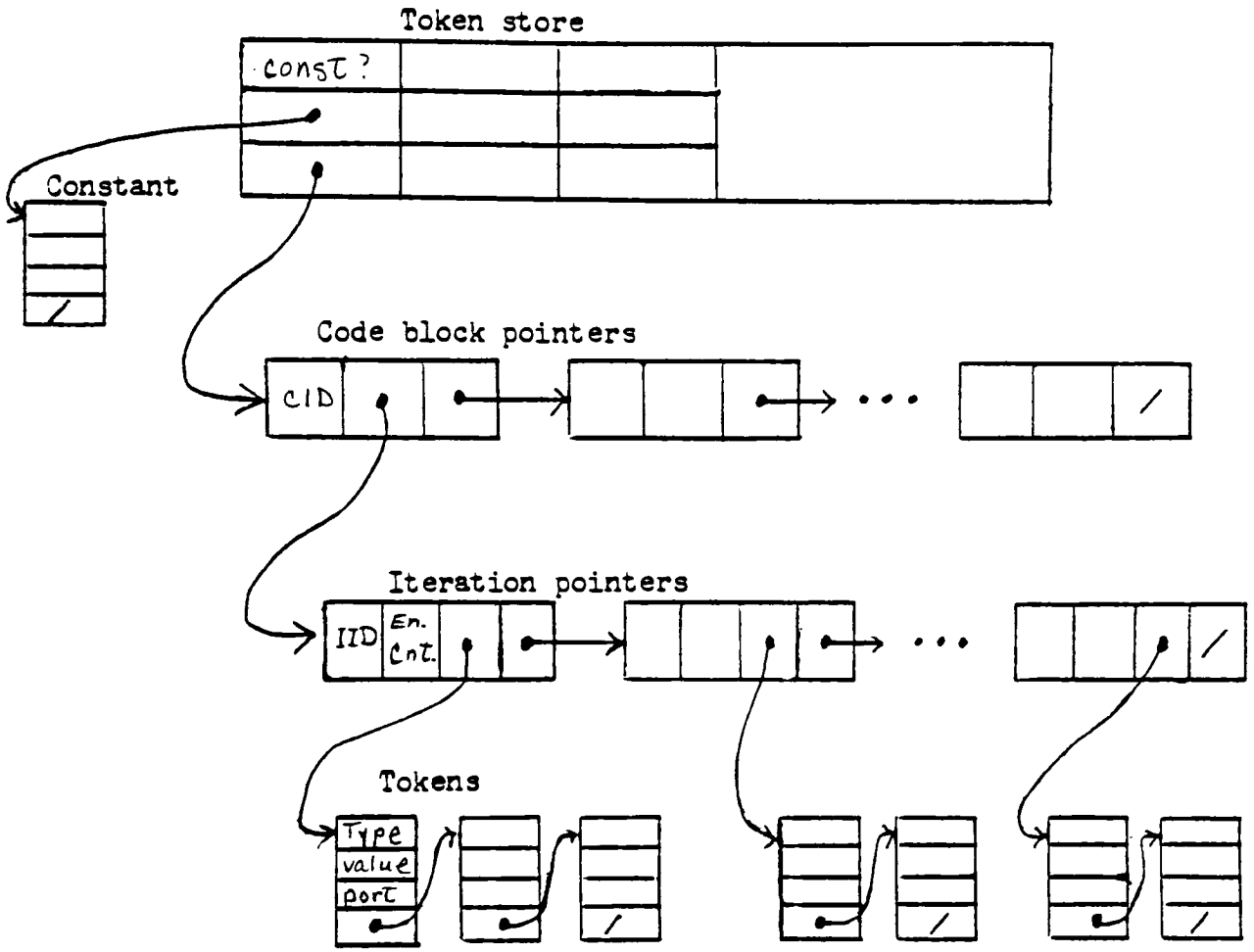


Figure 5.7 Token store in the Match Unit

(1) A boolean indicating whether or not there is a constant present as input to this instruction.

(2) A pointer to the constant data token, if one exists.

(3) A pointer to a list of code block numbers for which there presently are tokens stored.

The list of code block pointers is also a linked list; for each code block number, there is a pointer to a linked list of iteration numbers which occurred under that CID. For example, in a loop, each instruction could have several instantiations, where each instantiation would have the same CID but different IID's. The list of iteration pointers carries an enabling count for each instantiation of the instruction, and a pointer to the list of actual data token(s).

5.4. Structure of the Simulator Program

The program consists of a main module, three modules which are used by the main module, and five monitors. They are described in detail in this section.

There are also two files of definitions of constants, types, and data structures used throughout the program. The main module needs to include the larger of these files; the other modules include one or the other. The larger of these files is "definitions1", the other is "definitions2".

5.4.1. The Main Module

The main module of the simulator is in file sim.e. This module contains:

(1) The declarations for some additional data structures used in the program.

(2) An "initially" section which reads in and sets the trace indicator (refer to section 5.5). It also calls certain procedures in the Load module to read in the data flow machine language program, initialize the Match Unit, and set up the start condition for data flow program execution

(3) The processes which make up the data flow computer; i.e. the Match Unit, the Fetch Unit, and the Processors.

5.4.2. Load Module

The Load module, file load.e, contains all the procedures necessary to read in the machine language data flow program and start execution.

In procedure Readprogram, the number of statements is first read, then the statements themselves. If any input statements are included in the program, the files specified are opened.

The program constants used in the data flow program are read in Procedure Initialize_Match. For each statement which has a constant as input, the token store must reflect the fact that a constant is present by setting the constant indicator to true, and a token for that constant must be created and a pointer to the token stored. All other statements which do not have a constant as input are initialized to have the constant indicator false, and the pointer to a constant token is set to nil.

The start constants are read in procedure Get_the_ball_Rolling. For each constant, the procedure creates a packet for its destination instruction containing the constant as a data token. It then enters this packet into the Fetch Queue, which will cause execution of the destination instruction.

5.4.3. Match Module

The Match Module, file "match.e", contains all the procedures required to update and maintain the data structure "tokenstore" in the Match Unit (refer to figure 5.7). These procedures are called only from the Match Unit in the main module and they are always called in the context of a particular instruction.

Procedure Enter_all_into_FQ is called when the newly arrived data token has caused the enabling count of the instruction to go to zero. The procedure takes the data token along with the other token(s) already stored in tokenstore for instruction m, and enters them as one packet in the Fetch Queue. It then must remove the iteration number record from token store. If there are no other iteration numbers entered under that CID, it must also remove the CID record.

Procedure Find_CID is called to find an incoming data token's CID in token store; i.e. to see if there is an entry under the CID of the token. It is given the code block number of an incoming data token and looks in the tokenstore for an entry

for that particular CID. If there is one, a pointer to the list of IID records is returned along with a flag set to true; otherwise the flag is set to false, meaning the CID is not in token store.

Procedure Find_IID is called when a new token has arrived for which a CID record has already been found in token store. It now wants to know if there is already an entry under the IID of the token. The procedure is given the iteration number of the incoming data token and a pointer to the correct CID record and it looks in token store for an entry under the IID of the token. If one is found, a pointer to the correct IID record is returned along with a flag set to true; otherwise the flag is set to false.

Procedure Add_to_tokenstore simply enters a token into the token store under the correct CID and IID. The CID and IID records may have to be created at this point.

Procedure Enter_IID is called when a token has arrived for which it has already been established that there is no IID record in the list matching the token's IID; i.e. this is the first token to arrive with this particular IID. The procedure is given an incoming data token and a pointer to the list of IID records under the correct CID. A record for the IID of the token is entered in the list and, a pointer to the token is entered in the IID record.

Procedure Enter_CIDIID is called when it has been established that there is no CID record in token store which matches the CID of an incoming token. (Therefore there also is no IID record.) CID and IID records are entered into token store along with a pointer to the data token under the appropriate instruction.

5.4.4. Proc Module

The Proc module, file "proc.e", contains all the procedures for executing the data flow machine language statements. The execution of a statement (with the exception of the halt and output statements) produces one or more data tokens which are then sent in a packet to the Match Queue.

If, during the execution of an instruction, a data flow error condition is detected, a flag is set which stops execution of any more instructions. With no instructions being executed, the production of data tokens is stopped. Once the Match Unit clears all packets out of the Match Queue produced by previously executed instructions, all action is stopped due to the lack of packets in the system.

In Euclid, this is seen as the blocking of all processes (Match Unit, Fetch Unit, and processors). This condition is reported to the user, who then must press the delete key to end the program.

Error conditions detected by the procedures executing instructions are as follows:

(1) Tokens are not of the type expected. For example, the instruction was coded to expect two integer inputs and instead received two boolean input tokens.

(2) A token is missing for one or more input ports. There probably were multiple tokens sent to a single port.

(3) Received more than one token for a port (non-fatal error for output statement).

(4) Trying to print other than character or integer data in output statement.

(5) No instruction address token received by a begin statement, or no context control packet received by an end statement.

The above are all considered fatal error conditions which stop execution except error 3 for an output statement. In this case, only one of the values will be printed; the other is lost.

5.4.5. Monitors

In addition to the modules described, the data flow simulator contains five monitors. Monitors provide a convenient means for guaranteeing mutual exclusion to a portion of the program and data, and for blocking and waking up processes. They are included in Concurrent Euclid, the language in which the

simulator is written. A complete description of the operation and use of monitors is found in [Holt 1983].

The first monitor, the Qmgr monitor, manages the buffer which actually holds the packets in the Match and Fetch queues. The FQmgr, MQmgr and BufferMgr monitors are tied together in their usage to maintain the queues. They are based on the concept of producer/consumer pairs. A more detailed description of their interaction is found in [Holt 1983]. The Process_Control monitor controls the creation of processors.

5.4.5.1. Qmgr Monitor

The Qmgr monitor, file "Qmgr.e", makes additions and deletions from the pool of buffers which the Match and Fetch Queues reference. As mentioned before in section 5.3.3, these queues actually share one single data structure, declared in this monitor to be "queue" of type Q (an array of records). The only access allowed to "queue" is through this monitor; no other procedures can access the data structure.

