1988

Dataflow: Overview and simulation

Steven I. Benjamin

Rochester Institute of Technology
School of Computer Science and Technology

DATAFLOW: Overview and Simulation

by
Steven I. Benjamin

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: Dr. Andrew Kitchen 3/25/88
Date

Approved by: Dr. Lawrence Coon 3/25/88
Date

Approved by: Dr. Peter Lutz 3/25/88
Date

Dataflow: Overview and Simulation

I Steven I. Benjamin 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.

Abstract

The thesis project is a software simulation of the dataflow machine prototyped at the University of Manchester. It uses a dynamic token matching scheme based on the U-interpreter, and supports I-structures, an array-like data structure. An assembly language is provided for programming the simulator.

Table of Contents

1 Introduction 1

1.1 Data Flow vs. Control Flow 2

1.2 Previous Work 2

1.2.1 Dennis Machine at MIT 4

1.2.2 Arvind Machine at MIT 4

1.2.3 Manchester Machine 7

1.2.4 Utah DDM1 Machine 7

1.2.5 Toulouse LAU Machine 10

1.2.6 EDDY Machine 13

1.2.7 Other Projects 13

2 Project Description 15

2.1 Functional Specification 16

2.1.1 Program Representation 16

2.1.2 Data Representation 18

2.1.3 Structure Representation 18

2.1.4 Machine Organization 19

2.1.5 Instruction Set 19

3 Project Implementation 26

3.1 Assemble Instruction Process 28

3.2 Read Data Process 29

3.2.1 Token Implementation 29

3.3 Build Token Set Process 31

3.4 Build Operation Packet Process 34

3.5 Execute Operation Packet Process 34

3.5.1 I-Structure Implementation 35

3.6 Monitors and Process Synchronization 37

4 Project Application 40

4.1 Data Flow Graphs 40

4.1.1 Sequence Construct 41

4.1.2 Decision Construct 41

4.1.3 Repetition Construct 43

4.1.4 Subprograms 44

4.1.5 I-structures 46

4.2 Assembler Language 48

4.2.1 Operator Instructions 53

4.2.2 Predicate Instructions 54

4.2.3 Boolean Instructions 55

4.2.4 Branch Instructions 55

4.2.5 Loop Instructions 56

4.2.6 Procedure Instructions 57

4.2.7 I-structure Instructions 58

4.2.8 Output Instructions 58

4.2.9 Comment and Constant Instructions 59

4.3 Input File 59

4.4 Debugging 61

4.5 Sample Programs 63

4.5.1 Nested Loop Example 63

4.5.2 Recursive Factorial Example 70

4.5.3 Fibbonaci Series Example 73

4.5.4 Trapezoid Rule Example 77

4.5.5 I-Structure Example 81

4.5.6 Laplace Transform Example 85

4.6 Running the Simulator 94

4.7 Errors 95

5 Conclusion 96

Bibliography 98

List of Figures

Figure 1. Data Flow vs. Control Flow 3

Figure 2. Data Flow History 5

Figure 3. MIT Dennis Machine 6

Figure 4. MIT Arvind Machine 8

Figure 5. Manchester Machine 9

Figure 6. Utah DDM1 Machine 11

Figure 7. Toulouse Machine 12

Figure 8. EDDY Machine 14

Figure 9. Trapezoid Rule Compilation 17

Figure 10. Machine Organization 20

Figure 11. System Process Diagram 27

Figure 12. The config File 28

Figure 13. The commandTable File 30

Figure 14. Instruction Store Data Structure 31

Figure 15. Token Data Structure 32

Figure 16. Token Store Data Structure 33

Figure 17. Operation Packet Data Structure 34

Figure 18. Instruction Set Summary 35

Figure 19. I-structure Store Data Structure 36

Figure 20. Dataflow Graph Sequence Construct 41

Figure 21. Dataflow Graph Decision Construct 42

Figure 22. Dataflow Graph Repetition Construct 43

Figure 23. Dataflow Graph Subprogram 45

Figure 24. Dataflow Graph I-structure 47

Figure 25. Assembler Command Summary 48

Figure 26. Assembler Syntax Summary 50

Figure 27. Dataflow Graph and Assembler Program 52

Figure 28. Dataflow Graph and Input File 60

1. Introduction

This section is an introduction to data flow computers. Data flow and control flow computers are compared. A review of data flow computer projects is given.

There are two general approaches to making faster computer systems. The first approach is to use technology to make existing computer architectures faster. The second approach is to design new, faster architectures.

The prevalent existing architecture was developed by von Neuman and others over thirty years ago, it solved many engineering and programming problems that existed at that time. In its simplest form the von Neuman computer consists of three parts: a CPU (central processing unit), a memory unit, and a "tube" that transmits data between the two units. This tube has been called the "von Neuman bottleneck" by John Backus [Backus 1978].

The von Neuman bottleneck is both physical and conceptual. It is physical in that all changes to the memory unit can only be made by passing data one word at a time through the connecting tube. The bottleneck is conceptual in that most conventional programming languages have evolved to be high level versions of the von Neuman computer. This inhibits the natural expression of a given problem, instead a problem is expressed in terms of the underlying machine architecture.

The goal of many computer architects is to reduce the von Neuman bottleneck. The most common approach is to develop machines with several von Neuman processors, thus providing several connecting tubes, increasing the amount of data that can be passed between the CPU and memory at one time.

Other architects are eliminating the von Neuman bottleneck altogether. They are doing so by designing machines based upon models of computation that are altogether different from that of the von Neuman machine.

1.1. Data Flow vs. Control Flow

One novel architecture under study is that of the Data Flow machine. The data flow machine architecture is based upon the data flow model of computation. The data flow model is not a new idea, however, technology has only recently made it possible to consider computer architectures based upon this model.

The difference between the von Neuman, or "control flow" model and the data flow model, lies in what controls the process of computation within the individual models: [Miklosko, Kotov 1984]

Control Flow (CF) - it is the sequence of instructions.
Data Flow (DF) - it is the availability of data.

A CF model program is stored in memory as a serial sequence of instructions. Each instruction is fetched from memory and then executed in the processor (the von Neuman bottleneck). No instruction can execute until all previous instructions have executed. Thus, the process of computation is controlled by the sequence of instructions in the program. This is the main obstacle in exploiting the natural parallelism of algorithms.

In the DF computer, computation is controlled by the flow of data in the program. An instruction can execute only when all its operands are available. Whether an instruction precedes another depends upon the algorithm and not upon the location of the instructions in memory. Using this method of computation, it is possible to execute as many instructions in parallel as the given computer can simultaneously handle.

Refer to figure 1 for an illustration of the difference between CF and DF computation.

1.2. Previous Work

The originators of research on data driven computing can not be precisely defined. In 1968, Jack Dennis defined graphs to express algorithms by showing data dependencies only; Tesler and Enea published a report on single assignment programming languages. (Single assignment languages

Figure 1. Data Flow vs. Control Flow
[Ackerman 1982]

CONTROL FLOW

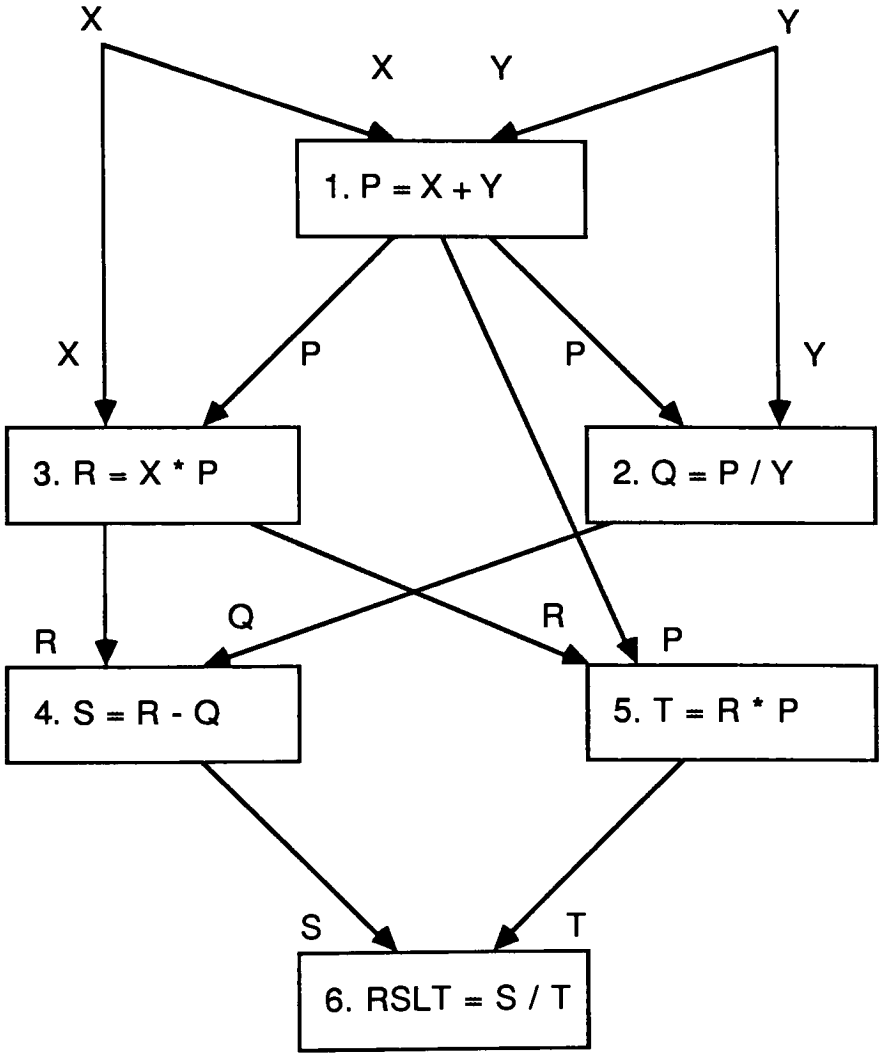
Sequence of Instructions	Computation Sequence
1. P = X + Y	1
2. Q = P / Y	2
3. R = X * P	3
4. S = R - Q	4
5. T = R * P	5
6.RSLT = S / T	6

DATA FLOW

Graph showing data dependencies

Computation Sequence

1
2 and 3
4 and 5
6



embody the syntactic and semantic features for data flow programming). Single assignment was developed by Klinkhamer and Chamberlin in 1971, and data flow graphs were developed at MIT by Misunas, Rumbaugh and Kosinski. Figure 2 shows some of the data flow research carried out since 1968 [Evans 1982]. Following is a survey of some current data flow projects.

1.2.1. Dennis Machine at MIT

Development is taking place at MIT by Dennis and Misunas. Project goals stated by Dennis, Second Data Flow Workshop [Misunas 1979]:

1. Develop user level programming language.
2. Build an engineering model.
3. Address translation, optimization, and code generation.
4. Develop specifications for full-scale machine.

Current project status [Hwang and Briggs 1984]:

1. Prototype hardware is under construction.
2. Compiler is being written for VAL programming language.
3. Fault tolerance studies are being made.

Figure 3 describes the Dennis machine architecture.

1.2.2. Arvind Machine at MIT

Development was begun at the University of California, Irvine and is continuing at MIT. It is being directed by Arvind and Gostelow. Project goals stated by Gostelow, Second Data Flow Workshop [Misunas 1979]:

1. Design general-purpose computer composed of many small processors.
2. Remove bottlenecks from the architecture.
3. Develop prototype based on ID programming language.
4. Investigate fault tolerance.

Current project status [Hwang, Briggs 1984]:

1. ID programming language has been developed.
2. The machine has not yet been built.
3. Extensive simulation studies on projected performance

Figure 2. Data Flow History

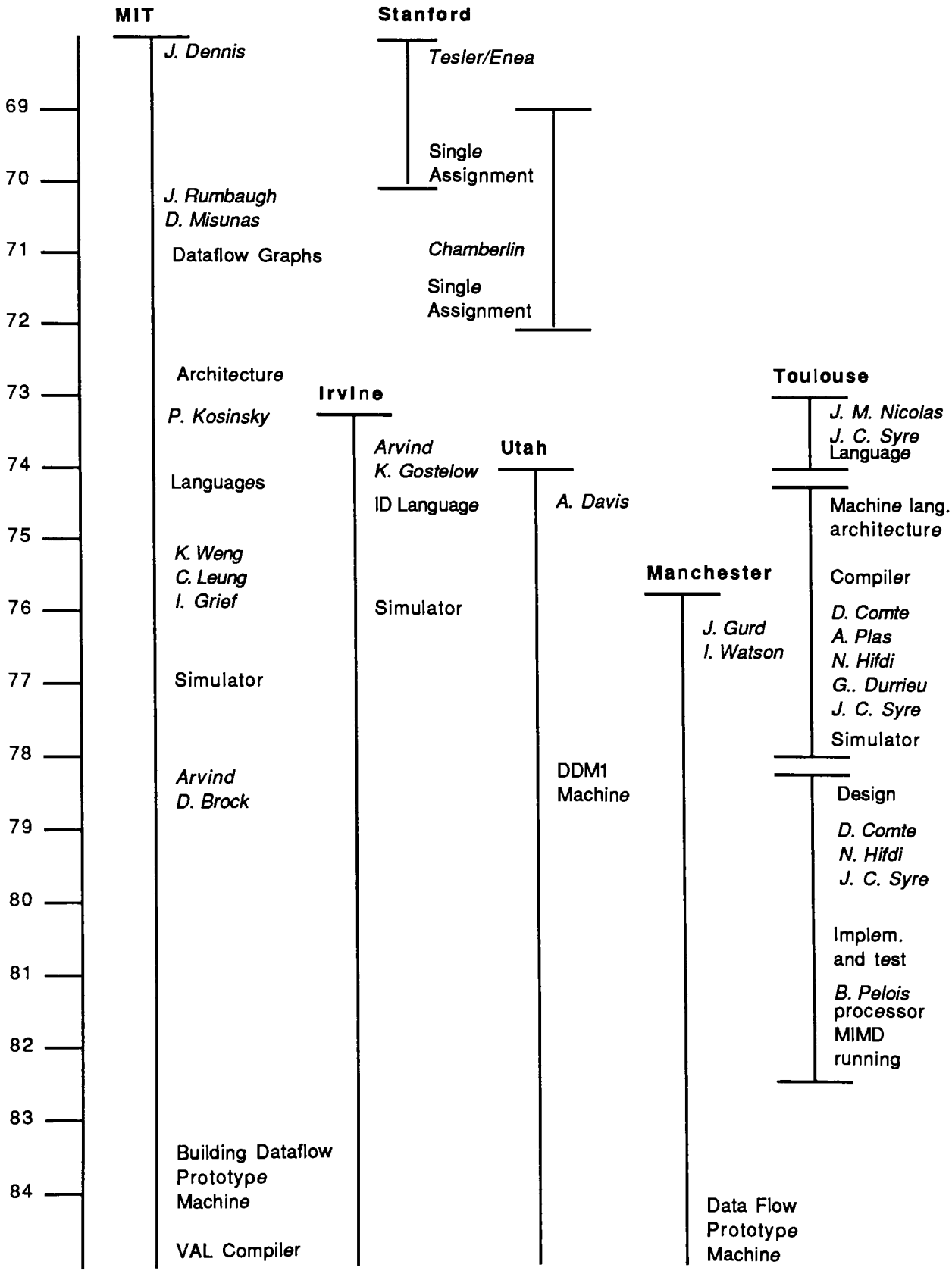
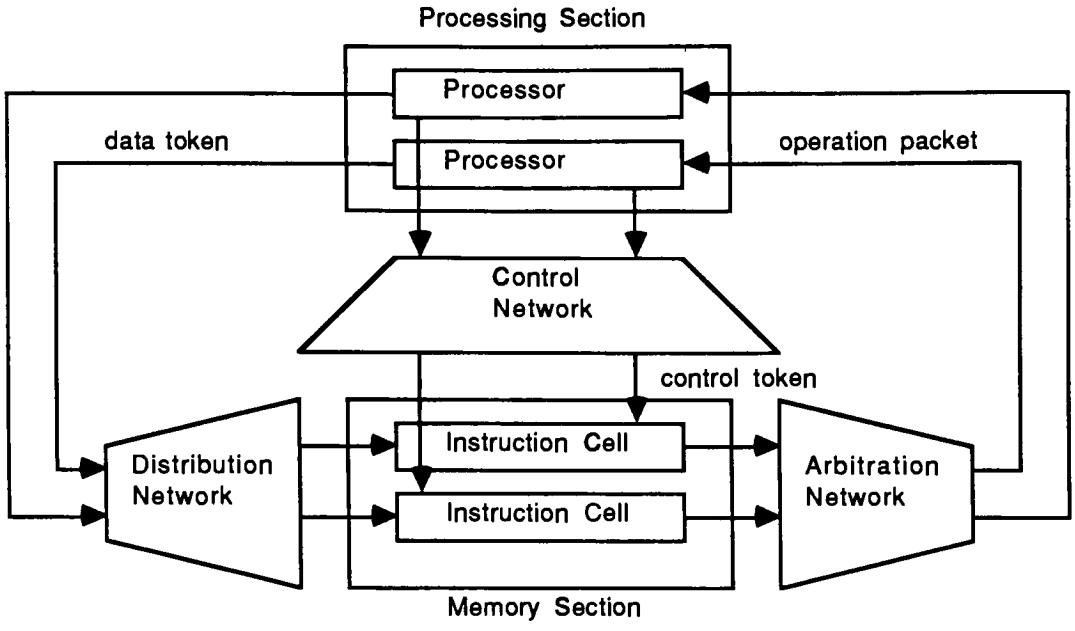


Figure 3. MIT Dennis Machine
[Misunas 1978] [Dennis,Boughton,Leung 1980]



Static Execution Rule: instruction is enabled (can be executed) if a data token is present on all its input arcs and no token is present on its output arcs

Data Token	- data traveling to the input arc of an instruction.
Control Token	- acknowledge signal indicating that data has been removed from an instruction's output arc.
Operation Packet	- enabled instruction and operands ready to be processed.
Memory Section	- instruction cells which hold instructions and their operands.
Processing Section	- processing units that perform functional operations on data tokens.
Arbitration Network	- delivers operation packets from memory section to processing section.
Control Network	- delivers control tokens from processing section to memory section.
Distribution Network	- delivers data tokens from processing section to memory section.

have been done.