5.4.5.2. FQmgr Monitor

The FQmgr monitor, file "FQmgr.e", is used to enter and remove packets from the Fetch Queue. Refer to section 5.3.3 for a complete description of the data structures making up the Fetch Queue.

Procedure FEnter accepts a buffer number of the shared array "queue" from the Match Unit and enters that number in the array "Fbuffer", which is an array of integers managed as a queue. By entering a buffer number into this queue, a packet has been entered into the Fetch Queue. The actual packet must have been entered into the array "queue" before the procedure FEnter was called, through use of the monitor Qmgr.

Procedure FRemove removes the first buffer number from the head of the Fetch queue (i.e. the array "Fbuffer") and passes it back to the Fetch Unit, effectively removing a packet from the queue. A copy of the information in the array "queue" must have been acquired before calling FRemove.

5.4.5.3. MQmgr Monitor

The MQmgr monitor, file "MQmgr.e", works in exactly the same way as the FQmgr monitor, except that it enters packets into the Match Queue for the processors using procedure MEnter and removes packets for the Match Unit using procedure MRemove.

5.4.5.4. BufferMgr Monitor

The BufferMgr monitor, file "bufmgr.e", contains the Acquire and Release procedures which are called by the producers and consumers of the Match and Fetch queues.

To enter a packet into the Match Queue, a processor must first acquire a free buffer number in the array "queue" into

which to put the packet information. The consumer of the Match Queue, the Match Unit, releases the buffer number after it has removed a packet from the queue. A similar situation holds for the Fetch Queue.

The list of available buffer locations is kept in an array "pool", which is a stack.

5.4.5.5. Process_Control Monitor

The Process_Control monitor, file "pcntrl.e", is the monitor which controls the creation of processors when they are needed. Procedure Spawn creates a processor; procedure Slp puts a processor back to sleep when it has finished executing an instruction. It works in the following manner.

When the simulation program begins to run, the processes in the main module which work as the processors of the computer are started and immediately call procedure Body_of_Processor in the Proc module. This procedure immediately enters a loop and calls procedure Slp, which puts the calling process to sleep on a "wait" queue as a generic processor, waiting to be put to work.

When the Fetch queue receives a packet of data tokens and the number of an instruction which is now to be executed, it calls procedure Spawn with the opcode of the instruction, a copy of the instruction itself, and the packet. This call results in waking up a process and taking it off the "wait" queue. It is

given the opcode, instruction and packet, and returns to the procedure Body_of_Processor in the Proc module, where it has now become a processor for the specific op code involved. The instruction is then executed through procedures in the Proc module.

5.5. Trace Feature

In the case of execution error in a data flow program running on the simulator, a trace capability has been built into the simulation program to aid in finding where the error occurred.

The first line of any data flow program must contain the string "trace" if a trace is desired, or "notrace" if it is not. If "notrace" is selected, only the program output is written to standard output; i.e. that output produced from an "output" statement.

However, if "trace" is selected, the user is given a running commentary of packets taken from the Match and Fetch Queues, along with their destination instruction numbers. In this way, the user can trace the number of packets produced for any instruction, and which instructions have been enabled and passed on to the Fetch Unit and processors. Then if an error does occur, the user can pinpoint which data flow instruction caused the error.

An example of a program trace is shown in Appendix E.

5.6. The Data Flow Language

The language executed by the simulator includes the following operators:

```
+ - * /
absolute value
negate (unary)
modulo
logical and, or, not
if <, <=, >, >=, =, /=
input
output
L, L1, D, D1 (used for loops)
T-gate, F-gate, switch
halt
activate, terminate, begin, end (for subprograms)
```

The graphical representation of these operators is shown in section 5.6.1. A data flow program would first be written in the graphical form, then written into a file in mnemonic form (section 5.6.2) for input into an assembler. The assembler would produce a machine language file (section 5.6.3), which is then read and executed directly by the simulator.

The assembler has not been written at this point; it stands as a project to be completed at some future time. It is a fairly straightforward translation from mnemonics to machine code; the translation is specified in section 5.6.5.

For purposes of testing the simulator, programs have been written in graphical form and directly translated into machine code. The advantage of using the assembler will be a built-in

check for certain programming errors, and error-free translation.

5.6.1. Graph Language

This section outlines the graphical representation of each operator that has been implemented on the simulator. (Note that the term "operator" in graph language is synonymous with the term "instruction" in the written mnemonic or machine language form.) Input and output ports are shown numbered in cases where there is more than one port and the order of inputs and/or outputs is critical to the correct execution of the program. The direction of data flow on the arcs is indicated by arrows.

There is no "duplicate" instruction; if it is desired to duplicate a token at several different locations, the token is sent to multiple destinations directly from the instruction that produced it. However, in data flow graph diagrams, the duplicate operator (a small circle) will still be shown.

The symbol X also will be used in graph diagrams, although there is no corresponding operator. This symbol was used in [Arvind, Gostelow and Plouffe 1978] and represents a legal merging of two lines where only one of the two lines will actually receive a value.

Constants are represented as a number in a circle being directed to an operator. The value of a program constant is always available to the instruction; it has to wait only for the

non-constant data token to arrive for the instruction to be enabled. The exception is a "start constant" which is a starting condition for the program; i.e. it triggers the start of execu-tuion, and it is the only input value required by an instruction (see section 5.1). Start constants are not legal in subprograms, only in the main program.

5.6.1.1. Arithmetic Operators

The arithmetic operators are addition, subtraction, multiplication, truncating integer divide, modulo, absolute value and unary negate (see figure 5.8).

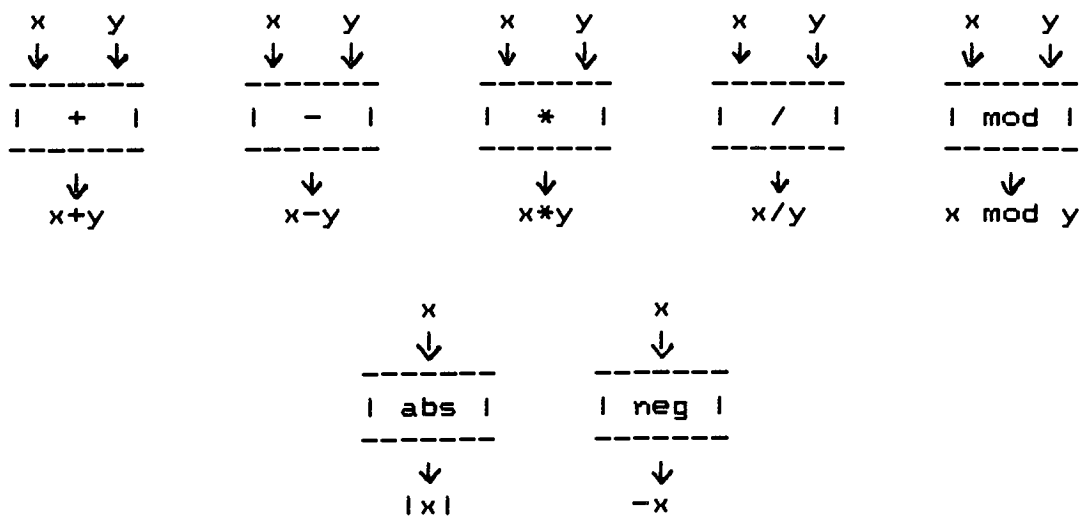


Figure 5.8 Arithmetic operators

Input:

+, -, *, /, mod	:	Two integer tokens, one each into port 1 and port 2.
abs, neg	:	One integer token into port 1.

Operation:

+, -, *, /, mod	:	(port1) operator (port2)
abs, neg	:	operator (port 1)

The integer result of the arithmetic operation leaves from output port 1.

There are no built-in checks for common arithmetic errors such as exceeding maxint, going below minint, and division by zero. If these errors occur, they will be handled by the computer as usual.

Enabling Count:

Context Control:		0
Data:	+, -, *, /, mod	2
	abs, neg	1

5.6.1.2. Logical Operators

The logical operators are "and", "or", and "not" (see figure 5.9).

Input: and, or: Two boolean tokens, one each into ports 1 and 2.
not: One boolean token into port 1.

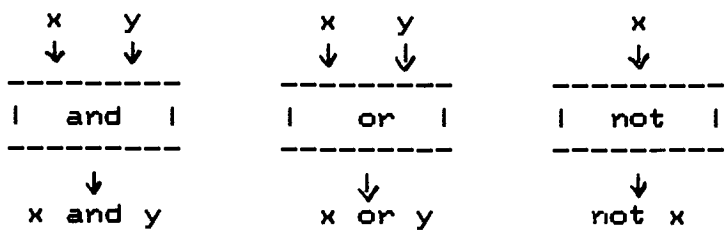


Figure 5.9 Logical operators

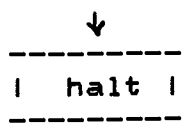
Operation:

and, or: (port 1) operator (port 2)
not: not (port 1)

Enabling Count:

Context control: 0
Data:
and, or 2
not 1

5.6.1.3. Halt



Input: One token of any type. Type is not checked.

Operation: The input token is absorbed and no output is produced. This instruction is actually a "sink" for a line of logic in a program which is finished but yet produces a token which must be sent somewhere. See Appendix D for an example of its use.

Enabling Count:

Context control:	0
Data:	1

5.6.1.4. Decider Operators

The Decider Operators make a decision based on a predicate p. The possible values of p are <, <=, >, >=, = and /= (see figure 5.10).

Inputs: Two integer tokens, one each into ports 1 and 2.

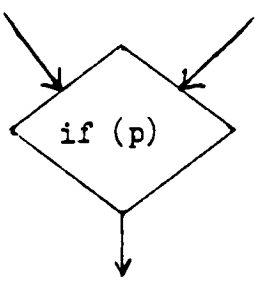


Figure 5.10 Decider operators

Operation: The integer inputs are compared according to the predicate of the operator. One boolean value is the result of the comparison, and it leaves on output port 1.

Enabling Count:

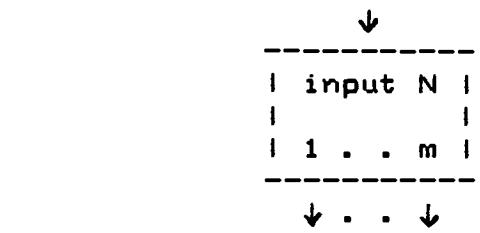
Context control:	0
Data:	2

5.6.1.5. Input

The input operator will read up to m values from file number N (see figure 5.11). Values read can be integer or character. The user can read from a maximum of five different files; the file number must be between 1 and 5. If more than one input statement specifies a certain file number, there is no guarantee as to the order in which the lines are read from the file, due to the asynchronous and non-sequential nature of a data flow computer.

Input: One token of any type. The value has no significance and is not looked at; it is merely a trigger to the execution of the input statement.

Operation: The input token triggers the execution of the instruction, which causes m integer values to be read from program argument file N. Program argument files are file names which are listed in the simulator run command after the file name which contains the data flow program. For example, in the following



where m = the number of values to be read (1 <= m <= 20)
N = the file number (1 <= N <= 5).

Figure 5.11 Input Statement

statement:

```
% dfsim <programfile datafile1 datafile2
```

datafile1 defines file 1 and datafile2 defines file 2. Refer to Appendix C for an example of a program using data files.

The first value read from the input file is sent from output port 1, the second value from port 2, etc. That is, each value read has its own destination; they are not sent as a group to any one destination.

Enabling Count:

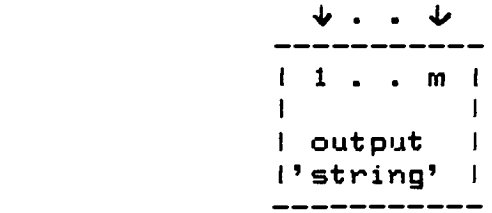
Context control: 0
Data: 1

5.6.1.6. Output

The output operator will cause the value of at most 20 tokens to be printed on standard output, along with an identifying string of characters (see Figure 5.12). Values can be integer or character.

Input: From 1 to m integer or character tokens (m <= 20) arrive on input ports 1 to m.

Operation: When input tokens have arrived on all m input arcs, the operator is ready to execute. The string, which is included as part of the instruction, is printed on standard output. Then the values of all the tokens which were received at the input ports are printed in order of input port number on standard output. The values are printed on the same line as the string. Integers are printed in a field of 10 spaces; characters in a field of one space with no blanks on either side. No tokens are



where 1 <= m <= 20
string <= 20 characters.

Figure 5.12 Output operator

produced on any output arcs.

Enabling Count:

Context control: 0
Data: m

5.6.1.7. Gate if true, Gate if false

These operators either pass on or absorb the input token, depending on the value of a boolean control token (refer to figure 5.13).

Input: One integer or character data token into port 1; one boolean token into port 2.

Operation:

Gate if true: If the boolean value is true, the data token is passed on from output port 1. If the boolean value is false,

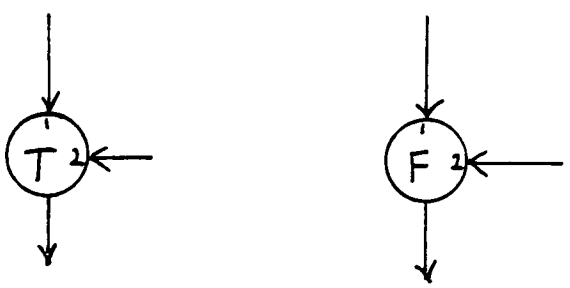


Figure 5.13 Gate if true, Gate if false

the data token is absorbed and no output token is produced.

Gate if false: If the boolean value is false, the data token is passed on from output port 1. If the boolean value is true, the token is absorbed and no output token is produced.

Enabling Count:

Context control:	0
Data:	2

5.6.1.8. Switch

The switch operator causes data to flow from one output port or the other depending on a boolean control value (see figure 5.14).

Input: One integer or character token into port 1; one boolean token into port 2.

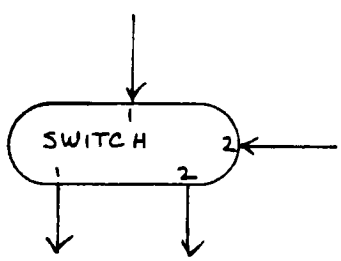


Figure 5.14 Switch Operator

Operation: If the input boolean token is true, then the data token from input port 1 is sent from output port 1; otherwise it is sent from output port 2.

Enabling Count:

Context control: 0
Data: 2

5.6.1.9. Loops

Four operators are required to implement a loop: L, L1, D and D1 [Arvind and Gostelow 1982] (see figure 5.15).

L Operator: This operator is required at the entry point to a loop; it creates a new context for execution by giving the tag of an input token a unique code block identifier and sets the iteration number to 1.

Input: One boolean, integer or character input token with tag (C, i).

Operation: The input token is passed on from output port 2 to its destination(s) with a new tag (C', 1), C' being the unique CID dynamically assigned to the loop just entered. The IID = 1 since this is the first iteration of the loop.

A context control token is sent from output port 1 to the L1 statement. The tag of the token is (C', 1); the old tag (C, i) is carried as the data value.

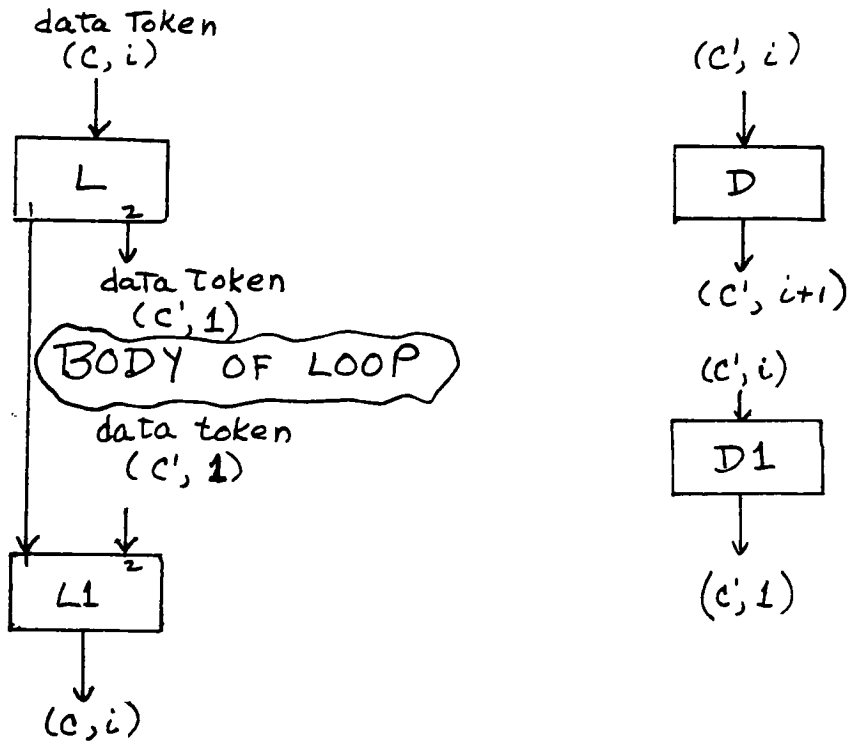


Figure 5.15 Loop operators: L, L1, D, D1

Enabling Count:

Context control: Ø
Data: 1

L1 Operator: This operator is required at the exit point of a loop. It restores the tag of the data token to the value it had on entry to the loop, which was (C, i).

Input:

Port 1: Context control token with new context as tag value and old context stored as data value of the token.

Port 2: One boolean, integer or character data token with tag of the new context.

Operation: The context control token is used only to restore the tag of the data token to that of the old context. The context control token is absorbed and not passed on. The data token from input port 2 is passed on from output port 1 to its destination(s) with the tag of the old context.

Enabling Count:

Context control: 1
Data: 1

D Operator: The iteration count of a token in a loop has to be incremented every time the token goes around the loop. The D operator accomplishes this.

Input: One boolean, integer or character data token with tag (C', i).

Operation: The input token is passed on from output port 1 with its data value unchanged but with its iteration number incremented by one. In this way, tokens arriving at an instruction within the loop but for different iterations can be matched up correctly.

Enabling Count:

Context Control: 0
Data : 1

D1 Operator: This operator is required at the exit point of a loop, immediately before the L1 instruction. It never receives more than one token for any instantiation of a loop.

Input: One boolean, integer or character data token with tag (C', i)

Operation: The input token is sent from output port 1 to its destination instruction (L1) with the iteration field of its tag set to 1. Everything else remains the same.

Enabling Count:

Context control: 0
Data: 1

An example of a loop and the use of the L, L1, D and D1 operators is shown in figure 5.16. The code block and iteration numbers are shown in parentheses. (C, I) is the context as the token enters the loop. C' is the new code block number for the loop domain; 2 is the last iteration of the loop.