Figure 4 describes the Arvind machine architecture.

1.2.3. Manchester Machine

Development is taking place at Manchester, England. It is being directed by Gurd, Watson, and Kirkham. Project goals stated by Watson, Second Data Flow Workshop [Misunas 1979]:

1. Major motivation is the exploitation of parallelism to develop high speed machine.
2. Secondary motivation is realization of cost effective and reliable design.

Current project status [Gurd, Kirkham, Watson 1985]:

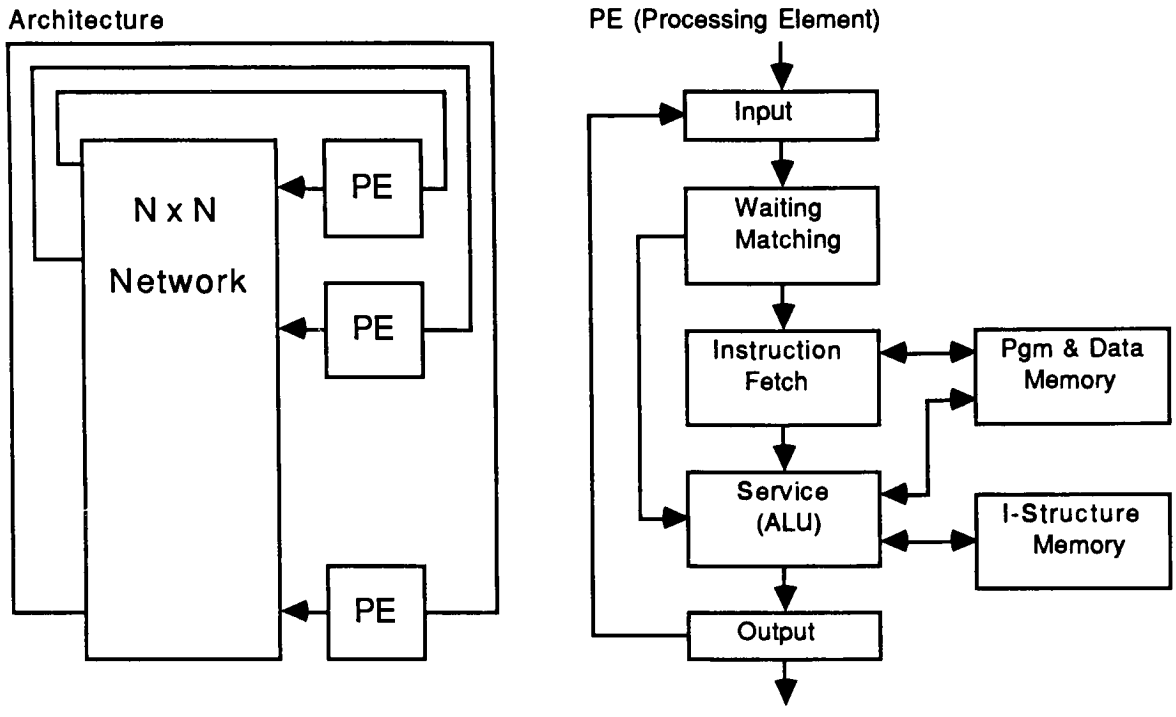
1. A data flow machine has been constructed large enough to tackle realistic applications.
2. A small range of benchmark programs has been written and executed.
3. Preliminary evaluation results are as follows:
   a. a wide variety of programs contain sufficient parallelism to exhibit speedup.
   b. a useful indicator of program parallelism has been established.
   c. a weakness in the present pipeline implementation has been identified and fixed.
   d. the need for a separate structure storage has been indicated.
4. Future studies:
   a. build and evaluate a multi-ring architecture.
   b. investigate programs that cause match-unit overflow.
   c. study low-level code optimization.
   d. study data flow implementation using VLSI technology.
   e. improve machine to exceed performance of VAX 11/780 mini-computer.

Figure 5 describes the Manchester machine architecture.

1.2.4. Utah DDM1 Machine

Development was begun at Burroughs. It is currently based at Utah University. Davis is directing the project. Project goals stated by Davis, Second Data Flow Workshop [Misunas 1979]:

Figure 4. MIT Arvind Machine
[Arvind,Kathail 1981] [Arvind,Gostelow 1977]



The Arvind machine does not follow the static execution rule as does the Dennis Machine (see figure 3). It instead allows several tokens to be present on the input and output arcs that lead into and out of an instruction. In this way several instantiations of the same instruction (provided there are no data constraints) can execute concurrently. The architecture of the Arvind machine is said to be dynamic.

Input Section

Accepts tokens from either the communication system or the output section of its same PE.

Waiting-Matching Section

Input tokens whose destination activity requires two operands are sent to this section and are buffered until they can be matched. When matched, the token set is then sent to the instruction-fetch section.

Instruction-Fetch Section

Combines an instruction's opcode with its operands, and sends the resulting operation packet to the service section.

Service Section

Processes the operation packet; sends the resulting data token to the output section.

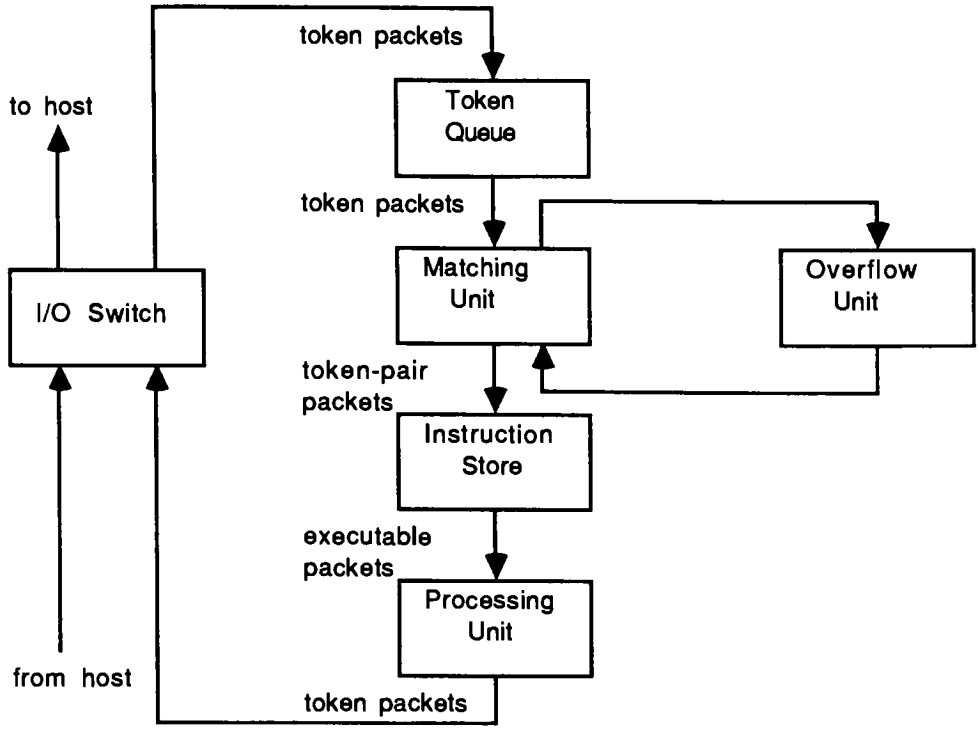
I-Structure Memory

Stores I-structure tokens. I-Structures are an array-like data structure [Arvind,Thomas 1980].

Output Section

Delivers tokens to either the communication system or the input section of its same PE.

Figure 5. Manchester Machine
[Gurd,Kirkham,Watson 1985] [Gurd,Watson 1977] [Watson,Gurd 1979] [Watson,Gurd 1982]



The Manchester machine is a dynamic dataflow machine, as is the MIT Arvind machine (see figure 4).

I/O Switch	- loads programs and data from host; permits results to be output for external inspection.
Token Queue	- smooths out uneven rates of generation and consumption of tokens in the ring.
Matching Unit	- pairs together tokens destined for the same activity.
Instruction Store	- contains machine code of dataflow program.
Processing Unit	- processes the executable packets.

1. Develop a recursive machine architecture.
   a. performance gain is made as machine is physically extended.
   b. no need for electronic tuning as hardware modules are added.
2. Develop a high-level data-driven graphical language [Davis, Lowder 1981].

Current project status [Treleaven, Brownbridge ,Hopkins 1982]:

1. DDM1 became operational in 1976 (first in the USA). The DDM1 communicates with a DEC 20/40. The DEC system is used for compilation, input, output, and performance measurement.
2. The language is currently a statement description of a directed graph. An interactive graphical programming language is under development.

Figure 6 describes the DDM1 machine architecture.

1.2.5. Toulouse LAU Machine

Development is taking place in Toulouse, France by Plas, Comte, Syre, Hifdi. Project goals stated by Comte, Second Data Flow Workshop [Misunas 1979]:

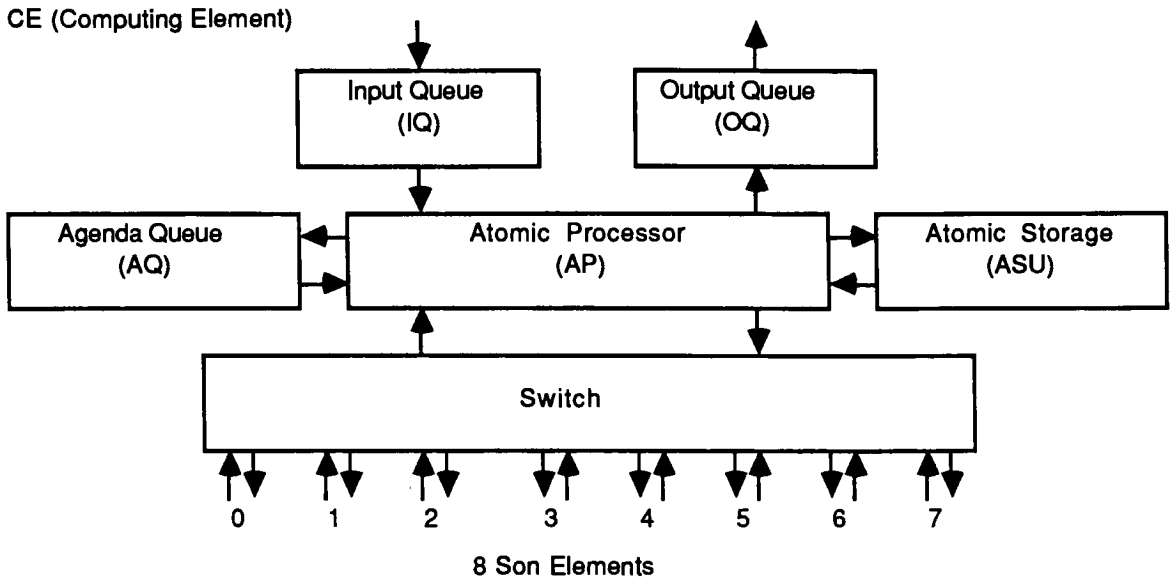
1. Project was inspired by Tesler and Enea paper on single assignment.
2. Design a single assignment high level language:
   a. that is easy to use by non-specialists.
   b. that naturally exploits parallelism in algorithms.
   c. that is readable and debuggable.
3. Develop a machine architecture to suit the single assignment language.

Current project status [Treleaven, Brownbridge ,Hopkins 1982]:

1. The first of 32 processors became operational in 1979.
2. The remaining processors have been constructed since.

Figure 7 describes the Toulouse LAU machine architecture.

Figure 6. Utah DDM Machine
[Davis 1979] [Davis 1978]



ASU - stores the program fragment.

AP - executes the instruction.

AQ - stores messages (program fragments, data tokens) for the local ASU.

IQ - stores messages from the superior CE.

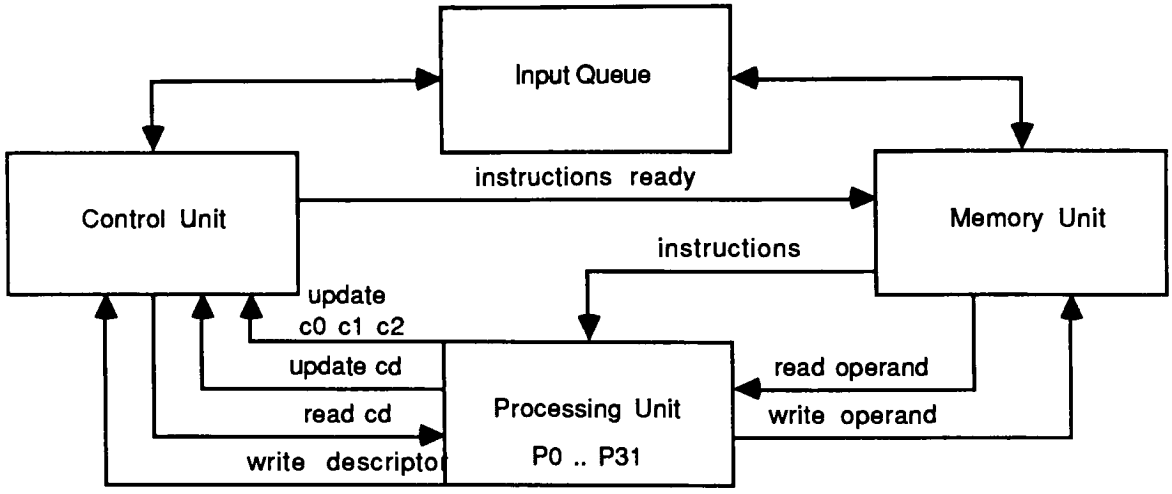
OQ - stores messages to the superior CE.

Switch - connects local CE with up to 8 inferior CEs.

Work in the form of a program fragment is allocated to a computing element by its superior via the IQ. If the fragment contains subprograms and the CE has sons, then it will decompose the fragment and allocate the subprograms to its inferior elements. Otherwise the fragment is stored in the local ASU.

When a data token arrives in the IQ it is either passed to the appropriate CE, or if the program fragment is local, the token is inserted into the instruction and executed immediately in the local AP. The result tokens are then distributed. If the result token is destined for a superior element it is placed in the OQ, for an inferior element it is placed in the switch, or if the receiving instruction is in the local ASU, the token is placed in the AQ.

Figure 7. Toulouse Machine
[Comte,Hifdi,Syre 1980] [Syre,Comte,Hifdi 1977] [Plas,et al 1976]



Memory Unit	- stores instructions and data.
Control Unit	- maintains the control memory (C0 C1 C2).
Processing Unit	- consists of 32 identical processing elements.
Input Queue	- pool of work for the memory unit.

The LAU programming language is based on single assignment rules, but the computer's program organization is based on control flow concepts. In the computer, data is passed via sharable memory cells that are accessed through addresses embedded in instructions. Separate control signals are used to enable instructions. However, as in data flow, the flow of control is tied to the flow of data.

Each instruction has three control bits that denote its state. C1 and C2 define whether the two input operands are present. C0 provides environment control (for example, instructions within loops). Cd is associated with each data operand and indicates if the operand is available.

Two processors scan the control memory: the update processor sets the C0 C1 C2 bits, and the instruction fetch processor associatively searches for 111 patterns. When an enabled instruction is found, its address is sent to the memory unit, and the control bits are reset to 011. The memory unit places the instruction on the instruction bus where it is accessed by an idle processing element. Once in a processing element, the instruction is decoded and the input adresses are sent to the memory unit to access the data operands. When the inputs return, the operation is performed, and the result is sent to the memory unit. The C0 C1 C2 bits of affected instructions are set, and the Cd bit is set.

1.2.6. EDDY Machine

In Japan, development of the EDDY (Experimental system for Data-Driven Processor arraY) is taking place.

Current project status [Hwang, Briggs 1984]:

1. Prototype has been built.
2. Compiler for the VALID programming language has been developed.
3. Statistical data has been collected:
   a. operation rates of function units.
   b. average queue lengths.
4. This data will be used to build custom hardware for the machine.