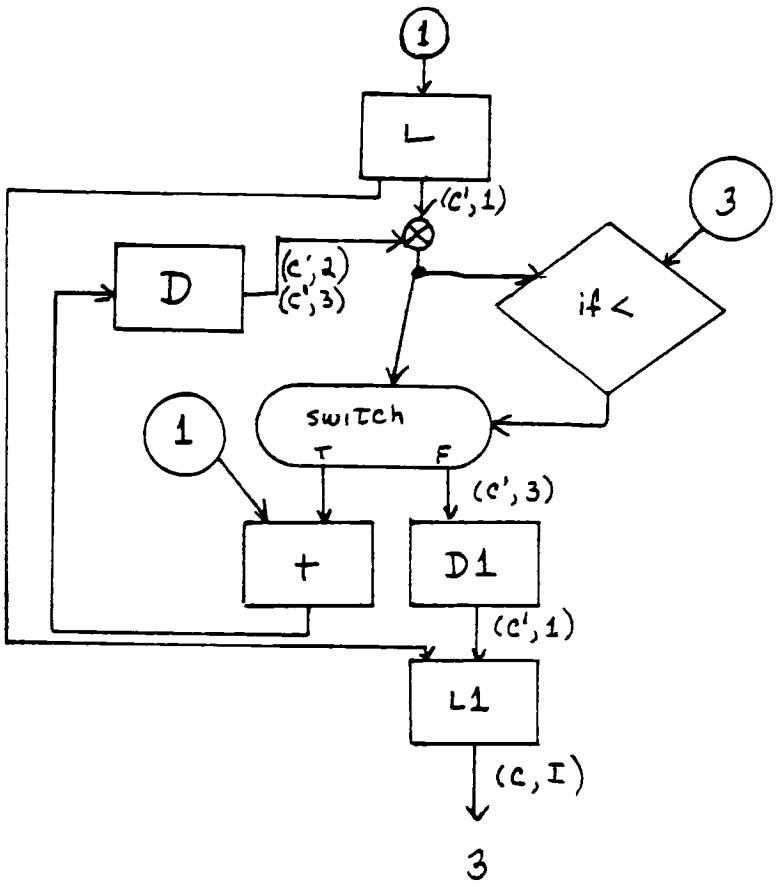


Figure 5.16 Loop with tagged tokens

5.6.1.10. Apply Operator and Subprograms

The apply operator represents a subprogram call. Referring to figure 5.17, a "symbolic" token carrying the name of the subprogram to be executed is shown being sent to the apply operator. This token is called symbolic because in the implementation of apply in the simulator, there is no actual token used for this purpose. The n input parameters to the subprogram arrive in

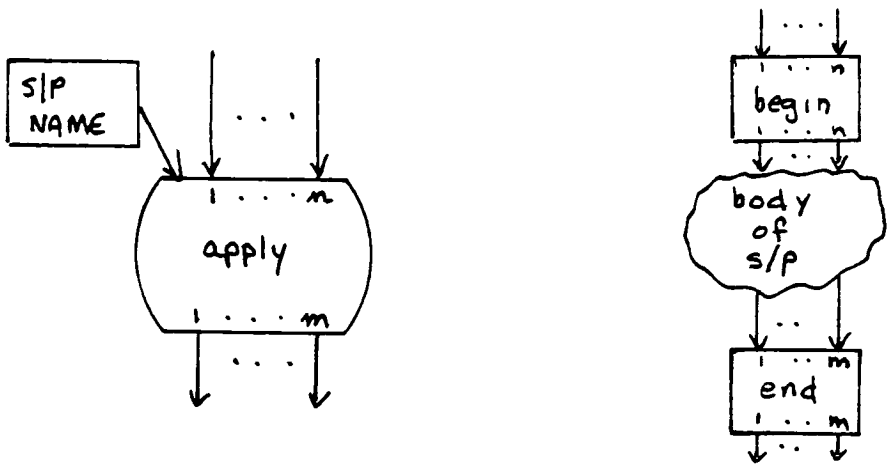


Figure 5.17 Apply Operator and Subprograms

tokens at input ports 1 to n. There must be at least one input parameter to a subprogram, if only to trigger its execution, and at most 19 parameters. Input parameters can be of type boolean, integer or character.

The first statement of a subprogram is the begin operator, with all input parameters to the subprogram being shown arriving at ports 1 to n. The begin statement then sends the tokens to the appropriate operators in the subprogram. All resulting output values from the subprogram are directed to the end operator, from which point they are assumed to be directed back to the calling program.

This is a rather simplified picture of what is actually happening in the application of a subprogram. However, it is an adequate representation for writing graph programs, and a more detailed description is deferred to section 5.6.3 on Machine Language.

5.6.1.11. Completeness of Graph Language

With the instruction set described above, it is possible to represent all control flow concepts which have been included in high level languages, including high level data flow languages such as ID [Arvind, Gostelow and Phouffe 1978]. The conditional schema:

```
if x then y else z
```

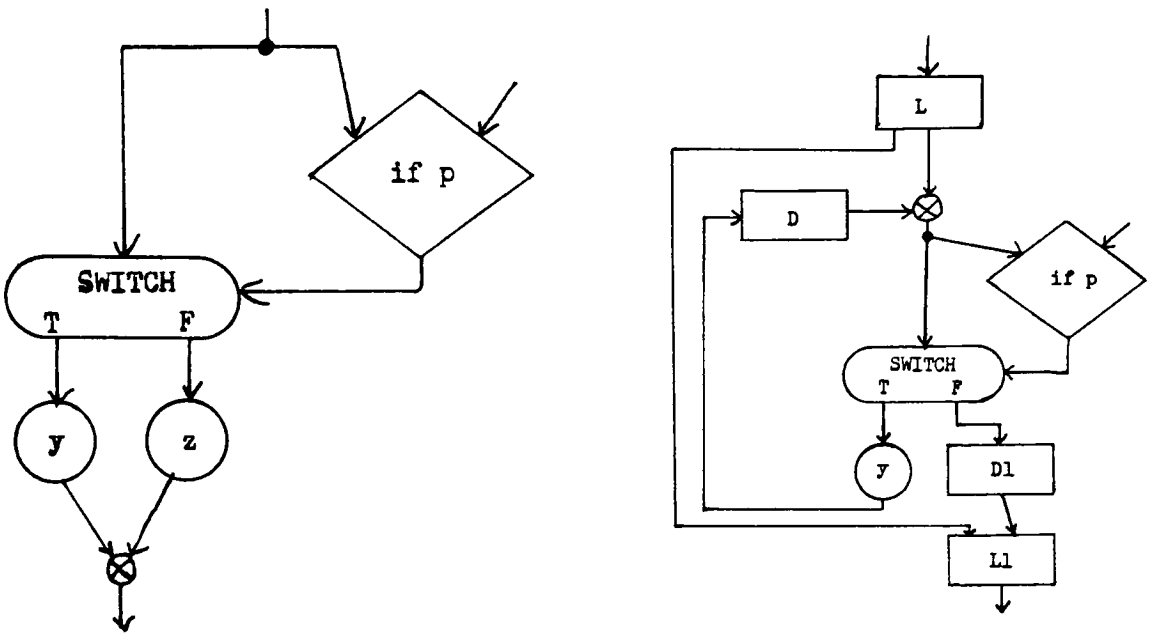
is shown in a data flow graph in figure 5.18. The while schema:

```
while x do
   y
```

is also shown in figure 5.18. A loop can be expressed in a while schema:

```
for i = 1 to 10 do          same as        i = 1
     f(x)                                  while i <= 10 do
                                           f(x)
```

and a repeat statement is a while schema with the test at the bottom of the loop instead of at the top. A case statement can be expressed as a series or nest of conditionals.



Conditional

While Schema

Figure 5.18 Conditional and While Schemas

5.6.2. Mnemonic Language

Suggestions for a possible mnemonic language are briefly outlined in table 5.1. The mnemonic language is the statement form of a graph program; i.e. the form in which a program would be entered into a file for subsequent input into an assembler.

In the headings shown in the table, "# copies" refers to the number of copies required of the token produced by the execution of the statement. In this way, duplicate tokens are sent to their destinations without the use of a duplicate operator. A destination instruction number and input arc number must be specified for each copy required.

Brackets { } on both sides of a list mean that one item from the list is to be used. Square brackets [ ] mean the items inside are optional. One bracket } with a number to the right indicates the number of items in the list enclosed by the bracket.

Instr Nbr	Opcode	Nbr of Inputs	File Nbr	Nbr of Outputs	Print String	Data Type	Nbr of Copies	Dest Inst, Arc
n	+						m	i, J k, 1 . . m
	-							
	*							
	/							
	mod							
	abs							
	neg							
	and							
	or							
	not							
n	input	I	J			t1	m1	i1, J1 k1, 11 . . m1
						t1	m2	i2, J2 k2, 12 . . m2
						. .		I
n	output			I	' . . . '	t1		I
						t1		I
						. .		
n	if <					t1	m	i, J k, 1 . . m
	if <=							
	if >							
	if >=							
	if =							
	if /=							
n	switch					t2	m1(true)	i1, J1 k1, 11 . . m1
							m2(false)	i2, J2 k2, 12 . . m2

Table 5.1. Mnemonic Language.

n	Tgate Fgate		t2	m	i, J k, l . .		m
n	L L1 D		t2	m	i, J k, l . .		m
n	D1		t2	1	p, 2		
n	halt						
n	apply name	i	t2 . .			i	
			t2 . .			J	
@	S/P name						
1	begin	i	t2 . .			i	
n	end	J	t2 . .			J	

where name = name of the subprogram
p = number of L1 instruction
t1 = int, char
t2 = int, char, bool

Table 5.1. Mnemonic Language (cont'd).

5.6.2.1. Form of Mnemonic Program

The first statement in any program is specification of a trace. If a trace is desired, the first line should read "trace"; otherwise "notrace".

The statements in the main program are numbered consequtively from one with one exception. The statement number following the apply statement must be two greater than the apply statement number. That is, if the apply statement number is n, then n+1 is unused and the next statement is number n+2. The main program statements immediately follow the trace/notrace statement. Constants used in the main program are specified immediately after the numbered statements, in the same way they are specified for machine language programs (see Chapter 5, section 6.4).

Following the main program are the subprograms. The statements in each subprogram are numbered from 0, starting with the S/P statement. The second statement of a subprogram is the begin statement; the last numbered statement is the end statement. (Note there is no end statement for the main program.) Following the numbered statements of each subprogram, the constant statements for that subprogram are specified.

Refer to appendix C for an example of a program in mnemonic form.

5.6.3. Machine Language Statements

The form for machine language statements is as follows:

instr #	opcode	Enab. Cnt CC	Enab. Cnt data	File nbr or print string	input types	nbr of dests
-----	------	--	----	----------	-----	-----

(1) Instruction number. In the machine language program, statements must be numbered consequtively from one, with no numbers skipped. Subprograms do not start over from one; the statement numbers start with the next available number.

(2) Opcode. The opcode as listed in Appendix A.

(3) Enabling count. Two separate numbers are required, the first for context control tokens and the second for data tokens.

(4) File number or print string. This field applies only to input and output statements. For an input statement, the file number from which data is to be read should be shown here. For an output statement, the string of characters to be printed with the output values should be written here, enclosed in single quotes. The string inside the quotes must be a maximum of 20 characters.

(5) Input types. Twenty separate numbers are required here, whether or not the statement can legally use that many input

tokens. The numbers correspond to the input port numbers of the statements; i.e. the first input type corresponds to port 1, the second to port 2, etc. The exception to this is the input statement. The types for an input statement refer to the type of the value which is to be read from the file. Codes for input types are shown in Appendix B.

(6) Number of destinations. One integer is shown here, representing the total number of destinations for all tokens from all output ports of the instruction.

Entries in fields of a machine language statement must be complete, separated by at least one blank, and in the correct order. Other than that, the input can be spaced out in any form and spread out over any number of lines. Refer to Appendix C for specification of all machine language statements.

The specifications for the destinations of output tokens produced by a statement follow the statement to which they apply. They are entered in the following form:

output port	type of data	instr. nbr.	input port

(1) Output port. The number of the output port from which the token is produced. This number will be one in most cases. The exceptions are:

switch Tokens are produced from port 1 in the true case, port 2 if false.

input The first data value read goes out from port 1, the second from port 2, etc.

output The number of destinations is zero since no tokens are produced by this statement.

L The context control token leaves from port 1, the data token from port 2.

halt The number of destinations is zero.

terminate,

end These statements use as many output ports as there are parameters. e.g. parameter 1 leaves from port 1, parameter 2 from port 2, etc.

activate,

begin These statements use one more output port than the number of parameters. The first output port is for either the instruction address or context control token, the rest are for the parameters.

(2) Type of data. The code for the type of the data which leaves by this output port.

(3) Instruction number. The destination instruction number.

(4) Input port. The port number through which the token should enter the destination instruction.

As with the machine language statements, the entries in the above fields need only be separated by one blank. The destination information for any one statement must, however, be ordered by output port. Duplicate tokens can be sent to different instructions by specifying the same output port and type but different instruction numbers and input ports, in any order.

There is no machine language statement for a constant. Program constants for the main program and subprograms are listed together in one section immediately after the program, and start constants for the main program and subprograms are listed together after the program constants. the program.

The apply operator for subprograms was introduced in a previous section on graph language. In machine language, the function of the apply operator is actually accomplished by two separate operators: activate and terminate. (The terminate operator must immediately follow activate in machine code.) The activate, terminate, begin and end operators all work together to allow tokens to be operated upon in the new context of the subprogram (and with a new tag) and then to return to the correct old context, thereby allowing more than one instantiation of a subprogram at one time. The interaction of these operators is

shown in figure 5.19.

The n tokens which are input parameters to the subprogram are the input to the activate operator. The activate operator then sends a special token to the begin operator of the subprogram, along with the tokens which are the input to the subprogram. This special token is an "instruction address" token, which is sent from output port 1. It contains as its data value the address (statement number) of the terminate statement. The n input tokens to the subprogram are sent from output ports 2 to (n+1); i.e. the token which arrived at port 1 leaves from port 2, and so on. These tokens are all sent with the same tag that they had when they arrived at the activate statement. There must be at least one input token to a subprogram since without it the execution of the subprogram would not be triggered. There can be at most 19 input parameters.

The begin statement, upon receiving these tokens which are sent all in one packet, changes the context to allow execution of the subprogram under its own unique context. The instruction address token is changed to a context control token, the tag is changed to the new tag, and the old tag as well as the address of the terminate statement are both sent as data values to the end statement from output port one. Input data tokens (the parameters) are sent to their destinations in the body of the subprogram with the new tag.

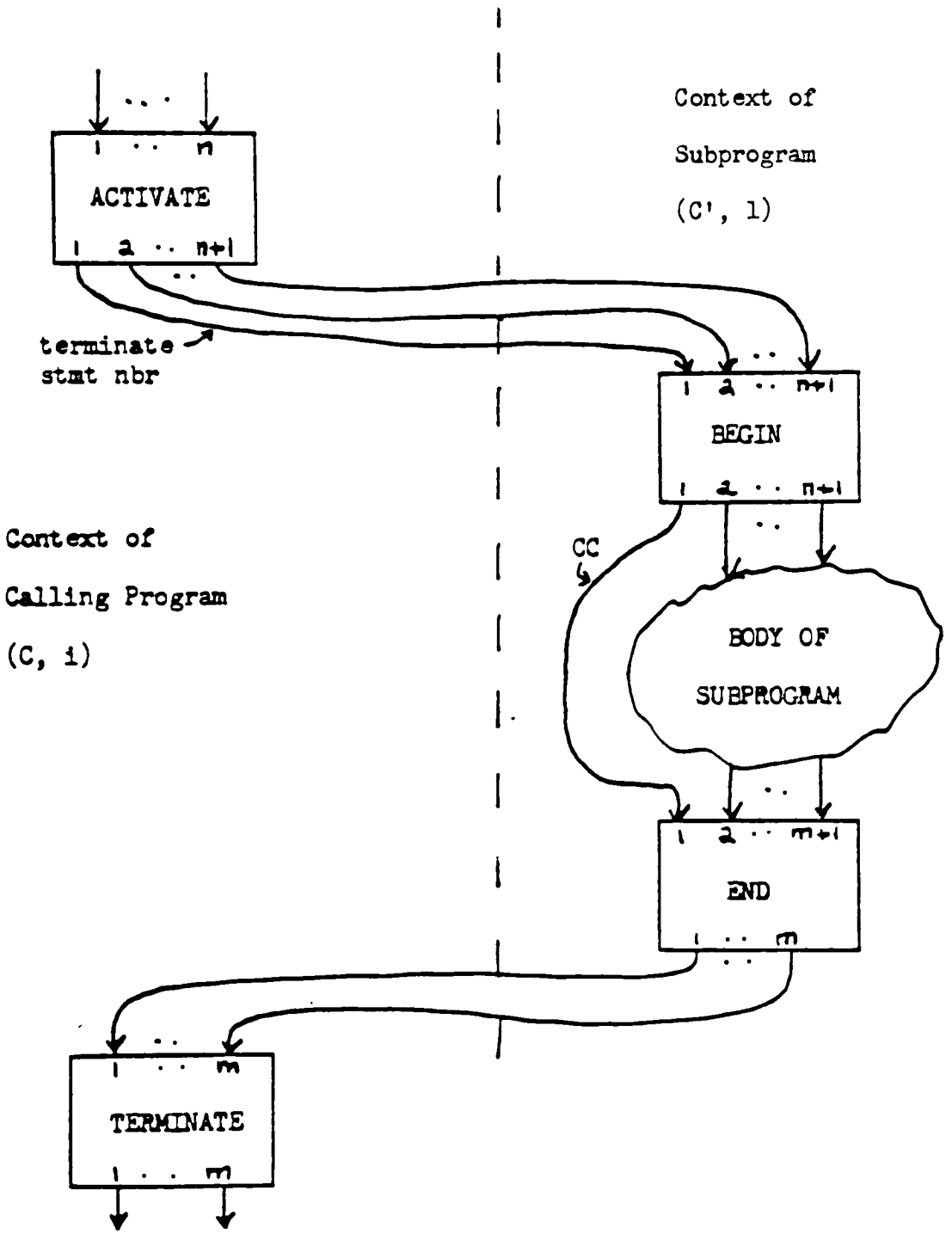


Figure 5.19 Subprogram activation

The end operator waits to receive the context control token and y data tokens with the same tag as the context control token. When they are all available, the tags of the data tokens are changed to the old tag which has been saved in the context control token. Then they are all sent in one packet to the address of the terminate operator which is also contained in the context control token. The data tokens are sent from a port number which is one less than the port number at which they arrived; i.e., a data token which arrived at input port 2 is sent from output port 1.

The terminate operator is enabled upon receiving this one packet containing all the data tokens. This operator has all the information as to where these returned values are to be sent in the calling program; they are sent in separate packets to their respective destination instruction(s).

The enabling counts for these four instructions is summed up in table 5.2.

Refer to Appendix C for an example of a program written in machine language.

5.6.4. The Machine Language Program

The machine language program must be entered into a file in a specific form, as outlined below.

Operator	Enabling Count: Context Control	Enabling Count: Data
activate	0	x
terminate	0	1
begin	0	1
end	1	y

Table 5.2. Enabling counts of subprogram operators.

(1) line 1: The specification of the trace feature: "trace" or "notrace"

(2) line 2: One integer specifying the number of machine language statements in the program. Since statements are numbered consequtively, this number will be the same as the statement number of the last statement in the program, including subprograms.

(3) starting at line 3: the machine language statements, immediately followed by the destination information for each statement.

(4) An integer specifying the number of unique program constants used throughout the program.

(5) Specification of program constants and their destinations, as follows:

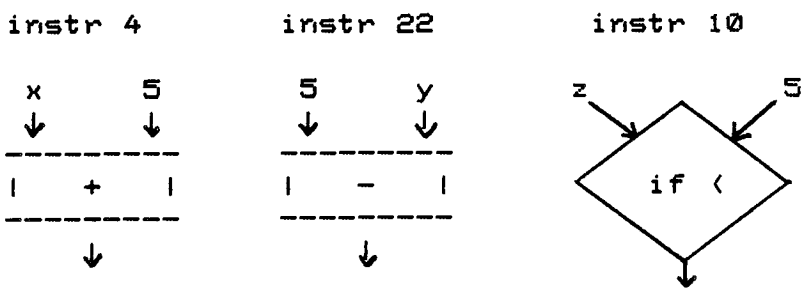
value of constant	nbr of destinations	instruction	arc	
N	m	i	J	m lines
		k	l	
		.	.	
		.	.	

As shown, one constant can be used as input to several instructions. Refer to the example in figure 5.20.

(6) An integer specifying the number of start constants in the program.

(7) Specification of start constants and their destinations. They are entered in the same form as program constants, in (5) above.

If a program contains these operators:



The constant 5 can be specified as follows:

```
5  3  4  2
      10  2
      22  1
```

Figure 5.20 Specification of constants

Refer to Appendix C for an example of a complete program.

5.6.5. The Assembler

As stated earlier, the assembler would read a file containing a data flow program in mnemonic form and produce a file containing the program in machine language form, which is directly executable by the simulator.

Translation of mnemonic statements into machine language is fairly straightforward and is done line for line, with the exception of the apply and subprogram statements. Mnemonic opcodes and types are translated to numeric codes, and the total number of copies specified becomes the number of destinations.