Figure 8 describes the EDDY machine architecture.

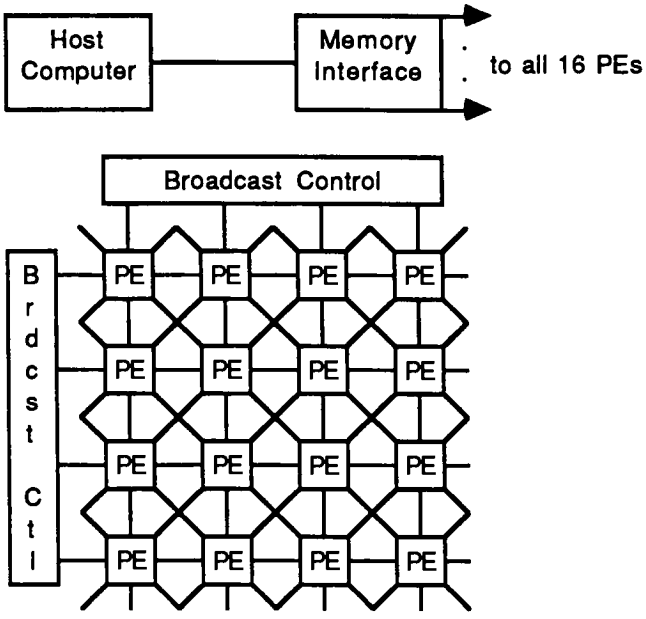
1.2.7. Other Projects

As of 1985 the only operational Data Flow machines are DDM at Utah, EDDY in Japan, the Manchester machine in the U.K., and the French LAU machine.

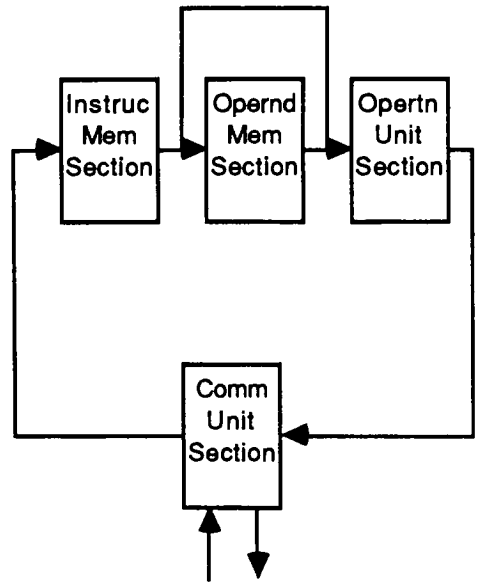
Some other projects of interest are the Texas Instruments Distributed Data Processor [Treleaven, Brownbridge, Hopkins 1982]. It was built using off-the-shelf technology and uses a cross compiler to translate FORTRAN 66 into directed graph representation. The Newcastle data-control flow computer integrates data flow and control flow computation [Treleaven, Brownbridge, Hopkins 1982]. Sigma-1 in Japan [Shimada, Hiraki, Nishida 1984] will be built with 256 processing elements and has as its goal to build a high performance machine with a speed of 100 MFLOPS.

Figure 8. EDDY Machine
[Hwang,Briggs 1984]

Architecture



PE (Processing Element)



Broadcast Control - loads or unloads programs and data to or from all PEs, in column or row, at the same time.

Instruction Memory Section - fetches an operand token's instruction and sends both the fetched instruction and the operand data to the Operand Memory Section.

Operand Memory Section - for two-operand operations the memory is searched associatively for its partner. If found, the packet is sent to the Operation Unit Section, otherwise it is stored.

Operation Unit Section - executes the operation packet and sends result tokens to the Communication Unit Section.

Communication Unit Section - sends tokens to the local PE or other PEs, also receives tokens from other PEs.

2. Project Description

This section describes the data flow computer being simulated. Program representation, data representation, structrure representation, machine organization, and the instruction set are detailed in this section.

This thesis project is an outgrowth of a previous thesis that was submitted to the Rochester Institute of Technology. The previous thesis, "Simulation of a Dataflow Computer", by Carol M. Torsone, suggests ways to improve its implementation.

Euclid was used to implement the original simulator, thus limiting the use and testing of the simulator to integers. Torsone concluded that many "real life" applications, which would have been interesting due to their high degree of concurrency, could not be programmed because of this limitation. To rectify this, Modula-2 is used to implement the new simulator, it supports both integers and reals, and provides coroutines for concurrent programming.

The original simulator only allowed for single-valued variables to be used as data tokens, which again limited the applications of the simulator. The new simulator supports, in addition to scalar variables, I-structures [Arvind, Thomas 1980], which are array-like data structures.

The new simulator follows more closely the U-interpreter algorithm [Arvind, Gostelow 1982] for tagging data tokens, thus allowing nested loops to be programmed.

Programs to be run on the original simulator had to be written using the simulator's machine language. The new simulator comes equipped with an assembler, and is programmed using an assembly language which is a statement representation of the dataflow graph.

Other differences between the two simulators include the machine instruction format, the handling of constants, and the handling of input and output.

Both projects simulate a dynamic dataflow machine based upon the machine organization under development at the University of Manchester [Gurd, Kirkham, Watson 1985]. The machine instruction set, of both simulators, is taken largely from [Dennis 1975].

2.1. Functional Specification

The simulator is based upon the Manchester machine architecture [Gurd, Kirkham, Watson 1985], uses the token tagging scheme of the U-interpreter [Arvind, Gostelow 1982], has an assembly language based on [Dennis 1975], and supports the I-structure data structure [Arvind, Thomas 1980].

2.1.1. Program Representation

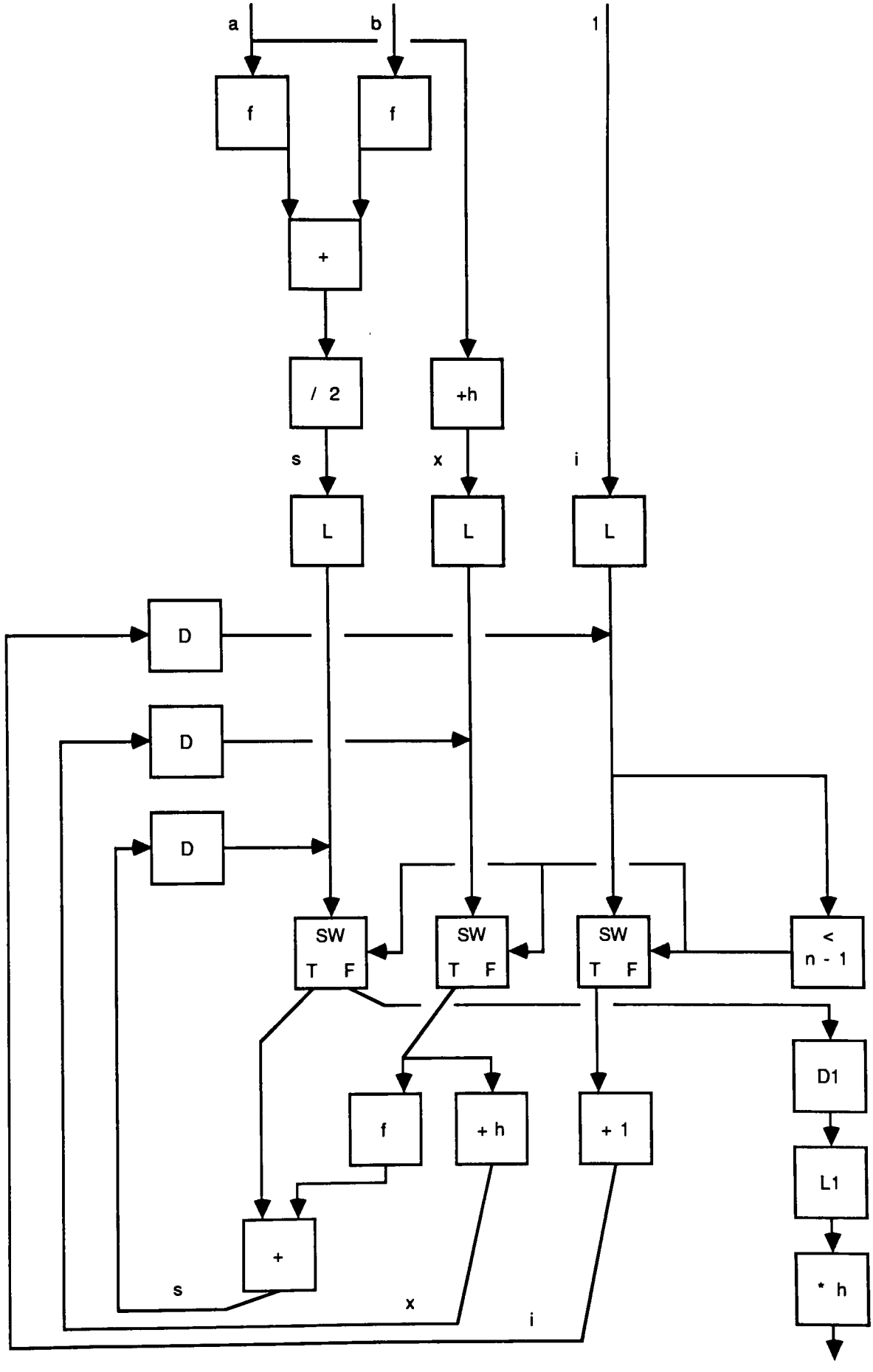
Dataflow compilers translate high-level programs into directed graphs. Vertices in the graph correspond to machine instructions, and edges correspond to the data dependencies which exist between the instructions.

The implication is that instructions which depend on other instructions should be sequenced accordingly, but where no dependencies exist, the instructions can be executed in parallel. A graphical translation is shown in figure 9, it was compiled from the following high-level program, which integrates a function f from a to b over n intervals of size h by the trapezoid rule:

```
s = (f(a) + f(b)) / 2
x = a + h
for i = 1 to n - 1
  s = s + f(x)
  x = x + h
end for
s = s * h
```

In figure 9 the box marked f represents the subgraph of function f. Instructions D, D1, L, and L1 are included to provide proper entry, iteration, and exit by manipulating context-identifying information (discussed in the next section). The remainder of the operators are arithmetic, relational, and conditional instructions.

Figure 9. Trapezoid Rule Compilation



2.1.2. Data Representation

It is the processors's task to propagate data values through the program graph, triggering instructions when the operands are available. Data values are carried by logical entities called tokens. A token contains not only a data value but also the address of its destination instruction. Conceptually, tokens move about on the vertices of the graph. Instructions are enabled when tokens are present on all input edges. Program execution consists of an instruction absorbing its input tokens, and producing an output token for the next instruction in the graph. A program terminates when there are no enabled instructions left.

In a dynamic model, more than one token is allowed to be present on an arc; therefore, the next-instruction label also contains dynamic, or context-sensitive information (called the tag). These next-instruction labels or activity names [Arvind, Gostelow 1982] contain three parts:

u: The context field, which uniquely identifies the context in which the instruction is invoked. The context field is itself an activity name.

i: The initiation number, which identifies the loop iteration in which this activity occurs. The field is 1 outside a loop.

s: The instruction address.

Since instructions may have more than one input operand, an index value, called the port (p), which specifies the operand number associated with this token, is also carried on each token. The complete token looks like this:

<u i s data>p

2.1.3. Structure Representation

Data structure operations present a problem for dataflow machines. In functional languages (the language of dataflow), a data structure is acted upon as a single entity; the entire data structure moves through the program graph, and each structure operation results in the creation of a new structure consisting of the changed element and all unchanged elements. Parallelism is reduced

because it is not possible to operate on more than one structure element at one time. [Arvind, Thomas 1980] propose the I-structure as a new array-like data structure, which can significantly reduce structure overhead, and provide for highly parallel structure creation and operation. I-structure operations are based on the premise that, in many circumstances, full generality of data structure operations is not needed; hence significant gains should be possible by substituting restricted data structure operations.

An I-structure is an asynchronous structure with a constraint on its construction. I-structure producers can only append a value once to a particular selector of an I-structure, no other value can ever be appended to that selector in that particular I-structure. The definition of I-structure producers permits individual appends to be done out of order, thus allowing concurrent construction of an I-structure. Because I-structures are asynchronous, values can be selected from an I-structure before the I-structure is complete. The read-before-write problem is handled by deferring all read requests of an empty cell until after the first write operation.

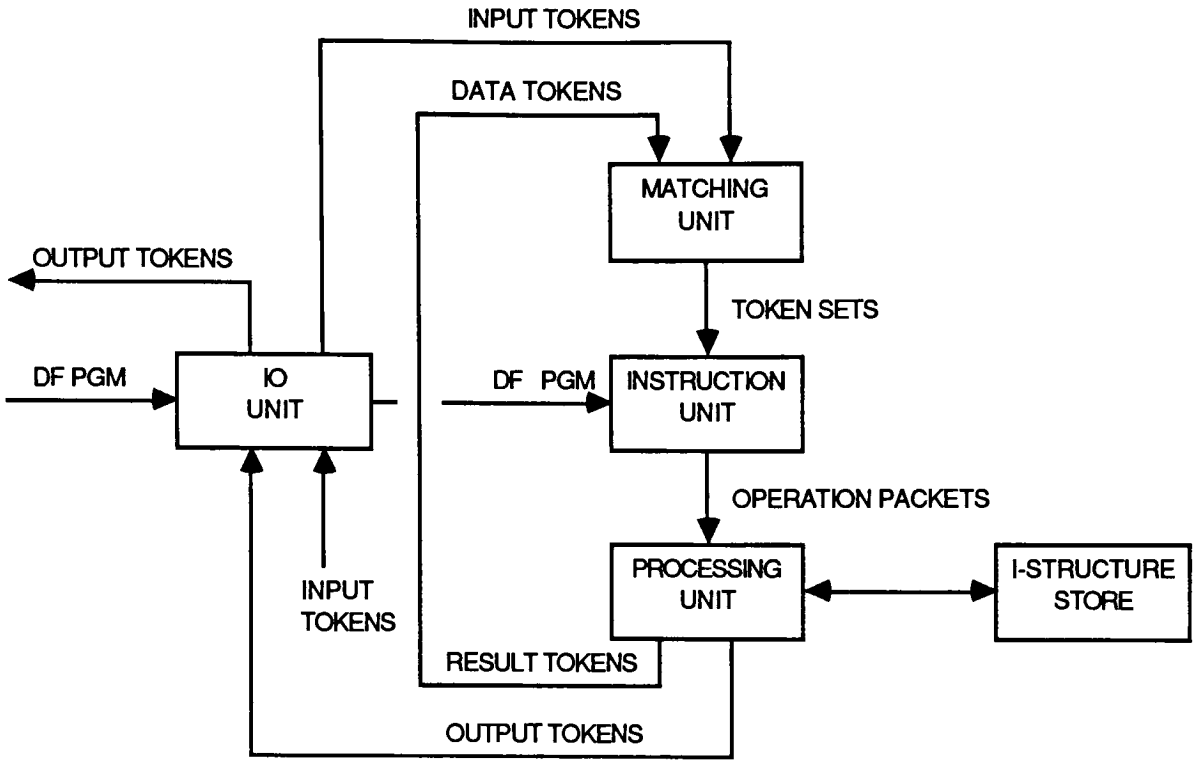
2.1.4. Machine Organization

The organization is based on the Manchester machine architecture. (See figure 10).

2.1.5. Instruction Set

The instruction set is taken largely from [Dennis 1975] in which he defines a data flow program as a bipartite directed graph where the two types of nodes are called links and actors. He regards the arcs of a data flow program as channels through which tokens flow carrying values from each actor to other actors by way of the links. The instruction set proposed here differs from [Dennis 1975], in that data is permitted to travel directly from one actor to another actor. Links are used only to replicate tokens with multiple destinations. The thesis instruction set also includes actors necessary for implementing the U-Interpreter [Arvind, Gostelow 1982], and for implementing I-structures [Arvind, Thomas 1980].

Figure 10. Machine Organization



IO Unit

Assembles program and loads machine instructions into the instruction store.
Sends input tokens to matching unit.
Sends output tokens to output device.

Matching Unit

Forms token set based on activity name.
Sends token set to instruction unit

Instruction Unit

Forms operation packet from machine instruction opcode and token set.
Sends operation packet to processing unit.

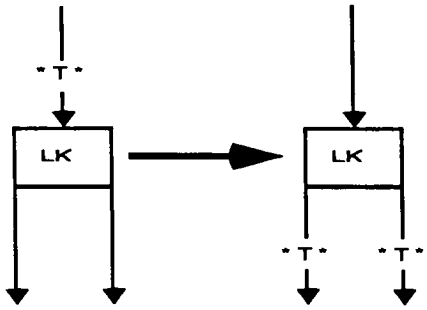
Processing Unit

Sends incoming operation packet to an available processor.
Executes operation packet.
Sends result token to the matching unit.
Sends output token to the IO unit.