The translation is best done in two passes. On the first pass, the "S/P" statements are dropped, and subprogram statements, beginning with the "begin" statement, are renumbered using a relocation factor (subprogram statement number + number of last statement before the S/P statement). All references (destination specifications) within the subprogram are changed in the same way. A symbol table is created with the name of the subprogram and the number of its begin statement.

On the second pass, apply operators are translated into two operators: the "activate" and "terminate" operators (the reason for leaving the statement number following "apply" unused). The activate statement looks up the subprogram name in the symbol

table to get the statement number to use for its destination information.

Constants specified for the main program and subprograms must be combined into one group in the machine language program. One way to do this would be to enter all constants and their destination instruction numbers in a table on the first pass using relocated instruction numbers for the subprograms. There would be one entry for each unique constant, with all the destinations specified for that constant throughout the program. After the second pass, the information in the table could be printed out in the correct form for constant specification. Start constants would, of course, have to be kept separate from program constants and printed in a separate section after the program constants.

An example of a translation from mnemonics to machine language is shown in Appendix C.

CHAPTER 6

CONCLUSIONS

Programming the simulator in Euclid has worked out well in many ways. For example, the "pseudo-concurrency" of Euclid has allowed an accurate simulation of how a data flow computer might actually operate. However, it has had its drawbacks, one being the lack of helpful diagnostics when a run-time error occurs. Also, the fact that files are not closed after a run-time error results in the lack of readable diagnostic information where output from various processes is written into separate files. This was partially overcome by the use of the "trace" feature, which allows a user of the simulator to watch execution of the program and know exactly which data flow instructions generated the error.

The fact that reals are not implemented in Euclid and therefore in the simulator and that integers have a maximum value of 32767 has been a limiting factor in the use and testing of the simulator. Many "real life" applications which would have been interesting due to their high degree of concurrency could not be programmed because of these limitations; e.g., integration of a function f by the trapezoidal rule [Arvind and Gostelow 1982].

The next immediate step to aid in the use of the data flow simulator is to write the assembler to convert mnemonic programs

to machine language; this would simplify the writing of test programs.

There is also work which could be done in the area of implementing data structures. At the moment, only single-valued variables can be used as data tokens, which again limits the applications of the simulator. The system could be expanded to allow data tokens to carry a pointer to a whole structure, or to certain parts of it. Work in the area of I-structures [Arvind and Thomas 1980] would also be very valuable as far as future real-life applications of a data-flow system.

At the level of graph language, the capability to write a program on a CRT using graphic symbols and having that program automatically translated into machine language would be useful. Efforts could also be made to optimize data flow graphs before they are converted to machine language. Research is being done at Manchester on a higher level macro-assembly language.

Implementing compilers for sequential programming languages and high level data flow languages is also a subject under research today. A similar compiler could be written to produce machine language for this simulator.

Study could be given to the features required in a high level data flow language as far as the type of applications which would lend themselves best to the special capabilities of a data flow computer, such as resource management. A study of various

types of languages could also be undertaken, including nondeterministic, deterministic, functional, and single-assignment languages, and their implementation using a data flow computer.

BIBLIOGRAPHY

[Ackerman 1982]

Ackerman, W. B. "Data Flow Languages," Computer 15 (February 1982), 15-25.

[Agerwala and Arvind 1982]

Agerwala, T., and Arvind. "Data Flow Systems," Computer 15 (February 1982), 10-13.

[Arvind and Gostelow 1975]

Arvind and Gostelow, K.P. "A New Interpreter for Dataflow Schemas and its Implications for Computer Architecture," Technical Report 72, Department of Information and Computer Science, University of California, Irvine, October 1975.

[Arvind and Gostelow 1977]

Arvind and Gostelow, K. P. "A Computer Capable of Exchanging Processors for Time," Proceedings IFIP Congress (1977), 849-854.

[Arvind and Gostelow 1978]

Arvind and Gostelow, K. P. "Dataflow Computer Architecture: Research and Goals," Technical Report 113, Department of Information and Computer Science, University of California, Irvine, February 6, 1978.

[Arvind and Gostelow 1982]

Arvind and Gostelow, K. P. "The U-Interpreter," Computer 15 (February 1982), 42-49.

[Arvind, Gostelow and Plouffe 1978]

Arvind, Gostelow, K. P., and Plouffe, W. "An Asynchronous Programming Language and Computing Machine," Technical Report 114a, Department of Information and Computer Science, University of California, Irvine, December 1978.

[Arvind and Kathail 1981]

Arvind and Kathail, V. "A Multiple Processor Data Flow Machine That Supports Generalized Procedures," Eighth Annual Symposium on Computer Architecture, Minneapolis, Mn., 12-14 May 1981 (New York: IEEE 1981), 291-302.

[Arvind and Thomas 1980]

Arvind and Thomas, R. E. "I-Structures: An Efficient Data Type for Functional Languages," Report MIT/LCS/TM-210, Laboratory for Computer Science, M.I.T., September 1980.

[Backus 1978]

Backus, J. "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," Communications of the ACM 21, 8 (August 1978), 613-641.

[Clark 1973]

Clark, B. "A Speed-Independent Implementation of Data Flow Schemas," Computation Structures Group Memo 82, Project MAC, M.I.T., June 1973.

[Davis 1978a]

Davis, A. L. "The Architecture and System Method of

DDM1: A recursively Structured Data Driven Machine," Proceedings 5th Annual Symposium Computer Architecture (Palo Alto, California, April 3-5) ACM, New York, 1978, 210-215.

[Davis 1978b]

Davis, A. L. "Data Driven Nets: A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems," Technical Report UUCS-78-108, Department of Computer Science, University of Utah, July 1978.

[Davis 1979]

Davis, A. L. "A Data Flow Evaluation System Based on the Concept of Recursive Locality," Proceedings 1979 National Computer Conference (New York, New York, June 4-7), volume 48, AFIPS Press, Arlington, Va., 1979, 1079-1086.

[Davis and Keller 1982]

Davis, A. L., and Keller, R. M. "Data Flow Program Graphs," Computer 15 (February 1982), 26-41.

[Dennis 1974]

Dennis, J. B. "On Storage Management for Advanced Programming Languages," Computation Structures Group Memo 109-1, Project MAC, M.I.T., October 1974 (revised November 1, 1974).

[Dennis 1975]

Dennis, J. B. "First Version of a Data Flow Procedure

Language," MAC Technical Memorandum 61, Project MAC, M.I.T., May 1975.

[Dennis 1977]

Dennis, J. B. "A Language Design for Structured Concurrency," Computation Structures Note 28-1, Laboratory for Computer Science, M.I.T., February 1977.

[Dennis 1979]

Dennis, J. B., "The Varieties of Data Flow Computers," Proceedings First International Conference Distributed Computing Systems (Toulouse, France, October 1979), 430-439.

[Dennis 1980]

Dennis, J. B. "Data Flow Supercomputers," Computer 13 (November 1980), 48-56.

[Dennis, Boughton, and Leung 1980]

Dennis, J. B., Boughton, G. A., and Leung, C. K. C. "Building Blocks for Data Flow Prototypes," Seventh Annual Symposium on Computer Architecture Conference Proceedings, SIGARCH Newsletter Volume 8 Number 3, May 6-8, 1980, 1-8.

[Dennis and Misunas 1974]

Dennis, J. B., and Misunas, D. P. "A Computer Architecture for Highly Parallel Signal Processing," Computation Structures Group Memo 108, Project MAC, M.I.T., August 1974.

[Dennis and Misunas 1975]

Dennis, J.B., and Misunas, D. P. "A Preliminary Architecture for a Basic Data Flow Processor," Proceedings 2nd International Symposium Computer Architecture (Houston, Texas, January 20-22), IEEE, New York, 1975, 126-132.

[Dennis, Misunas and Leung 1977]

Dennis, J. B., Misunas, D. P., and Leung, C. K. "A Highly Parallel Processor Using a Data Flow Machine Language," Computation Structures Group Memo 134, Laboratory for Computer Science, M.I.T., January 1977.

[Friedman and Wise 1976]

Friedman, D. P., and Wise, D. S. "The Impact of Applicative Programming on Multiprocessing," Technical Report No. 52, Computer Science Department, Indiana University, Bloomington, July 1976.

[Gajski, Padua, Kuck and Kuhn 1982]

Gajski, D. D., Padua, D. A., Kuck, D. J., and Kuhn, R. H. "A Second Opinion on Data Flow Machines and Languages," Computer 15 (February 1982), 58-69.

[Gostelow and Thomas 1979]

Gostelow, K. P., and Thomas, R. E. "A View of Dataflow," Proceedings National Computer Conference (New York, New York, June 4-7), Volume 48, AFIPS Press, Arlington, Va., 1979, 629-636.

[Gurd and Watson 1977]

Gurd, J., and Watson, I., "A Multilayered Data Flow

Computer Architecture," Proceedings 1977 International Conference on Parallel Processing (August 1977), 94.

[Holt 1983]

Holt, R. C. Concurrent Euclid, the Unix System, and Tunis, Addison-Wesley Publishing Company, Inc., USA, 1983.

[Ho and Irani 1983]

Ho, L. Y., and Irani, K. B. "An Algorithm for Processor Allocation in a Dataflow Multiprocessing Environment," Proceedings 1983 International Conference on Parallel Processing, IEEE Computer Society Press, Silver Spring, Maryland, August 1983, 338-340.

[Hogenauer, Newbold and Inn 1982]

Hogenauer, E. B., Newbold, R. F. and Inn, Y. J. "DDSP - A Data Flow Computer for Signal Processing," Proceedings 1982 International Conference on Parallel Processing (August 1982), 126-133.

[Keller 1977]

Keller, R. M. "Semantics of Parallel Program Graphs," Technical Report UUCS-77-110, Department of Computer Science, University of Utah, July 1977.

[Keller 1980]

Keller, R. M. "Divide and CONCer: Data Structuring in Applicative Multiprocessing Systems," Proceedings Lisp Conference, August 1980, 196-202.

[Keller, Lindstrom and Patil 1978]

Keller, R. M., Lindstrom, G., and Patil, S. "An Architecture for a Loosely-Coupled Parallel Processor," Technical Report UUCS-78-105, Department of Computer Science, University of Utah, October 1978.

[Keller, Lindstrom and Patil 1979]

Keller, R. M., Lindstrom, G., and Patil, S. "A Loosely-Coupled Applicative Multi-Processing System," Proceedings National Computer Conference, AFIPS Press, New Jersey, 1979, 613-622.

[Keller and Yen 1981]

Keller, R. M., and Yen, W. J. "A Graphical Approach to Software Development Using Function Graphs," Digest of Papers Compcon Spring 81, February 1981, 156-161.

[Leler 1983]

Leler, W. "A Small, High-Speed Dataflow Processor," Proceedings 1983 International Conference on Parallel Processing, IEEE Computer Society Press, Silver Spring, Maryland, August 1983, 341-343.

[Lerner 1984]

Lerner, E. J. "Data Flow Architecture," IEEE Spectrum, April 1984, 57-62.

[Leung 1975]

Leung, C. K. C. "Formal Properties of Well-Formed Data Flow Schemas," MAC Technical Memorandum 66, Project MAC, M.I.T., June 1975.

[Litvin 1983]

Litvin, Y. "Top Down Data Flow Programming," Proceedings 1983 International Conference on Parallel Processing, IEEE Computer Society Press, Silver Spring, Maryland, August 1983, 252-254.

[Myers 1982]

Myers, G. J. Advances in Computer Architecture, John Wiley & Sons, New York, 1982.

[Plas, A. et al 1976]

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

[Rumbaugh 1975]

Rumbaugh, J. E. A Parallel Asynchronous Computer Architecture for Data Flow Programs, Ph. D. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., May 1975.

[Rumbaugh 1977]

Rumbaugh, J. E. "A Data Flow Multiprocessor," IEEE Transactions on Computers C-26, 2 (February 1977), 138-146.

[Schwartz 1980]

Schwartz, J. T. "Ultracomputers," ACM Transactions on Programming Languages and Systems, Vol. 2, No. 4, October 1980, 484-521.

[Sharp 1980]

Sharp, J. A. "Some Thoughts on Data Flow Architectures," Computer Architecture News 8 (15 June 1980), 11-21.

[Srini 1981]

Srini, V. P. "An Architecture for Extended Abstract Data Flow," Eighth Annual Symposium on Computer Architecture, Minneapolis, Mn., 12-14 May 1981 (New York: IEEE 1981), 303-325.

[Syre, Comte and Hifdi 1977]

Syre, J. C., Comte, D. and Hifdi, N. "Pipelining, Parallelism and Asynchronism in the LAU System," Proceedings 1977 International Conference on Parallel Processing (August 1977), 87-92.

[Tanenbaum 1981]

Tanenbaum, A. S. Computer Networks, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1981.

[Todd 1982]

Todd, K. W. "Function Sharing in a Static Data Flow Machine," Proceedings 1982 International Conference on Parallel Processing (August 1982), 137-139.

[Treleaven 1979]

Treleaven, P. C. "Exploiting Program Concurrency in Computing Systems," Computer 12, 1 (January 1979), 42-49.

[Treleaven 1980]

Treleaven, P. C. (Ed.) "VLSI: Machine Architecture and

Very High Level Languages," SIGARCH Computer Architecture News 8 (15 December 1980), 27-38.

[Treleaven 1983]

Treleaven, P. C. "The New Generation of Computer Architecture," Tenth Annual International Conference on Computer Architecture Conference Proceedings, Stockholm, Sweden, 13-16 June 1983 (New York: IEEE 1983), 402-409.

[Treleaven, Brownbridge and Hopkins 1982]

Treleaven, P. C., Brownbridge, D. R., and Hopkins, R. P. "Data Driven and Demand Driven Computer Architecture," Computing Surveys 14 (March 1982), 93-143.

[Watson and Gurd 1979]

Watson, I., and Gurd, J. "A Prototype Data Flow Computer with Token Labelling," Proceedings National Computer Conference, AFIPS Press, New Jersey, 1979, 623-628.

[Woo and Agrawala 1983]

Woo, N. S., and Agrawala, A. "The DC1 Flow Schema with the Data/Control-Driven Evaluation," Proceedings 1983 International Conference on Parallel Processing, IEEE Computer Society Press, Silver Spring, Maryland, August 1983, 244-251.

APPENDIX A

Opcodes

Operator	Code
abs	1
neg	2
input	3
not	4
halt	5
output	6
L	7
L1	8
D	9
D1	10
+	11
-	12
*	13
/	14
mod	15
and	16
or	17
Tgate	18
Fgate	19
switch	20
if <	21
if <=	22
if >	23
if >=	24
if =	25
if /=	26
begin	27
end	28
activate	29
terminate	30

APPENDIX B

Data Type Codes

Data Type	Code
no data	0
boolean	1
integer	2
character	3
instruction address	4
pointer	5
context control	7

Machine Language Statements

Mnemonic Opcode	Inst #	Opcode	Enabling CC	Count Data	File # or Print string	20 Input Types	Nbr of Destinations
+	n	11	0	2		2 2 0 . . 0	d
-	n	12	0	2		2 2 0 . . 0	d
/	n	13	0	2		2 2 0 . . 0	d
*	n	14	0	2		2 2 0 . . 0	d
abs	n	1	0	1		2 0 0 . . 0	d
neg	n	2	0	1		2 0 0 . . 0	d
mod	n	15	0	2		2 2 0 . . 0	d
input	n	33	0	1	J	t2 t2 t2 . . t2 (1)	d
output	n	6	0	i	'..'	t2 t2 t2 . . t2 (2)	0
and	n	16	0	2		1 1 0 . . 0	d
or	n	17	0	2		1 1 0 . . 0	d
not	n	4	0	1		1 0 0 . . 0	d
L	n	7	0	1		t1 0 0 . . 0	d
L1	n	8	1	1		7 t1 0 . . 0	d
D	n	9	0	1		t1 0 0 . . 0	d
D1	n	10	0	1		t1 0 0 . . 0	1
if <	n	21	0	2		t3 t3 0 . . 0 (3)	d
if <=	n	22	0	2		t3 t3 0 . . 0 (3)	d
if >	n	23	0	2		t3 t3 0 . . 0 (3)	d
if >=	n	24	0	2		t3 t3 0 . . 0 (3)	d
if =	n	25	0	2		t3 t3 0 . . 0 (3)	d
if /=	n	26	0	2		t3 t3 0 . . 0 (3)	d
halt	n	5	0	1		t1 0 0 . . 0	0
switch	n	20	0	2		t1 1 0 . . 0	d (4)
Tgate	n	18	0	2		t1 1 0 . . 0	d
Fgate	n	19	0	2		t1 1 0 . . 0	d
activate	m	29	0	k		t4 t4 t4 . t4 0 (5)	d
terminate	m+1	30	0	1		t4 t4 t4 . t4 0 (6)	d
begin	n	27	0	1		4 t4 t4 . . t4 (7)	d
end	n	28	1	m		7 t4 t4 . . t4 (8)	d

where: n = instruction number
i = number of values to output
j = file number
k = number of parameters going to subprogram
m = number of parameters returned from subprogram
d = number of destinations
t1 = integer, character or boolean
t2 = integer, character or no data
t3 = integer or character
t4 = integer, character, boolean or no data

(1) Types are listed for as many variables as will be read from the file. The rest should be entered as Ø. Note that types refer to the variables to be read from a file, rather than actual input to the statement as is the case for the other opcodes. The actual input to the statement which triggers its execution can be of any type and its type is not noted in the statement.

(2) Types are listed for as many variables as will be printed in the output statement (same number as enabling count). The rest should be entered as Ø.

(3) Both types must be the same.

(4) There must be at least one destination specified for each output port. Destinations for output port 1 must be given before those for port 2.

(5) Types are listed for k variables which will be passed to the subprogram as parameters (1 (= k (= 19).

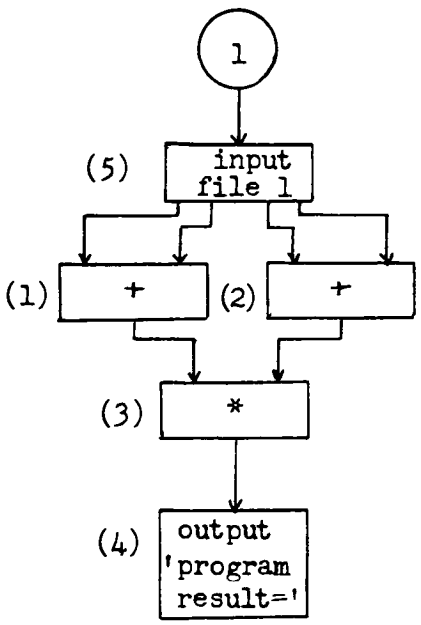
(6) Types are listed for m variables which will be passed back from the subpreogram (1 (= m (= 19). The enabling count will not match this number. The enabling count is 1 since all data tokens are sent in one packet to the terminate statement.

(7) Types are listed for k variables which will be passed to the subprogram as parameters. The number of types will not

equal the enabling count since all tokens are sent at once in one packet. First data type shown must be a 4.

(8) Types are listed for m variables which are passed back to the calling program, plus the first data type which refers to the context control token and must be a 7.

The graph program shown below reads four integers from one input file, multiplies the sum of the first and second by the sum of the third and fourth integers, and outputs the product along with the identifying comment "program result =". The numbers in parentheses to the left of each operator indicate the statement number in the mnemonic and machine language programs which will contain this particular operator.