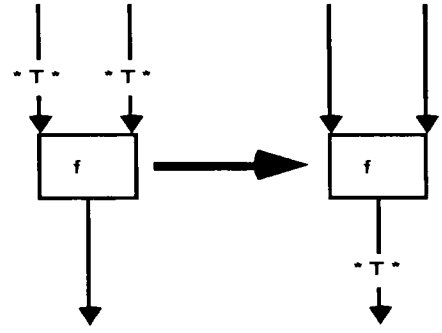
I-Structure Store

Stores I-structure tokens.

LINK: replicates its input token and distributes the copies to its output destinations.

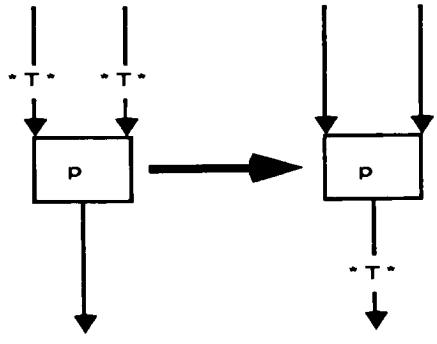


OPERATOR ACTOR: applies its function to its two input tokens (one input token for unary functions) and sends the result to its output destination.



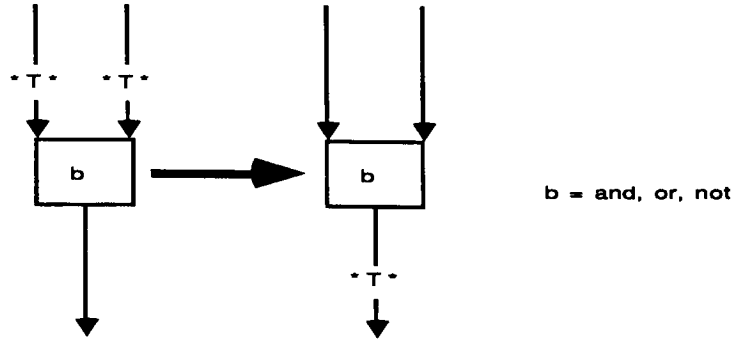
f = negation, sqrt, abs, +, -, /, *

DECIDER ACTOR: applies its predicate to its input tokens and sends the resulting control token (true or false) to its output destination.

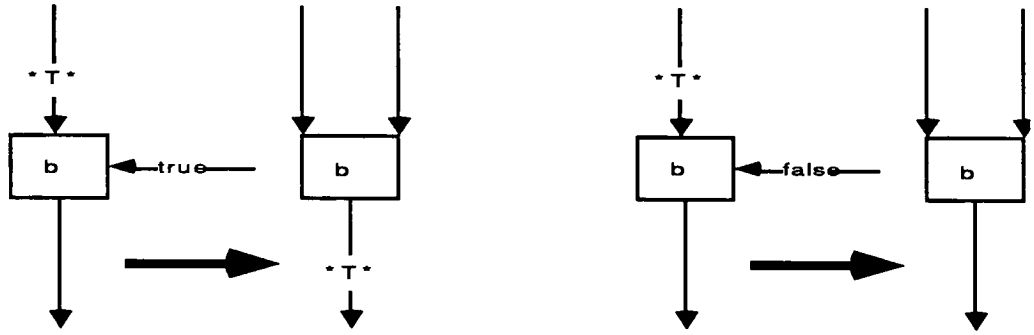


p = <, >, <>, =, <=, >=

BOOLEAN ACTOR: applies its boolean function to its input tokens and sends the resulting control token (true or false) to its output destination.

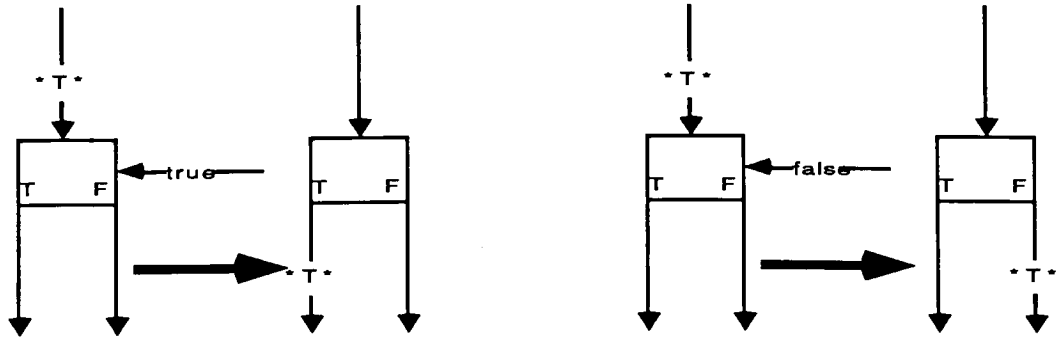


T-GATE CONTROL ACTOR: passes its input token to its output destination if it receives the value true at its control operand; the data operand is discarded if false is received.



F-GATE CONTROL ACTOR: passes its input token to its output destination if false is received, discards its input if true is received.

SWITCH CONTROL ACTOR: allows a control value to determine which of two output destinations its input should be passed to. A true control value will cause the input data to be routed to the T-destination; a false value will cause the data to be routed to the F-destination.

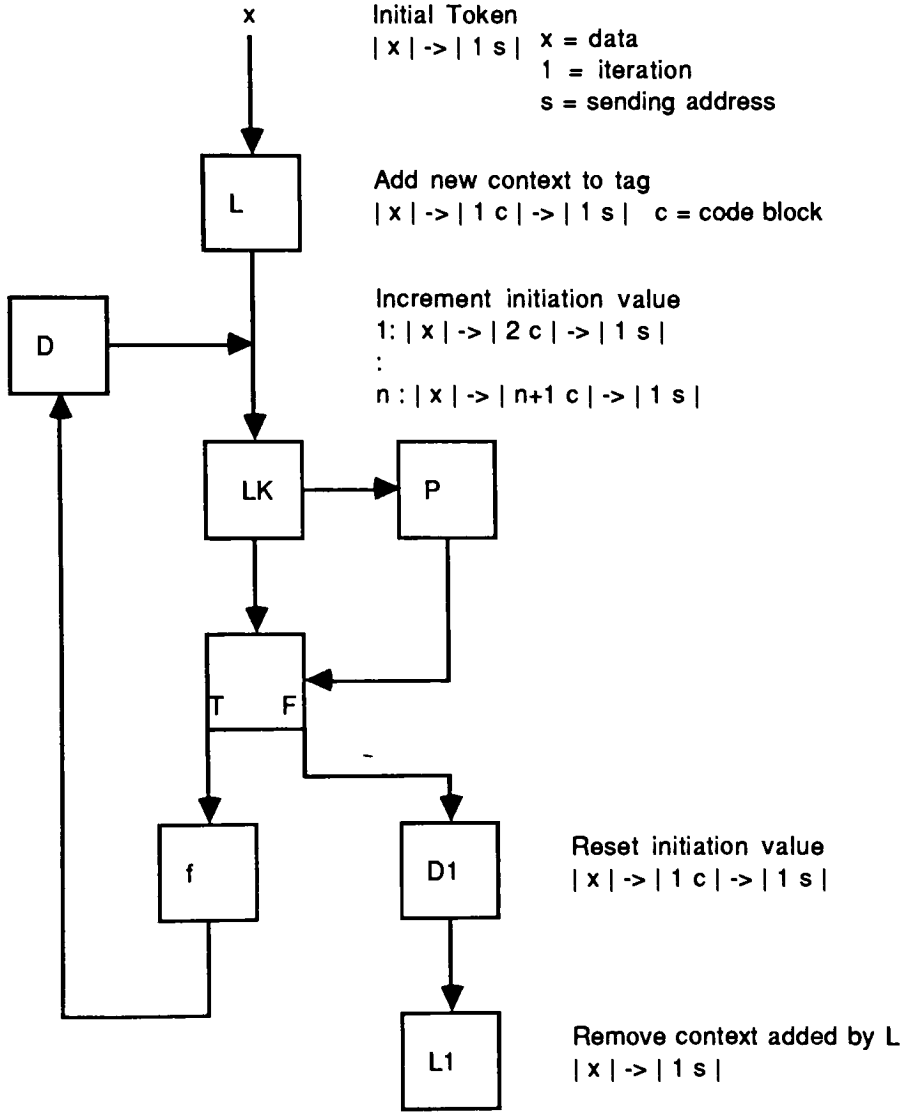


LOOP ACTORS (L, L1, D, D1): manipulate context-sensitive information in the token tag, making it possible to concurrently execute several iterations of a loop.

The L actor adds new contexts to the token tag when loops are entered; L1 removes contexts added by L when loops are exited.

The D actor adds 1 to the initiation value; D1 resets the initiation value to 1.

```
while (p(x))
   x = f(x)
end while
```



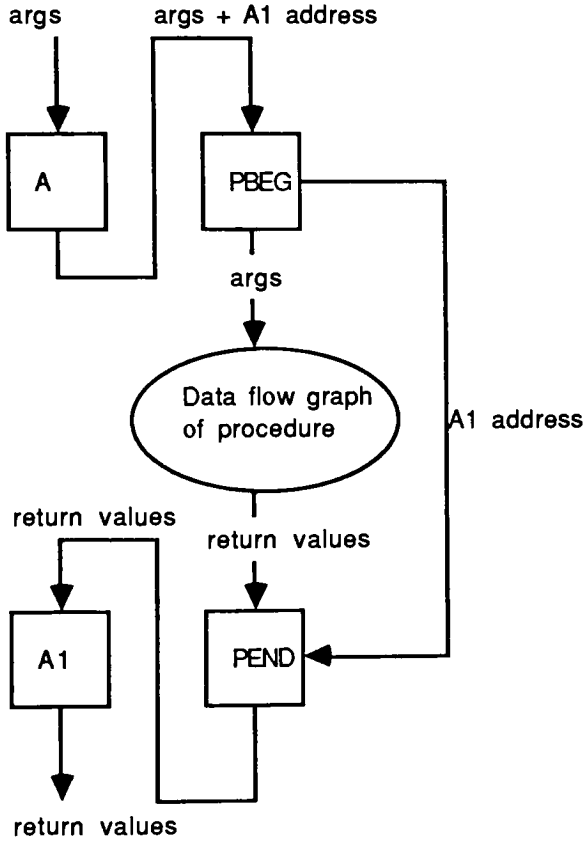
APPLY ACTORS (A, PBEG, PEND, A1): operate on context-sensitive information in the token tag, making it possible to concurrently execute several instantiations of a procedure.

The A instruction adds a new context to the token tag each time a procedure is invoked, sends its input tokens (procedure arguments) to the PBEG instruction, and sends the A1 instruction address (return address) to the PBEG instruction.

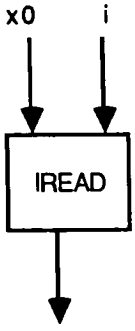
The PBEG instruction is always the first instruction of a procedure, it collects the procedure arguments and distributes them to the statements of the procedure, and sends the A1 address (return address) to the PEND instruction.

The PEND instruction is always the last instrruction of a procedure, it collects the procedure result tokens and sends them to the A1 instruction.

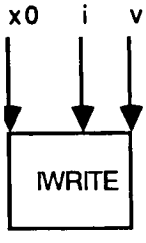
The A1 instruction removes the context added by A and distributes the tokens to the statements of the calling procedure.



I-STRUCTURE ACTORS (IREAD, IWRITE): manipulate I-structures.



The IREAD operation retrieves the data value of I-structure x0 at selector i.



The IWRITE operation appends the value v to I-structure x0 at selector i. It also satisfies any pending reads.

3. Project Implementation

This section discusses the simulator implementation. It describes the inputs, outputs, data structures, and algorithms of the processes that constitute the system. It ends with a discussion of the monitors and process synchronization used to simulate parallel execution.

The simulator was written in Modula-2. Modula-2 was chosen because it supports real numbers and provides coroutines as a vehicle for simulating concurrent execution.

The simulator will support a maximum of five hundred instructions per program, twenty arguments per subroutine call, and one hundred i-structures of fifty cells apiece. Any of these limits may be changed by modifying the DataFlowDecls.def file and recompiling the simulator. The current limits were chosen to achieve reasonable run-time performance.

The simulator operates only on numerical data (with the exception of output labels). Data may be entered as either integer or real; the simulator converts all values to real numbers. Control values are represented by a 1 for true and a 0 for false.

The simulator consists of five functionally different processes (see figure 11). Build token set simulates the match unit, build operation packet simulates the instruction unit, execute operation packet simulates the processor, and the remaining processes simulate the IO unit.

The simulator begins by executing the assemble instruction process to convert the dataflow assembler program into the simulator's machine code. The read data process reads the input file and produces input tokens. The config file (see figure 12) is read to determine the number of processors that are to be activated for this invocation; a separate execute operation packet coroutine is started for each active processor. Build token set and build operation packet are started as coroutines to simulate the match unit and instruction unit, respectively.

Execution begins when the read data process produces input tokens. Execution of the simulator precedes as the token, token set, and operation packet queues are produced and consumed. Execution halts when all queues have been fully consumed.

Figure 11. System Process Diagram

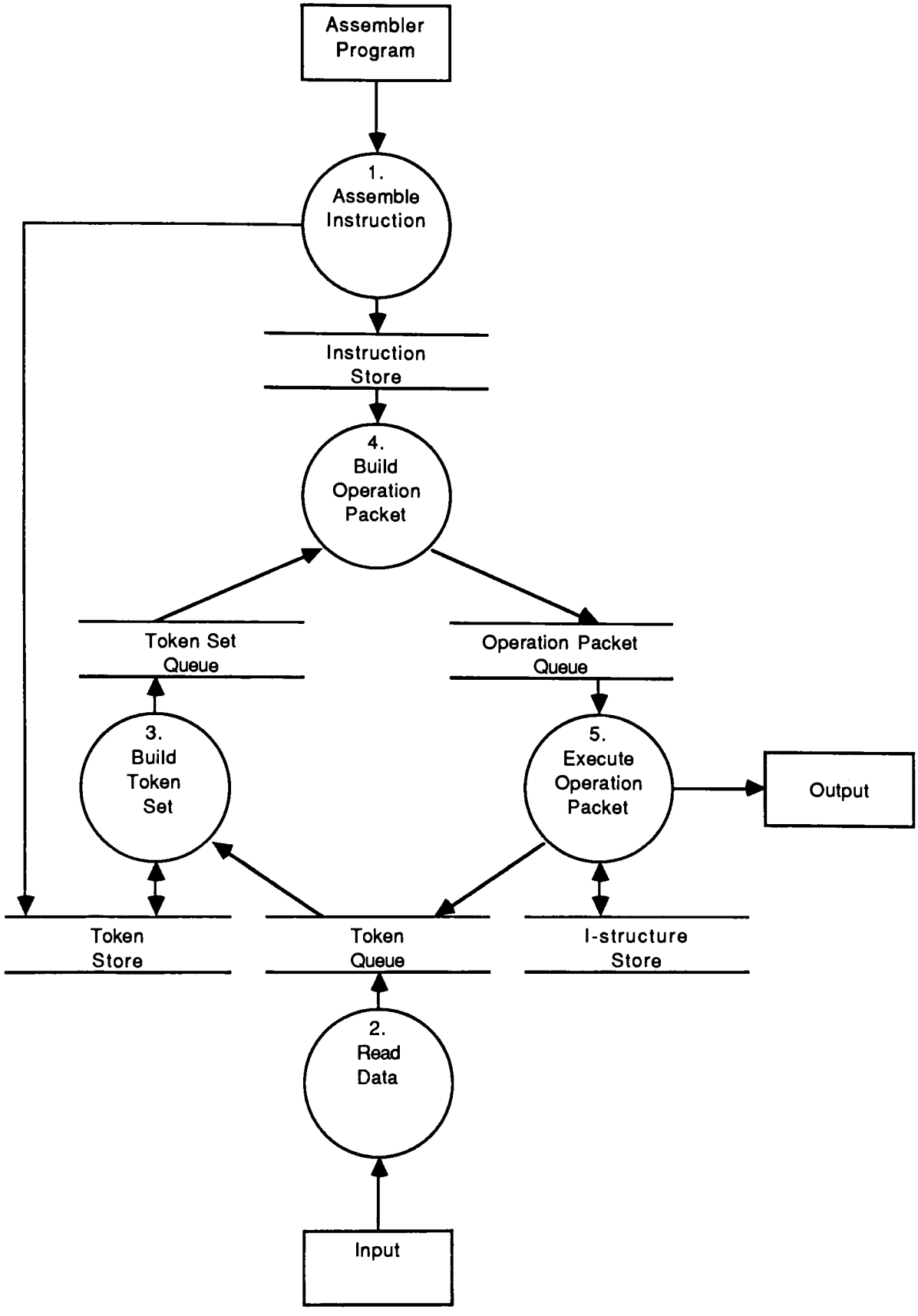


Figure 12. The config file

```
------------------------------
20  2000000  2000000  2000000
------------------------------
```

20	is the number of processors to be activated.
2000000	is the working set size of the match unit coroutine.
2000000	is the working set size of the instruction unit coroutine.
2000000	is the working set size of the processor coroutines.

The config file may be edited prior to running the simulator to establish the number of processors that are to be activated during the simulation.