This same graph program is shown below in mnemonic form:

```
notrace
1   +   1   3,1
2   +   1   3,2
3   *   1   4,1
4   output    1   'program result ='   int
5   input     4   1 int   1   1,1
                    int   1   1,2
                    int   1   2,1
                    int   1   2,2
1
1   1   5   1
```

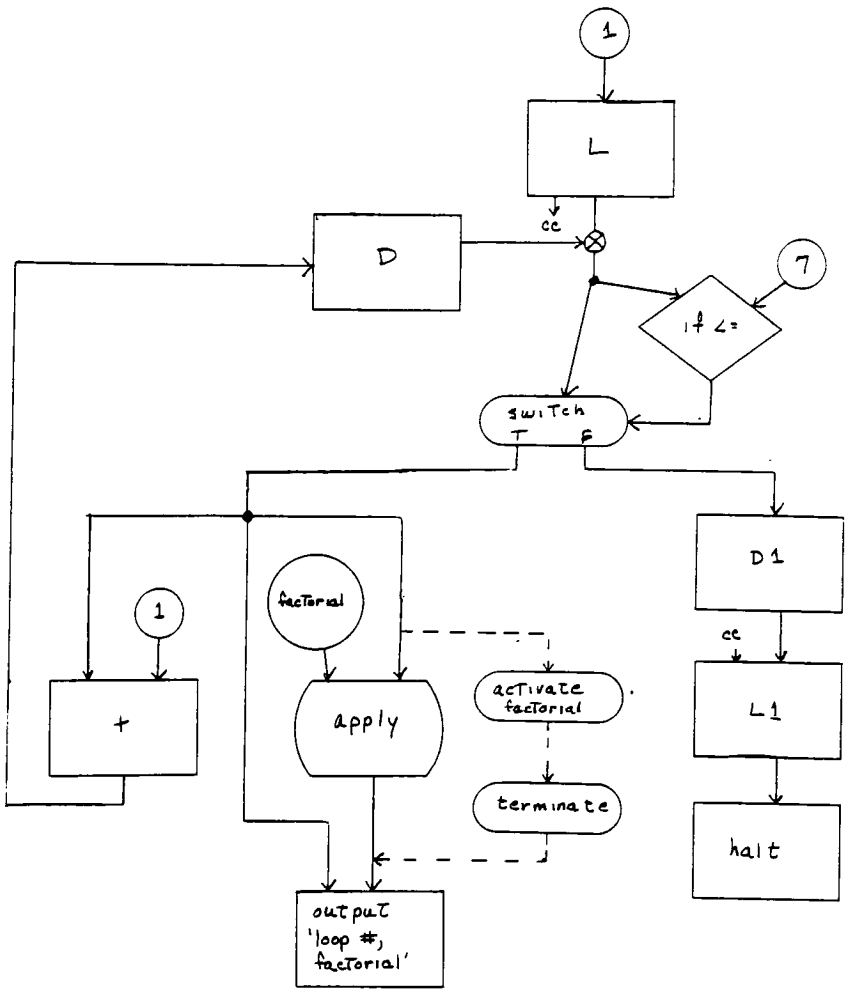
The same program is again shown below in its machine language form. This is the form that the simulator will accept for execution.

```
notrace
5
1 11 0 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 2 3 1
2 11 0 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 2 3 2
3 13 0 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 2 4 1
4 6 0 1 'program result =' 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5 3 0 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4
1 2 1 1
2 2 1 2
3 2 2 1
4 2 2 2
0
1
1 1 5 1
```

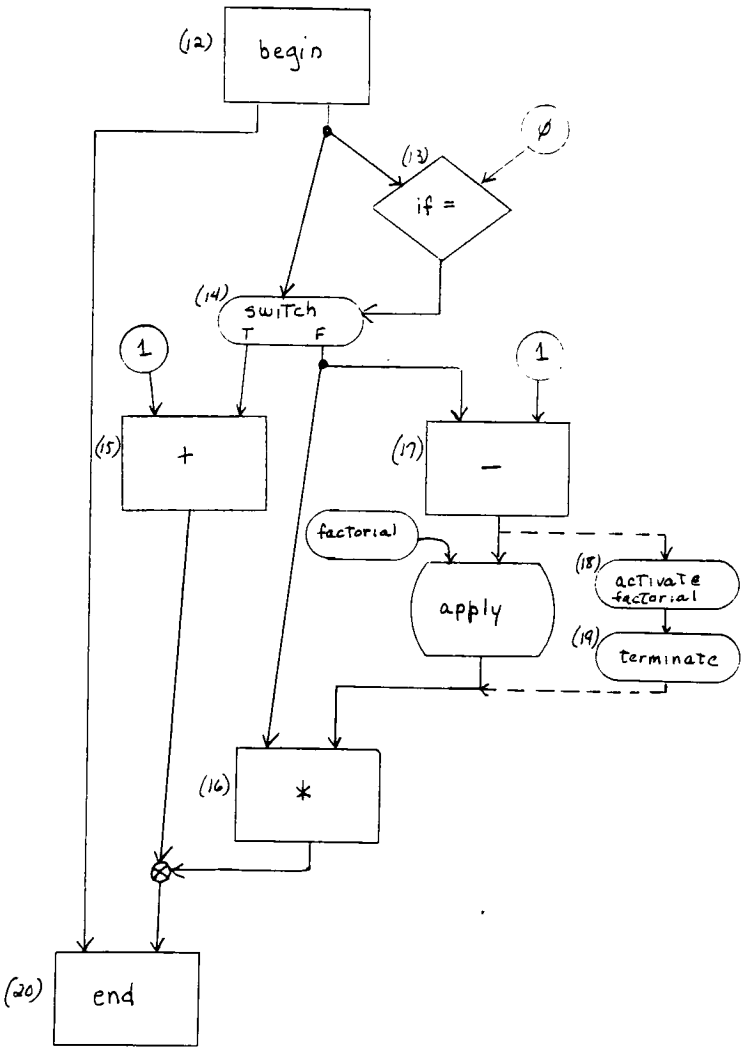
APPENDIX D

Graph Program to compute factorials from 1! to 7! using recursion.

Main Program



Factorial Subprogram



Machine Language form of the factorial program:

```
notrace
20
1 7 0 1     2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     3
1 7 7 1
2 2 5 1
2 2 2 1
2 22 0 2    2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 1 5 2
3 11 0 2    2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 2 4 1
4  9 0 1    2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     2
1 2 5 1
1 2 2 1
5 20 0 2    2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     4
1 2 3 1
1 2 9 1
1 2 10 1
2 2 6 1
6 10 0 1    2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 2 7 2
7  8 1 1    7 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 2 8 1
8 5 0 1     2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     0
9 6 0 2     'loop #, factorial:'  2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   0
10 29 0 1   2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     2
1 4 12 1
2 2 12 2
11 30 0 1   2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 2 9 2
12 27 0 1   4 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     3
1 7 20 1
2 2 14 1
2 2 13 1
13 25 0 2   2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 1 14 2
14 20 0 2   2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     3
1 2 15 2
2 2 16 1
2 2 17 1
15 11 0 2   2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 2 20 2
16 13 0 2   2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 2 20 2
17 12 0 2   2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 2 18 1
18 29 0 1   2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     2
1 4 12 1
2 2 12 2
19 30 0 1   2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 2 16 2
20  28 1 1  7 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1
1 2 0 1
3
7 1 2 2
0 1 13 2
1 3 15 1
    17 2
     3 2
1
1 1 1 1
```

Output of the factorial program:

```
Read the program.

Start execution.
loop #, factorial:          1          1
loop #, factorial:          2          2
loop #, factorial:          3          6
loop #, factorial:          4         24
loop #, factorial:          5        120
loop #, factorial:          6        720
loop #, factorial:          7       5040
```

APPENDIX E

Trace of the program shown in Appendix C:

```
Read the program.
Finished reading in program.
open file 1
Echo print of program:
1 11 0 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 1
2 11 0 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 2
3 13 0 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 4 1
4 6 0 1 program result = 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5 3 0 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2
1 2 1 1
2 2 1 2
3 2 2 1
4 2 2 2
End of Program.  Number of statements =  5
Constants:
Starting constants:
1 1 5 1
Start execution.
started p1
started p2
started p4
started p5
started p6
started p7
started p8
started p9
started process match
Started Fetch Unit
started p3
started p10
Match Unit:  Got packet for instruction      5
     Fetch Unit:  Got instruction     5
Match Unit:  Got packet for instruction      1
Match Unit:  Got packet for instruction      1
     Fetch Unit:  Got instruction     1
Match Unit:  Got packet for instruction      2
Match Unit:  Got packet for instruction      2
     Fetch Unit:  Got instruction     2
Match Unit:  Got packet for instruction      3
Match Unit:  Got packet for instruction      3
     Fetch Unit:  Got instruction     3
Match Unit:  Got packet for instruction      4
     Fetch Unit:  Got instruction     4
program result =          21
```

5.7. Appendix F

Error conditions detected by the procedures executing instructions are as follows:

(1) Tokens are not of the type expected. For example, the instruction was coded to expect two integer inputs and instead received two boolean input tokens.

(2) A token is missing for one or more input ports. There probably were multiple tokens sent to a single port.

(3) Received more than one token for a port (non-fatal error for output statement).

(4) Trying to print other than character or integer data in output statement.

(5) No instruction address token received by a begin statement, or no context control packet received by an end statement.

USER MANUAL

In using the simulator, the easiest way to go about preparing a program is to first write the program in graph language. Then each operator should be numbered starting from 1, remembering to skip one number after any apply statement. The operators can be ordered in any sequence; order is not important. What is important is that all destination information gives the correct destination operator number. The program can then be written in mnemonics or directly into machine language, and at this point, the statements must be listed in the order in which they were numbered.

The file containing the machine language program, say pgmname, is then specified in the run statement, followed by any data file names:

```
% dfsim <pgmname  datafile1 datafile2 ... datafile5
```

The output in this case will appear on the terminal. Output can be redirected to a file as follows:

```
% dfsim <pgmname datafile1 ... datafile5 >outputfile
```

If the program runs normally with no errors, the run will end with a message to the effect that all processes are blocked. This message will appear on the terminal whether or not program output is redirected to a file. If a run time error is encoun-

tered, however, the output files will not be closed and therefore are not accessible. All the user is given on the terminal is a very cryptic message such as "Bus error - core dumped" or "Memory Fault - core dumped". In that case, output should not be redirected into a file but sent directly to the terminal and the user will be able to see what output, if any, was produced before the error was encountered. The use of the trace function as described in section 5.5 is also extremely helpful in finding the program error and is the only way to watch execution statement by statement.