3.1. Assemble Instruction Process

The assemble instruction process reads the file containing the dataflow assembler commands, assembles the commands into the simulator machine code, and stores the machine code in the instruction store data structure. The assemble instruction process also stores each instruction's enable and constant counts in the token store.

The simulator instruction format is as follows:

OP EC DC CC 20{PORT}20 20{DEST}20

OP EC DC CC LABEL 20{PORT} 20{DEST}20
(OUTPUT and DEBUG instructions)

OP Instruction opcode.

EC Instruction enable count.
This is the number of operands needed to execute the instruction.

DC Instruction destination count.

CC Instruction constant count.

LABEL Character string that labels output data.

PORT Slots in the instruction where incoming
operands are stored. These slots include
a constant presence bit. The CONST assembler
directive stores the constant value in the
port, sets the constant presence bit, and
increments the constant count by one.
There are twenty slots per instruction.

DEST Destination address and port.
These are the instruction addresses and ports
where the result tokens (produced by executing
the instruction) are to be delivered.
There are twenty of these per instruction.

The assemble instruction process begins by loading the command table. The commandTable file contains the syntactical information needed to assemble the simulator's dataflow programs. The table has for each assembler instruction, the instruction's mneumonic, opcode, enable count, and destination count (see figure 13).

The instruction store is an array of five hundred instruction store records (see figure 14). An instruction's address is its index value into the array. The zero instruction address has a special meaning for the assembler; it is reserved for specifying a null destination address.

3.2. Read Data Process

The read data process reads the input file, builds an input token from each record in the file, and sends the input tokens to the token queue. This is the process which puts the simulator in motion.

The input file record consists of a data value and a destination (recall that a destination consists of an instruction address and a port number). The input file is further discussed in the Project Application section.

3.2.1. Token Implementation

Tokens consist of a data value (data values are not distinguished from control values), an activity name count, and an activity name (tag), which contains the token's destination and context

Figure 13. The commandTable file

```
-------------------------------------------
#                100        0        0
A                  1        0        1
A1                 2        0        0
ABS               31        1        1
ADD                3        2        1
AND                4        2        1
CONST            101        0        0
D                  5        1        1
D1                 6        1        1
DEBUG             34        0        0
DIV                7        2        1
EQ                 8        2        1
FGATE              9        2        1
GE                10        2        1
GT                11        2        1
HALT              32        1        0
IFREE             33        3        3
IREAD             12        2        1
IWRITE            13        3        1
L                 14        2        1
L1                15        1        1
LE                16        2        1
LINK              17        1        2
LT                18        2        1
MUL               20        2        1
NE                21        2        1
NEG               22        1        1
NOT               23        1        1
OR                24        2        1
OUTPUT            19        0        0
PBEG              25        0        0
PEND              26        0        1
SQRT              27        1        1
SUB               28        2        1
SWITCH            29        2        2
TGATE             30        2        1
TRACE            102        0        0
-------------------------------------------
```

Column 1 contains the instruction mneumonic.
Column 2 contains the instruction opcode.
Column 3 contains the instruction enable count.
Column 4 contains the instruction destination count.

Figure 14. Instruction Store Data Structure

1	OP EC DC CC L P_1 .. P_{20} D_1 .. D_{20}
:	
500	OP EC DC CC L P_1 .. P_{20} D_1 .. D_{20}

OP instruction opcode.

EC instruction enable count (number of operands needed to execute instruction).

DC instruction destination count.

CC instruction constant count.

L character string that labels output data.

P slots for receiving instruction operands.

D destination address and port where the instructions's result tokens are to be sent.

information (see figure 15).

3.3. Build Token Set Process

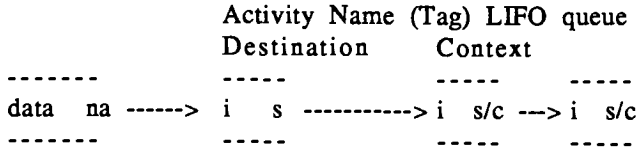
The build token set process forms tokens into sets which are destined for the same instruction and have the same context. A token set is complete when all the tokens needed to enable an instruction are gathered together. Completed token sets are sent to the token set queue.

Tokens which by themselves enable an instruction, and tokens which when combined with an instructions's constants enable an instruction, are made into a token set of one and queued.

The token matching algorithm searches the token store (see figure 16) for any token sets whose destination is that of the token. If a matching set is found then the token activity name is compared. If the names match, the token's data is inserted into

---

Figure 15. Token Data Structure



data — data value or control value.

na — number of activity names in the tag.

i — initiation number (loop iteration counter).

s — instruction address and port.

c — loop code block number.

s/c indicates that in some instances the address is part of the activity name, while other times it is the code block number. When loops are entered it is the loop code block number that is needed to match tokens. When subprograms are invoked it is the invoking instruction's address (the A instruction) that is needed to match tokens.

na is used to improve the simulator's run-time execution speed. It speeds up the token matching algorithm by allowing it to immediately discard tokens whose activity name counts are unequal; this eliminates needless traversing of activity name queues when matching token activity names.

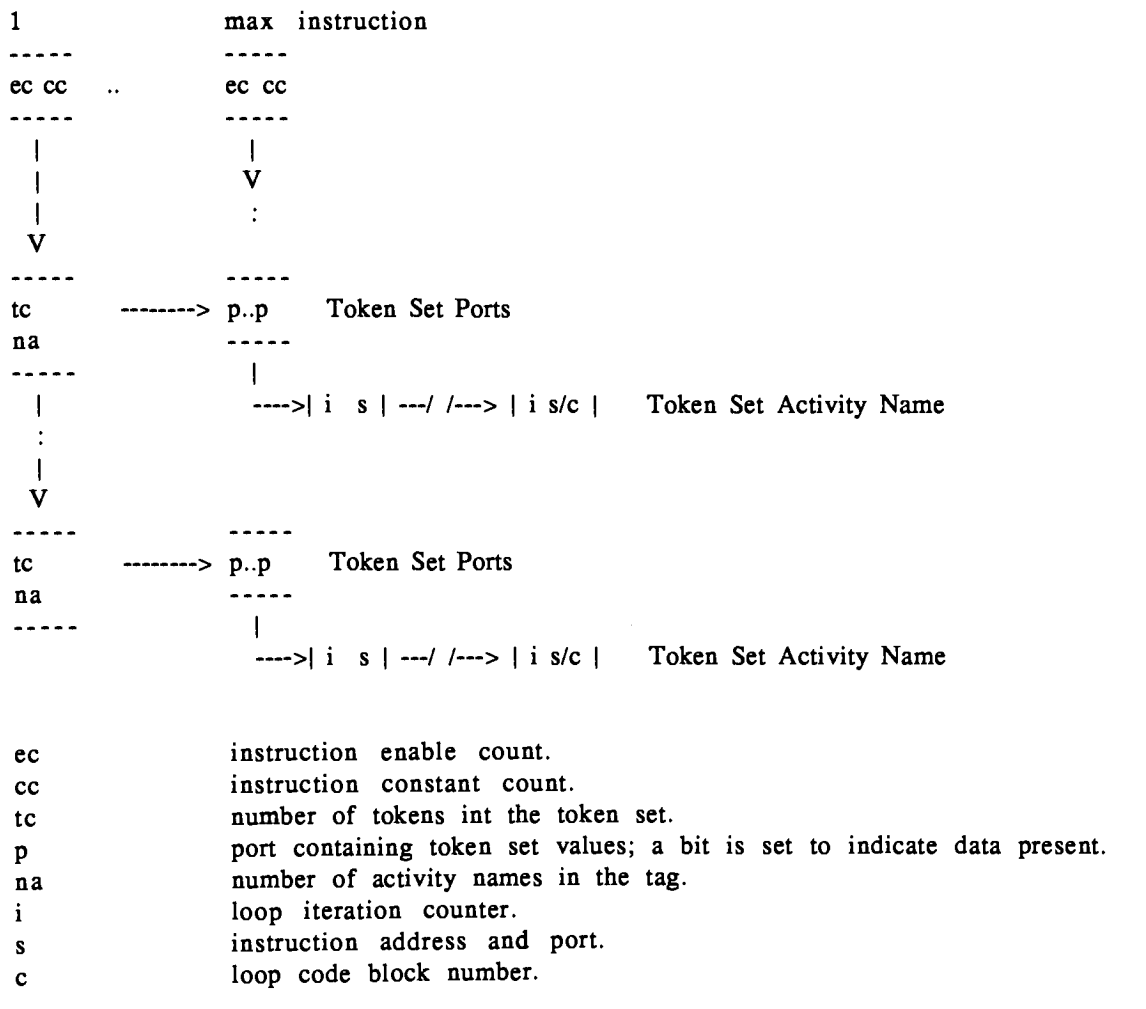
---

the appropriate token set port, and the token set count is incremented. If the instruction enable count equals the token set count then the set is queued and deleted from the token store.

If a token is the first in its set to arrive, a token set record is created and inserted into the token store at the end of the queue for that token's destination address. The token data is inserted into the appropriate port, the token set activity name becomes the token's activity name, and the token set count is set to one.

In the event where a token is destined for an instruction that is already occupied, the simulator will issue an error message and abort execution.

Figure 16. Token Store Data Structure



ec instruction enable count.
cc instruction constant count.
tc number of tokens int the token set.
p port containing token set values; a bit is set to indicate data present.
na number of activity names in the tag.
i loop iteration counter.
s instruction address and port.
c loop code block number.

3.4. Build Operation Packet Process

The build operation packet process combines token sets with their destination instruction's opcode, enable count, destination count, output label (if there is one), and constant values (if there are any). The completed packet is sent to the operation packet queue where it waits to be executed by the next available processor.

3.5. Execute Operation Packet Process

The execute operation packet invokes the appropriate procedure to perform the operation specified by the operation packet's opcode. The data resulting from the operation, and the operation packet's destination and context are formed into a result token and sent to the token queue.

The simulator instruction set is completely defined in the Project Description section (see section 2.1.5). Each instruction is implemented as a separate procedure. The instruction set can be easily extended by

---

Figure 17. Operation Packet Data Structure

| op ec dc l p..p d..d na | ----> | i s | ---/ /---> | i s/c |

op	instruction opcode.
ec	instruction enable count.
dc	instruction destination count.
l	output label.
p	port containing operand value; a bit is set to indicate data present.
d	destination address and port of result token.
na	number of activity names in the tag.
i	loop iteration counter.
s	instruction address and port.
c	loop code block number.

---

writing the instruction procedure and linking it with the rest of the simulator modules. Placing the instruction's command syntax into the commandTable file (see figure 13) will incorporate the instruction into the simulator's assembler language.

3.5.1. I-Structure Implementation

The i-structure store (figure 19) is an array of one-hundred by fifty i-structure cells. The i-structure is implemented as an asynchronous (allows reads to occur before writes) data structure that provides most of the general functionality of data structures. It is constrained by allowing only

---

Figure 18. Instruction Set Summary

Arithmetic Functions:
SQRT, NEG, ABS, ADD, SUB, MUL, DIV

Relational Functions:
EQ, NE, GE, GT, LE, LT

Logical Functions:
AND, OR, NOT

I-Structure Functions:
IREAD, IWRITE, IFREE (reclaims an i-structure cell)

Loop Functions:
L, L1, D, D1

Subprogram Functions:
A, A1, PBEG, PEND

Branch Functions:
LINK, TGATE, FGATE, SWITCH, HALT

Output Functions:
OUTPUT, DEBUG

Trace Function:
TRACE

---

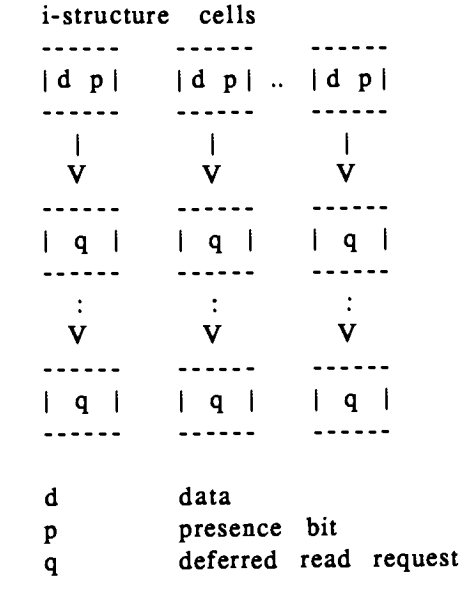
one write to a particular cell. The simulator relaxes this constraint by providing the IFREE instruction to reinitialize an i-structure cell, allowing it to be written to again. In addition the i-structure write (the simulator's IWRITE instruction) has been enhanced to send a true signal when the write is complete; this provides a mechanism to create a data dependency between a cell write and a cell reinitialization.

The IREAD instruction sends a copy of an i-structure cell's data value to a destination instruction. If the cell's presence bit is set a token is created with a copy of the cell's data value and sent to the token queue. If no data is present then the read request is placed into that cell's deferred read FIFO (first-in/first-out) queue, where it will await a write operation to that cell.

The IWRITE instruction stores a data value at a particular i-structure cell. If the cell's presence bit is set then an error is issued and the simulator aborts. If the cell is empty then the data value from the IWRITE instruction is stored in the cell and the presence bit is set. Any deferred

---

Figure 19. I-structure Store Data Structure



---

read requests are satisfied by sending a copy of the cell's contents as a token to the read request's destination instruction. When the write is complete a true control token is issued to the token queue.

The IFREE instruction reinitializes an i-structure cell. The reinitialization consists of resetting the cell's presence bit to zero, and setting the cell's deferred read queue pointer to null. The IFREE instruction is designed so that it may directly precede an IWRITE instruction by allowing a programmer to create an IWRITE data dependency on the IFREE instruction (an example of this can be found in the Laplace transform sample program). In this manner a programmer can ensure that a cell is reinitialized before it is reused.

3.6. Monitors and Process Synchronization

The multi-programming of the concurrently executing processes, the match unit, instruction unit, and twenty processors, conforms to the standard suggested by Niklaus Wirth [Wirth 1985]. The Processes module linked to the simulator follows closely the standard algorithms for the Init, Wait, Send, and StartProcess functions.

Init	- initializes a signal
Wait	- suspends execution until a signal is given
Send	- sends a signal
StartProcess	- defines and transfers control to a process

The simulation of concurrent execution is accomplished by launching parallel processes as coroutines. A coroutine is a procedure that executes independently of other coroutines and procedures in the program, and via synchronization signals allows for the direct transfer of control from one coroutine to another. Coroutines, though viewed as executing in parallel, are in fact quasi-concurrent processes in that they share a single physical processor. The terms coroutine and process will be used interchangeably throughout the remainder of this section.

Shared variables, such as the token, token set, and operation packet queues, present a problem for coroutines. If two processes are concurrently accessing and possibly changing a shared variable,

the integrity of the variable's value could be lost. Modula-2 provides a monitor construct which is a separate module that contains the shared variables and the procedures for operating upon them. Because a monitor can only execute one procedure at a time, only one process can access the shared variables at any instant. In this way a monitor protects against simultaneous updating of shared variables. All queues in the simulator are implemented as producer/consumer monitors. These monitors consist of FIFO (first-in first-out) queues, and the operators, send and receive, which act upon the queue.

Process synchronization in the simulator occurs via signals. The StartProcess routine is used to start coroutines, it inserts the coroutine into the coroutine ring (a circular queue), sets the ready state flag to true, indicating the process is ready to execute, and transfers control to the process. The Wait routine is used to suspend a process pending a particular condition, it places the process in the particular signal's wait queue, sets the ready state flag to false, indicating the process is in the blocked state, and transfers control to the first coroutine in the ring with its ready state flag set to true. If no coroutines are in the ready state then execution is halted. The SEND is used to signal a process that a particular condition has occurred. It removes a coroutine from the signal's wait queue, sets the coroutine's ready state flag to true, and transfers control to the process. If the signal's queue is empty, then the SEND function becomes in effect a null operation.

The simulator's process synchronization follows the algorithms outlined below.

Send to queue:

```
Insert object into queue.
Send arrival signal.
```

Receive from queue:

```
If queue is empty then
  Wait for arrival signal
else
  Remove object from queue
  Deliver it to requesting process.
```

Coroutine:

```
Loop
  Receive object from queue.
  Process object.
  Send resulting object to queue.
Forever
```

4. Project Application

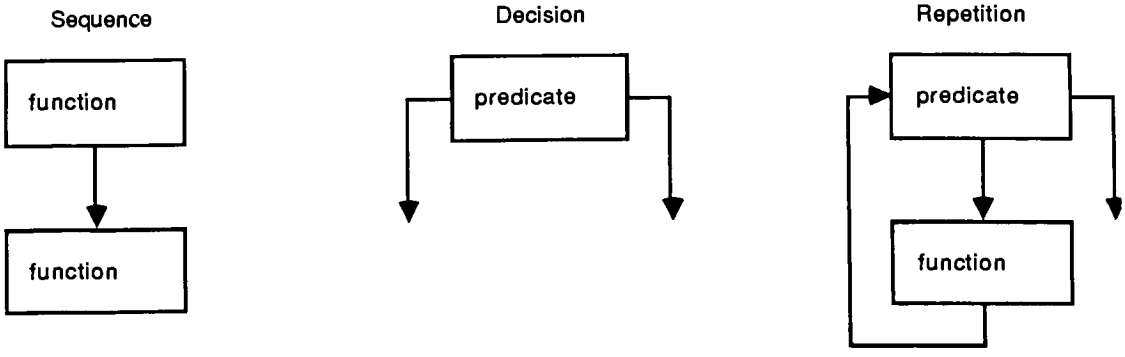
This section discusses the manner in which the simulator may be applied. Topics covered include the data flow graph, the assembler language, the input file, debugging, sample programs, running the simulator, and simulator errors.

The simulator may be applied as a tool for learning data flow programming. A wide range of programs can be written in the simulator's assembler language and tested using the simulator.

Because the simulator only simulates concurrency, its application is limited; consequently some interesting statistics, such as average queue lengths, speed-up versus number of processors, and speed-up versus degree of program parallelism, would be meaningless.

4.1. Data Flow Graphs

Data flow programming begins with developing an algorithm in the usual fashion, then deriving the algorithm's data flow graph. The data flow graph is a complete language, providing the basic language constructs of sequence, decision, and repetition.

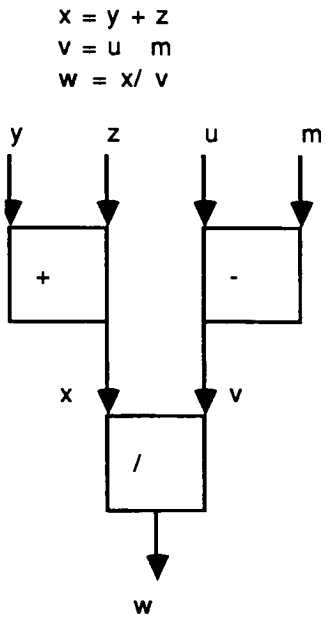


In addition, recursive procedures, and the i-structure data structure are also supported.

4.1.1. Sequence Construct

The data flow sequence construct (figure 20) is the result of data dependencies among two or more functions.

Figure 20. Data Flow Graph Sequence Construct



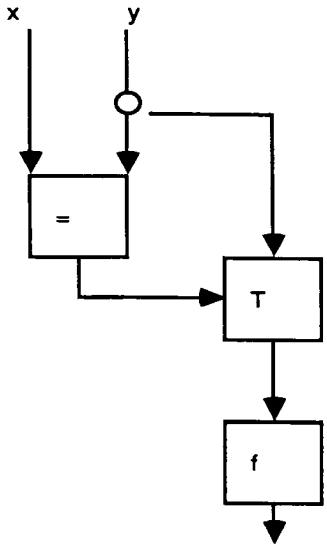
The division is data dependent on both the addition and subtraction.

4.1.2. Decision Construct

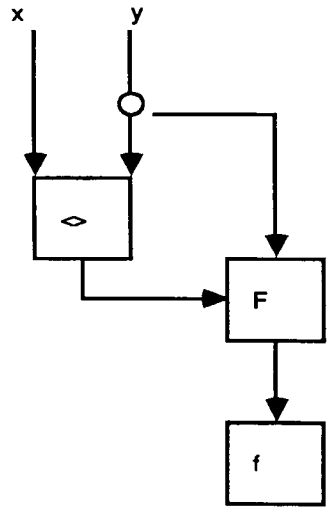
The TGATE, FGATE, and SWITCH instructions are used for decision branching in a data flow graph (see figure 21).

Figure 21. Data Flow Graph Decision Construct

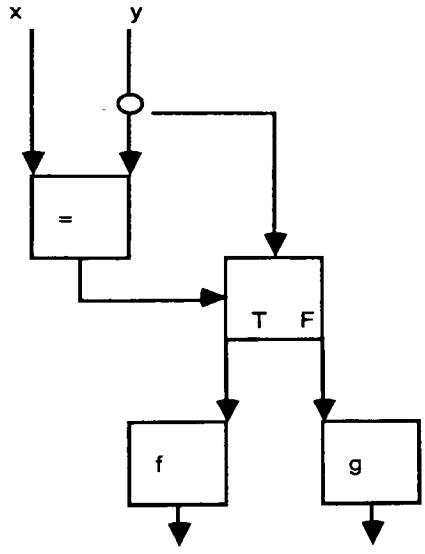
if x = y then f(y)



or



If x = y then f(y) else g(y)

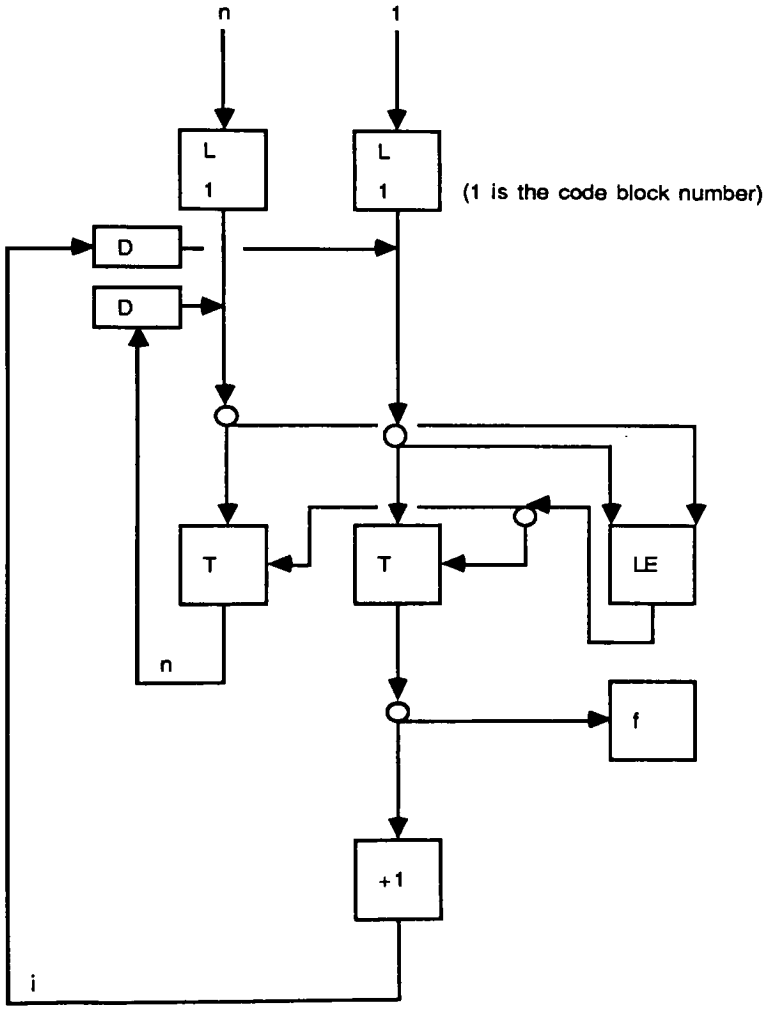


4.1.3. Repetition Construct

The loop instructions (figure 22) place loop context information in the token tag, which is used to match tokens according to loop code block number and loop iteration count.

Figure 22. Data Flow Graph Repetition Construct

```
for i = 1 to n
   f(i)
end for
```

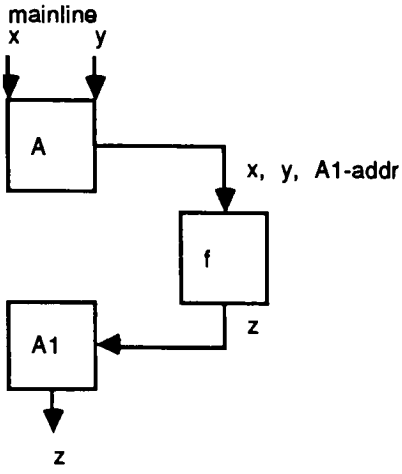


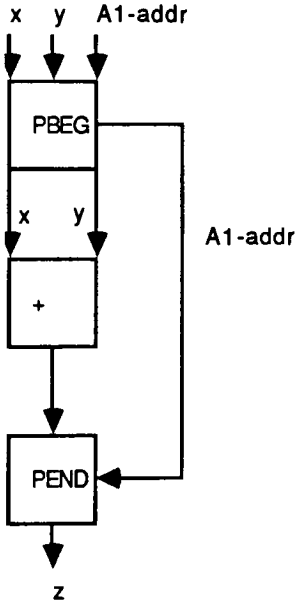
4.1.4. Subprograms

Recursive subprograms are supported by the A, A1, PBEG, and PEND instructions (figure 23).

Figure 23. Data Flow Graph Subprogram

```
mainline        subprogram
-------         ----------
call  f(x,y,z)  z = x + y
                return
```



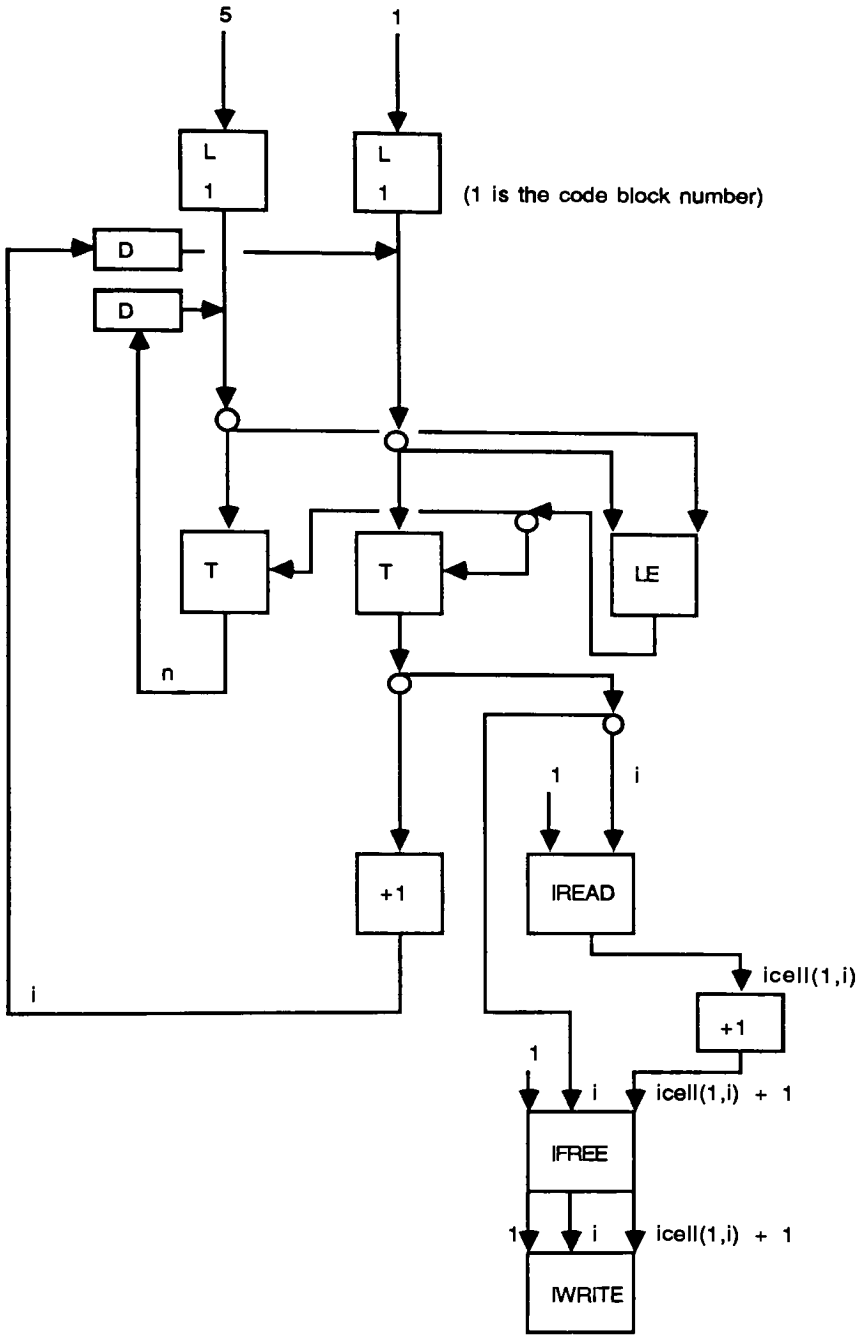


4.1.5. I-structures

The IREAD, IWRITE, and IFREE instructions read, write, and reinitialize i-structure cells (figure 24).

Figure 24. Data Flow Graph I-structure

```
for i = 1 to 5
   icell(1,i) = icell(1,i) + 1
end for
```



4.2. Assembler Language

The dataflow assembler language is a statement description of the dataflow graph. The assembler commands are summarized in figure 25; figure 26 summarizes the assembler syntax.

Figure 25. Assembler Command Summary

#	comment; characters from # to end of line are skipped
a	add subprogram context; invoke subprogram
a1	remove subprogram context; distribute result tokens
abs	absolute value
add	addition
and	boolean and
const	store instruction constants
d	increment loop iteration counter
debug	output token values at particular point in graph
d1	reset loop iteration counter to 1
div	division
eq	equal
fgate	propagate token if control signal is false
ge	greater than or equal to
gt	greater than
halt	halt program
ifree	reinitialize i-structure cell
iread	read i-structure cell
iwrite	write i-structure cell
l	add loop context
l1	remove loop context
le	less than or equal to
link	duplicate token
lt	less than
mul	multiplication
ne	not equal
neg	negate value
not	boolean not
or	boolean or
output	output token value(s)
pbeg	initiate procedure
pend	terminate procedure
sqrt	square root
sub	subtraction
switch	choose token path depending on control signal
tgate	propagate token if control signal is true
trace	trace simulator execution

Figure 26. Assembler Syntax Summary

---

addr	instruction address address range is from 1 to 500; 0 specifies a null address
dest	destination instruction address and port
numarg	number of procedure arguments (maximum 20)
numtoken	number of tokens (maximum 20)
[]	optional field
m{}n	from m to n of field

#	comment text		
a	addr	numarg	a1-addr pbeg-addr
a1	addr	numarg	numarg{dest}numarg
abs	addr		dest
add	addr		dest
and	addr		dest
const		value	dest
d	addr		dest
debug	addr	numtoken	numtoken{dest}numtoken [label]
d1	addr		dest
div	addr		dest
eq	addr		dest
fgate	addr		dest
ge	addr		dest
gt	addr		dest
halt	addr		
ifree	addr		3{dest}3
iread	addr		dest
iwrite	addr		dest
l	addr	codeblock	dest
l1	addr		dest
le	addr		dest
link	addr		2{dest}2
lt	addr		dest
mul	addr		dest
ne	addr		dest
neg	addr		dest
not	addr		dest
or	addr		dest
output	addr	numtoken	[label]
pbeg	addr	numarg	numarg{dest}numarg pend-dest
pend	addr	numarg	
sqrt	addr		dest
sub	addr		dest
switch	addr		true-dest false-dest
tgate	addr		dest
trace f1 .. f18			
	where fn is 0 for off		

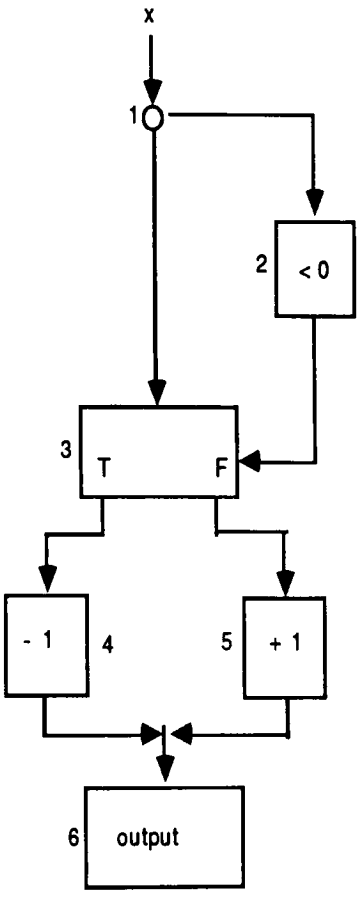
	1 for on
f1	trace instruction store
f2	trace token queue arrival
f3	trace token queue after arrival
f4	trace token queue departure
f5	trace token queue after departure
f6	trace token set queue arrival
f7	trace token set queue after arrival
f8	trace token set queue departure
f9	trace token set queue after departure
f10	trace token store before arrival
f11	trace token store arrival
f12	trace token store after arrival
f13	trace operation packet queue arrival
f14	trace operation packet queue after arrival
f15	trace operation packet queue departure
f16	trace operation packet queue after departure
f17	trace processor arrival
f18	trace processor departure

Figure 27 shows a simple dataflow graph and its corresponding assembler program. The node addresses are assigned arbitrarily, in this case from left to right, top to bottom.

Figure 27. Data Flow Graph and Assembler Program

if x < 0 then y = x - 1 else y = x + 1

Data Flow Graph :



Assembly Program:

```
#*****
# Assembly program for data flow graph
#*****

link      1                2 1    3 1
lt        2                3 2
switch    3                4 1    5 1
sub       4                6 1
add       5                6 1
output    6        1       y=

const     0        2 2     # place const 0 at addr 2 port 2
const     1        4 2
const     1        5 2
```

4.2.1. Operator Instructions

ABS addr dest

addr - address in the graph
dest - result token destination address and port

Find absolute value of port 1.

ADD addr dest

addr - address in the graph
dest - result token destination address and port

Add port 1 to port 2.

DIV addr dest

addr - address in the graph
dest - result token destination address and port

Divide port 1 by 2.

MUL addr dest

addr - address in the graph
dest - result token destination address and port

Multiply port 1 by port 2.

NEG addr dest

addr - address in the graph
dest - result token destination address and port

Multiply port 1 by -1.

SQRT addr dest

addr - address in the graph
dest - result token destination address and port

Find square root of port 1.

SUB addr dest

addr - address in the graph
dest - result token destination address and port

Subtract port 2 from port 1.

4.2.2. Predicate Instructions

EQ addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is equal to port 2;
false otherwise.

GE addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is greater than or equal to port 2;
false otherwise.

GT addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is greater than port 2;
false otherwise.

LE addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is less than or equal to port 2;
false otherwise.

LT addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is less than port 2;
false otherwise.

NE addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is not equal to port 2;
false otherwise.

4.2.3. Boolean Instructions

AND addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 and port 2 are true;
false otherwise.

NOT addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is false;
false otherwise.

OR addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1, port 2, or both are true;
false otherwise.

4.2.4. Branch Instructions

FGATEaddr dest

addr - address in the graph
dest - result token destination address and port

Send port 1 if port 2 is false; otherwise absorb port 1.

HALT

Halt program execution.

LINK addr 2{dest}2

addr - address in the graph
dest - result token destination address and port;

Replicate port 1 token; send the token copies to dest 1 and 2.

SWITCH addr true-dest false-dest

addr - address in the graph
true-dest - result token destination address and port; if true control signal is received
false-dest - result token destination address and port; if false control signal is received

Send port 1 to true-dest if port 2 is true, send port 1 to false-dest if port 2 is false.

TGATE addr dest

addr - address in the graph
dest - result token destination address and port

Send port 1 if port 2 is true; otherwise absorb port 1.

4.2.5. Loop Instructions

D addr dest

addr - address in the graph
dest - result token destination address and port

Increment port 1 initiation number.

D1 addr dest

addr - address in the graph
dest - result token destination address and port

Reset port 1 initiation number to 1.

L addr codeblock dest

addr - address in the graph
codeblock - loop code block number
dest - result token destination address and port

Add codeblock to port 1 activity name.

L1 addr dest

addr - address in the graph
dest - result token destination address and port

Remove codeblock from port 1 activity name.

4.2.6. Procedure Instructions

A addr numarg A1-addr PBEG-addr

addr - address in the graph
numarg - number of procedure arguments
A1-addr - A1 instruction address
PBEG-addr - procedure PBEG instruction address

Send arguments in ports 1 to numarg to PBEG-addr;
send A1-addr (return address) to PBEG-addr;
add addr to token activity names.

A1 addr numarg numarg{dest}numarg

addr - address in the graph
numarg - number of procedure arguments
dest - result token destination address and port

Send arguments in ports 1 to numarg to corresponding dest;
Remove A addr from token activity names.

PBEG addr numarg numarg{dest}numarg PEND-dest

addr - address in the graph
numarg - number of procedure arguments
dest - result token destination address and port
PEND-dest - procedure PEND address and port

Distribute port tokens 1 to numarg to procedure instructions;
send A1-addr (return address) to PEND-dest.

PEND addr numarg

addr - address in the graph
numarg - number of procedure arguments

Send result parameters in ports 1 to numarg to the A1 instruction address.

4.2.7. I-structure Instructions

IFREE addr 3{dest}3

addr - address in the graph
dest - result token destination address and port

Reinitialize i-structure cell located by port 1 and port 2; send port 1 to dest 1, port 2 to dest 2, and port 3 (icell value) to dest 3.

IREAD addr dest

addr - address in the graph
dest - result token destination address and port

Send a copy of the value stored at the i-structure cell located by port 1 and port 2 to dest.

IWRITE addr dest

addr - address in the graph
dest - result token destination address and port

Store the value in port 3 at the i-structure cell located by port 1 and port 2.

4.2.8. Output Instructions

DEBUG addr numtoken numtoken{dest}numtoken [label]

addr - address in the graph
numtoken - number of tokens to be output
dest - result token destination address and port
label - character string prefix to output tokens

If a label exists output the label; output the tokens in ports 1 to numtoken; send the tokens in ports 1 to numtoken to the corresponding dest.

OUTPUT addr numtoken [label]

addr - address in the graph
numtoken - number of tokens to be output
label - character string prefix to output tokens

If a label exists output the label; output the tokens in ports 1 to numtoken.

4.2.9. Comment and Constant Instructions

\# comment text

Characters from the # to the end of the line are ignored by the assembler, and may be used for program comments.

CONST value dest

value - constant value
dest - instruction address and port where constant value is to be stored

Stores value at dest.

4.3. Input File

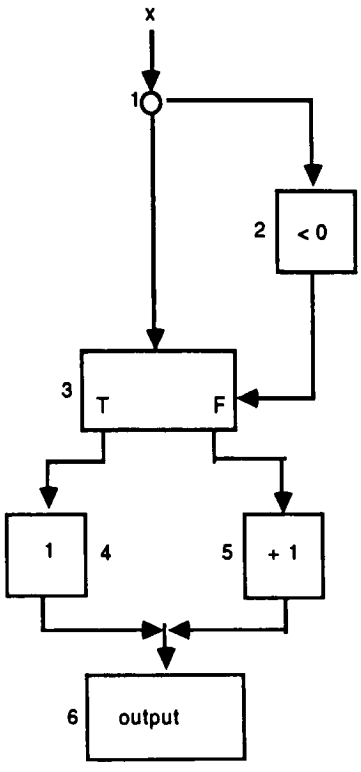
The input file is used to put the simulator in motion. The input file must contain the data required to enable one or more instructions in the program, such that the remaining program instructions will be triggered leading to the program results.

The input file contains a data value and its instruction destination (address and port).

Figure 28. Data Flow Graph and Input File

if x < 0 then y = x - 1 else y = x + 1

Data Flow Graph :



Assembly Program:

```
#*****
# Assembly program for data flow graph
#*****

link      1                2 1    3 1
lt        2                3 2
switch    3                4 1    5 1
sub       4                6 1
add       5                6 1
output    6          1     y=

const     0          2 2   # place const 0 at addr 2 port 2
const     1          4 2
const     1          5 2
```

Example 1.

Input file		Output file
Data	addr/port	
2	1 1	y= 3

Example 2.

Input file		Output file
Data	addr/port	
-2	1 1	y= -3

4.4. Debugging

The simulator provides two kinds of debugging tools. The DEBUG and OUTPUT instructions are used for debugging assembler programs written by the application programmer. The TRACE instruction is used by the simulator's maintainer to debug the simulator itself.

Debugging an application program is best approached by studying the program graph, and choosing nodes where knowing the token values will provide clues to solving the problem. The DEBUG or OUTPUT instruction can be inserted to display these token values. If the DEBUG instruction is used then the tokens will be propagated to the next node in the program graph, the OUTPUT instruction will absorb the tokens.

The TRACE instruction, though it may provide some insight into the application program, is used to debug the simulator itself. The TRACE instruction has available eighteen options of which any combination may be employed.

TRACE f1 f2 f3 .. f18

fn is enabled if set to 1
fn is disabled if set to 0

All trace results are written to the "trace.lis" file.
This file has a tendency toward being very large.

Whereas the DEBUG and OUTPUT instructions only display the token data, the TRACE instruction also displays the token tag.

f1	displays the machine code assembled from the application program.
f2	displays all tokens that arrive at the token queue.
f3	displays the token queue after a token arrival.
f4	displays all tokens that depart from the token queue.
f5	displays the token queue after a token departure.
f6	displays all token sets that arrive at the token set queue.
f7	displays the token set queue after a token set arrival.
f8	displays all token sets that depart from the token set queue.
f9	displays the token set queue after a token set departure.
f10	displays the token store, prior to applying the matching algorithm, when a token arrives at the match

unit.

f11 displays the token that has arrived at the match unit.

f12 displays the token store after the matching algorithm has been applied to the token.

f13 displays all operation packets that arrive at the operation packet queue.

f14 displays the operation packet queue after an operation packet arrival.

f15 displays all operation packets that depart from the operation packet queue.

f16 displays the operation packet queue after a departure.

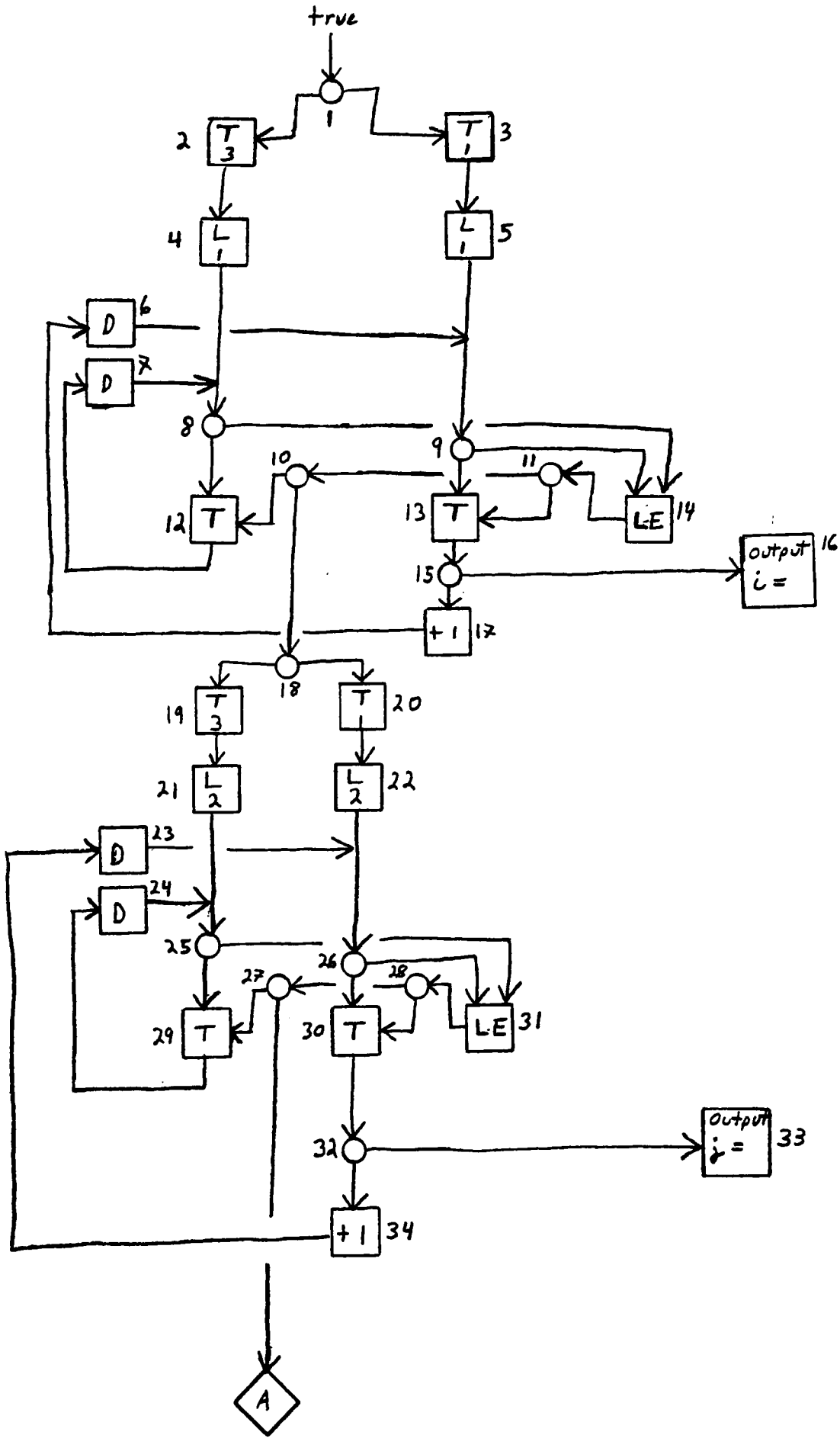
f17 displays all operation packets that arrive at the processing unit.

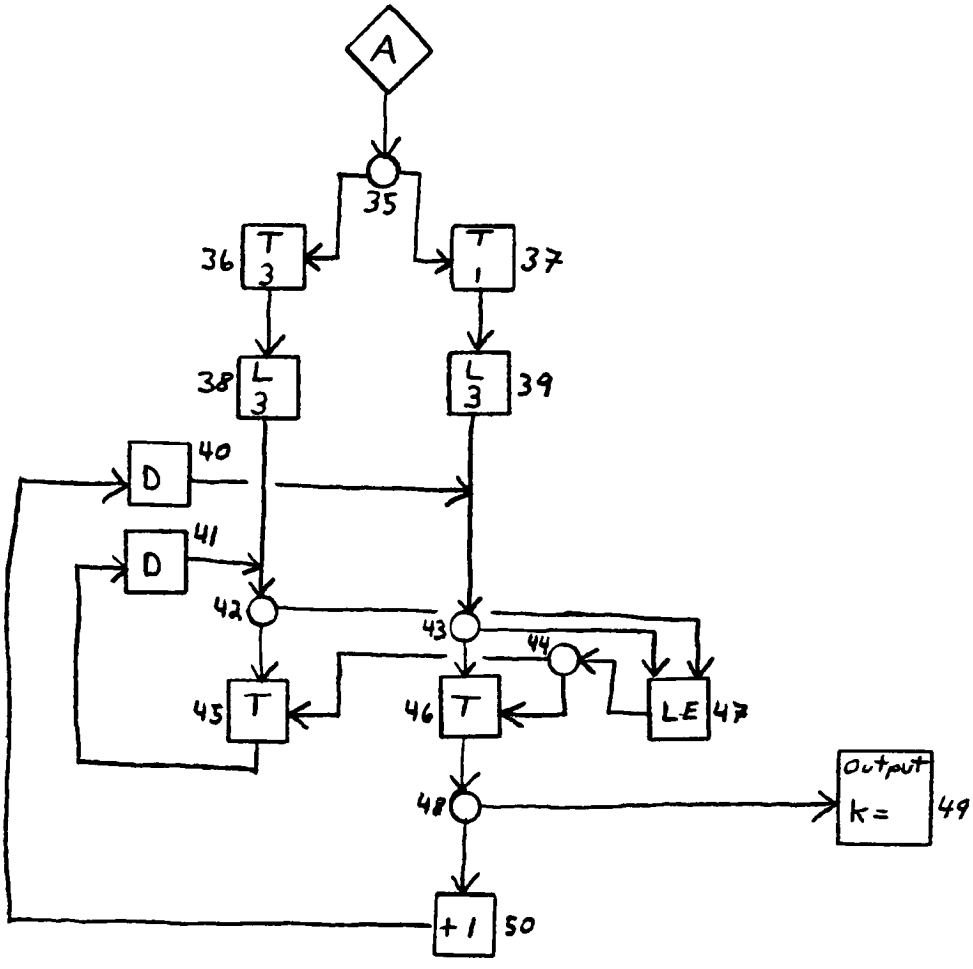
f18 displays all result tokens that depart from the processing unit.

4.5. Sample Programs

4.5.1. Nested Loop Example

```
mainline
  for i = 1 to 3
    output("i=",i)
    for j = 1 to 3
      output("j=",j)
      for k = 1 to 3
        output("k=",k)
      end for
    end for
  end for
end mainline
```


true
1
2
T 3
3
T 1
4
L 1
5
L 1
D
6
D
7
8
9
10
11
12
T
13
T
LE
14
output i =
16
15
+1
17
18
19
T 3
T
20
21
L 2
L 2
22
D
23
D
24
25
26
27
28
29
T
30
T
LE
31
32
output j =
33
+1
34
A


A
35
36
T
3
37
T
1
38
L
3
39
L
3
40
D
41
D
42
43
44
45
T
46
T
LE
47
48
output
k=
49
+1
50

```
# Sample Program One
#   Nested loop

# I loop code block

link      1              2 2      3 2
tgate     2              4 1
tgate     3              5 1
l         4       1      8 1
l         5       1      9 1
d         6              9 1
d         7              8 1
link      8              12 1     14 2
link      9              13 1     14 1
link      10             12 2     18 1
link      11             13 2     10 1
tgate     12             7 1
tgate     13             15 1
le        14             11 1
link      15             16 1     17 1
output    16      1      i=
add       17             6 1

# I to J loop connection

link      18             19 2     20 2
tgate     19             21 1
tgate     20             22 1

# J loop code block

l         21      2      25 1
l         22      2      26 1
d         23             26 1
d         24             25 1
link      25             29 1     31 2
link      26             30 1     31 1
link      27             29 2     35 1
link      28             27 1     30 2
tgate     29             24 1
tgate     30             32 1
le        31             28 1
link      32             33 1     34 1
output    33      1      j=
add       34             23 1

# J to K loop connection

link      35             36 2     37 2
tgate     36             38 1
tgate     37             39 1

# K loop code block
```

```
l       38      3       42 1
l       39      3       43 1
d       40              43 1
d       41              42 1
link    42              45 1    47 2
link    43              46 1    47 1
link    44              45 2    46 2
tgate   45              41 1
tgate   46              48 1
le      47              44 1
link    48              49 1    50 1
output  49      1       k=
add     50              40 1

# instruction constants

const           3       2 1
const           1       3 1
const           1       17 2
const           3       19 1
const           1       20 1
const           1       34 2
const           3       36 1
const           1       37 1
const           1       50 2
```

INPUT FILE

```
value   node/port
1        1 1
```

OUTPUT FILE

```
i=  1.000000
i=  2.000000
j=  1.000000
i=  3.000000
j=  1.000000
j=  2.000000
k=  1.000000
j=  1.000000
j=  2.000000
k=  1.000000
j=  3.000000
k=  1.000000
k=  2.000000
j=  2.000000
k=  1.000000
j=  3.000000
k=  1.000000
k=  2.000000
k=  1.000000
k=  2.000000
k=  3.000000
j=  3.000000
k=  1.000000
```

k= 2.000000

k= 1.000000

k= 2.000000

k= 3.000000

k= 2.000000

k= 3.000000

k= 1.000000

k= 2.000000

k= 3.000000

k= 2.000000

k= 3.000000

k= 3.000000

k= 2.000000

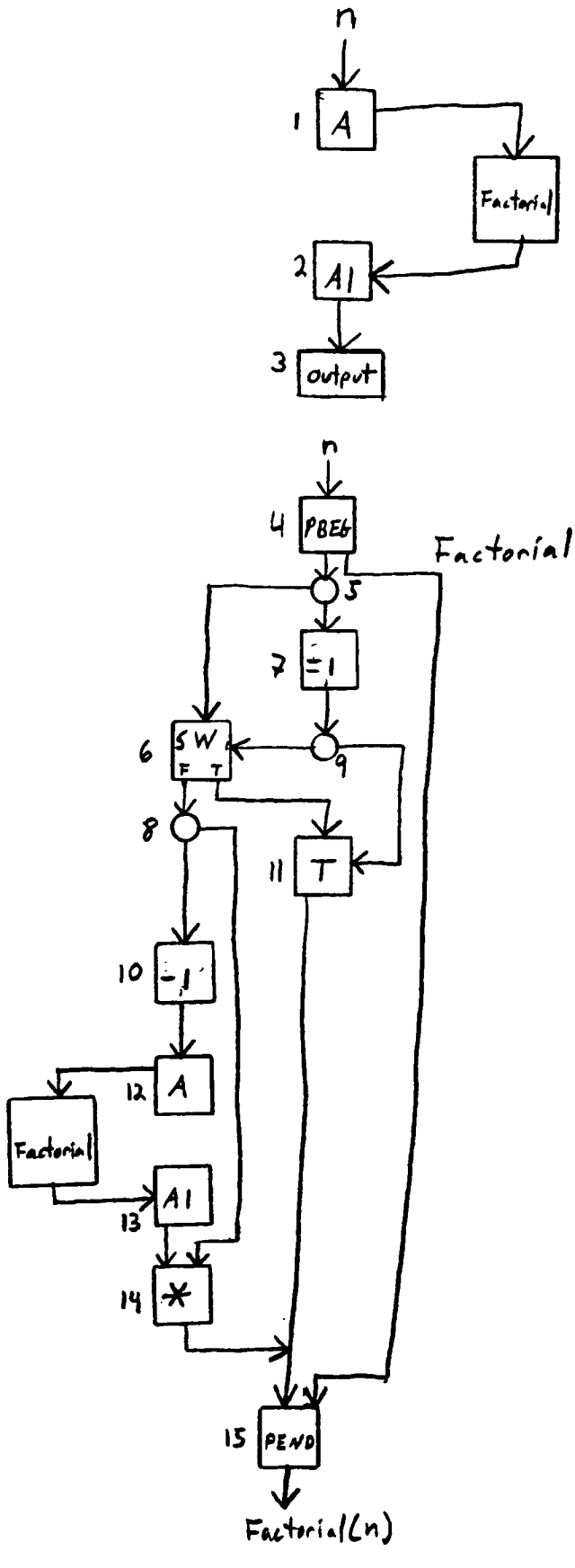
k= 3.000000

k= 3.000000

k= 3.000000

4.5.2. Recursive Factorial Example

```
mainline
  input(n)
  factorial(n)
  output("factorial=",n)
end mainline

factorial(n)
  if (n = 1) then
    return((n)
  else
    return(n * factorial(n - 1))
  end if
end factorial
```


n
1
A
Factorial
2
A1
3
output
n
4
PBEb
Factorial
5
7
-1
6
SW
F
T
9
8
11
T
10
-1
12
A
Factorial
13
A1
14
*
15
PEND
Factorial(n)

```
# Sample Program Two
#   Recursive factorial

# main program

a         1       1       2       4
a1        2       1       3 1
output    3       1       factorial=

# factorial subprogram (recursively calls itself)

pbeg      4       1       5 1     15 2
link      5               6 1     7 1
switch    6               11 1    8 1
eq        7               9 1
link      8               10 1    14 2
link      9               6 2     11 2
sub       10              12 1
tgate     11              15 1
a         12      1       13      4
a1        13      1       14 1
mul       14              15 1
pend      15      1

# instruction constants

const             1       7 2
const             1       10 2
```

INPUT FILE

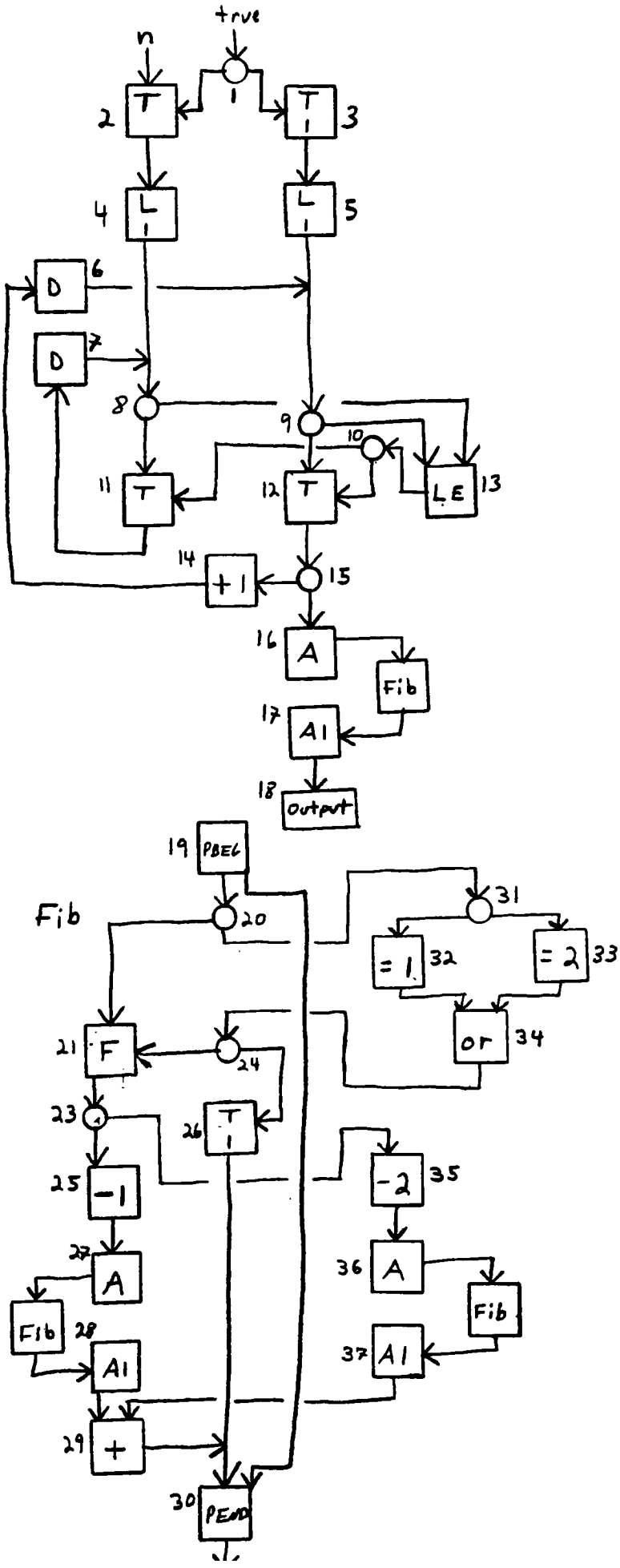
```
value     node/port
9         1 1
```

OUTPUT FILE

```
          factorial=   362880.000000
```

4.5.3. Fibbonaci Series Example

```
mainline
  input(n)
  for i = 1 to n
    fib(n)
    output(n)
  end for
end mainline

fib(n)
  if ((n = 1) or (n = 2)) then
    return(1)
  else
    return(fib(n - 1) + fib(n - 2))
  end if
end fib
```


true
n
1
2 T
3 T
4 L
5 L
6 D
7 D
8
9
10
11 T
12 T
13 LE
14 +1
15
16 A
Fib
17 A1
18 output
19 PBEG
Fib
20
31
= 1 32
= 2 33
or 34
21 F
24
23
26 T
25 -1
-2 35
27 A
36 A
Fib
Fib 28
A1
37 A1
29 +
30 PEND

```
# Sample Three
#   Fibbonaci Series

# main program

link    1               2 2     3 2
tgate   2               4 1
tgate   3               5 1
l       4       1       8 1
l       5       1       9 1
d       6               9 1
d       7               8 1
link    8               11 1    13 2
link    9               12 1    13 1
link    10              11 2    12 2
tgate   11              7 1
tgate   12              15 1
le      13              10 1
add     14              6 1
link    15              14 1    16 1
a       16      1       17      19
a1      17      1       18 1
output  18      1

# fib subprogram

pbeg    19      1       20 1    30 2
link    20              21 1    31 1
fgate   21              23 1
link    23              25 1    35 1
link    24              21 2    26 2
sub     25              27 1
tgate   26              30 1
a       27      1       28      19
a1      28      1       29 1
add     29              30 1
pend    30      1
link    31              32 1    33 1
eq      32              34 1
eq      33              34 2
or      34              24 1
sub     35              36 1
a       36      1       37      19
a1      37      1       29 2

# instruction constants

const           1       3 1
const           1       14 2
const           1       25 2
const           1       26 1
const           1       32 2
const           2       33 2
```

```
const           2       35 2
```

INPUT FILE

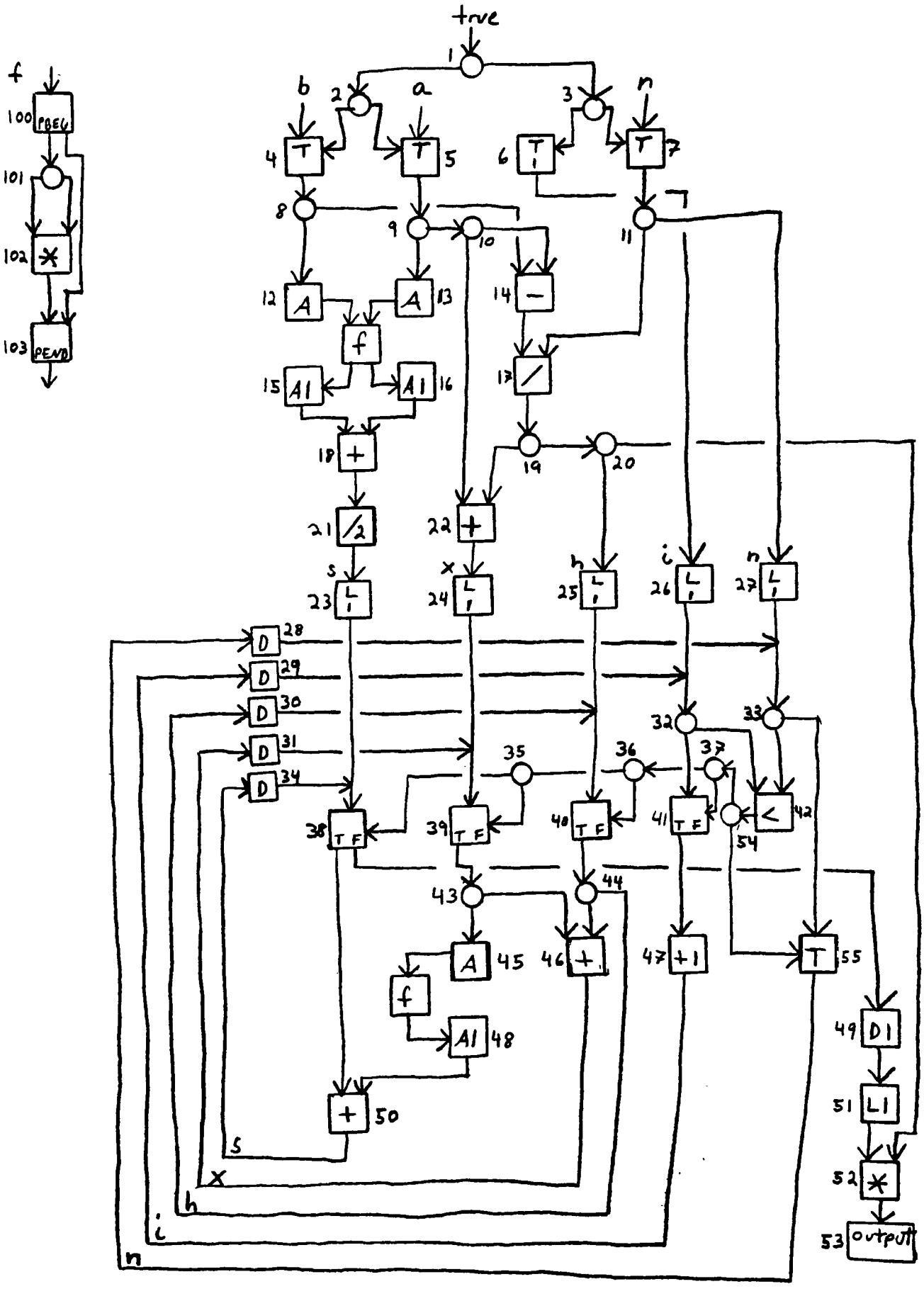
```
value   node/port
1        1 1
9        2 1
```

OUTPUT FILE

```
1.000000

1.000000

2.000000

3.000000

5.000000

8.000000

13.000000

21.000000

34.000000
```

4.5.4. Trapezoid Rule Example

```
mainline
  input(a,b,n)
  h = (b - a) / n
  s = (f(a) + f(b)) / 2
  x = a + h
  for i = 1 to n - 1
    s = s + f(x)
    x = x + h
  end for
  s = s * h
  output(s)
end mainline

f(x)
  return(x * x)
end f
```


f
100 PBEG
101
102 *
103 PEND
true
1
b
2
a
3
n
4 T
5 T
6 T
7 T
8
9
10
11
12 A
13 A
14 −
f
15 AI
16 AI
17 /
18 +
19
20
21 /2
22 +
s
23 L
x
24 L
h
25 L
i
26 L
n
27 L
D 28
D 29
D 30
D 31
D 34
32
33
35
36
37
38 T F
39 T F
40 T F
41 T F
42 <
54
43
44
A 45
46 +
47 +1
T 55
f
AI 48
+ 50
49 DI
51 LI
52 *
53 output
s
x
h
i
n

```
# Sample Program Four
#   Trapezoid rule

# mainline

link      1              2 1      3 1
link      2              4 2      5 2
link      3              6 2      7 2
tgate     4              8 1
tgate     5              9 1
tgate     6              26 1
tgate     7              11 1
link      8              12 1     14 1
link      9              13 1     10 1
link      10             22 1     14 2
link      11             17 2     27 1
a         12    1        15       100
a         13    1        16       100
sub       14             17 1
a1        15    1        18 1
a1        16    1        18 2
div       17             19 1
add       18             21 1
link      19             22 2     20 1
link      20             25 1     52 2
div       21             23 1
add       22             24 1
l         23    1        38 1
l         24    1        39 1
l         25    1        40 1
l         26    1        32 1
l         27    1        33 1
d         28             33 1
d         29             32 1
d         30             40 1
d         31             39 1
link      32             41 1     42 1
link      33             42 2     55 1
d         34             38 1
link      35             38 2     39 2
link      36             35 1     40 2
link      37             36 1     41 2
switch    38             50 1     49 1
switch    39             43 1     0 0
switch    40             44 1     0 0
switch    41             47 1     0 0
lt        42             54 1
link      43             45 1     46 1
link      44             46 2     30 1
a         45    1        48       100
add       46             31 1
add       47             29 1
a1        48    1        50 2
```

```
d1      49              51 1
add     50              34 1
l1      51              52 1
mul     52              53 1
output  53      1
link    54              37 1    55 2
tgate   55              28 1

# function

pbeg    100     1       101 1   103 2
link    101             102 1   102 2
mul     102             103 1
pend    103     1

# instruction constants

const           1       6 1
const           2       21 2
const           1       47 2
```

INPUT FILE

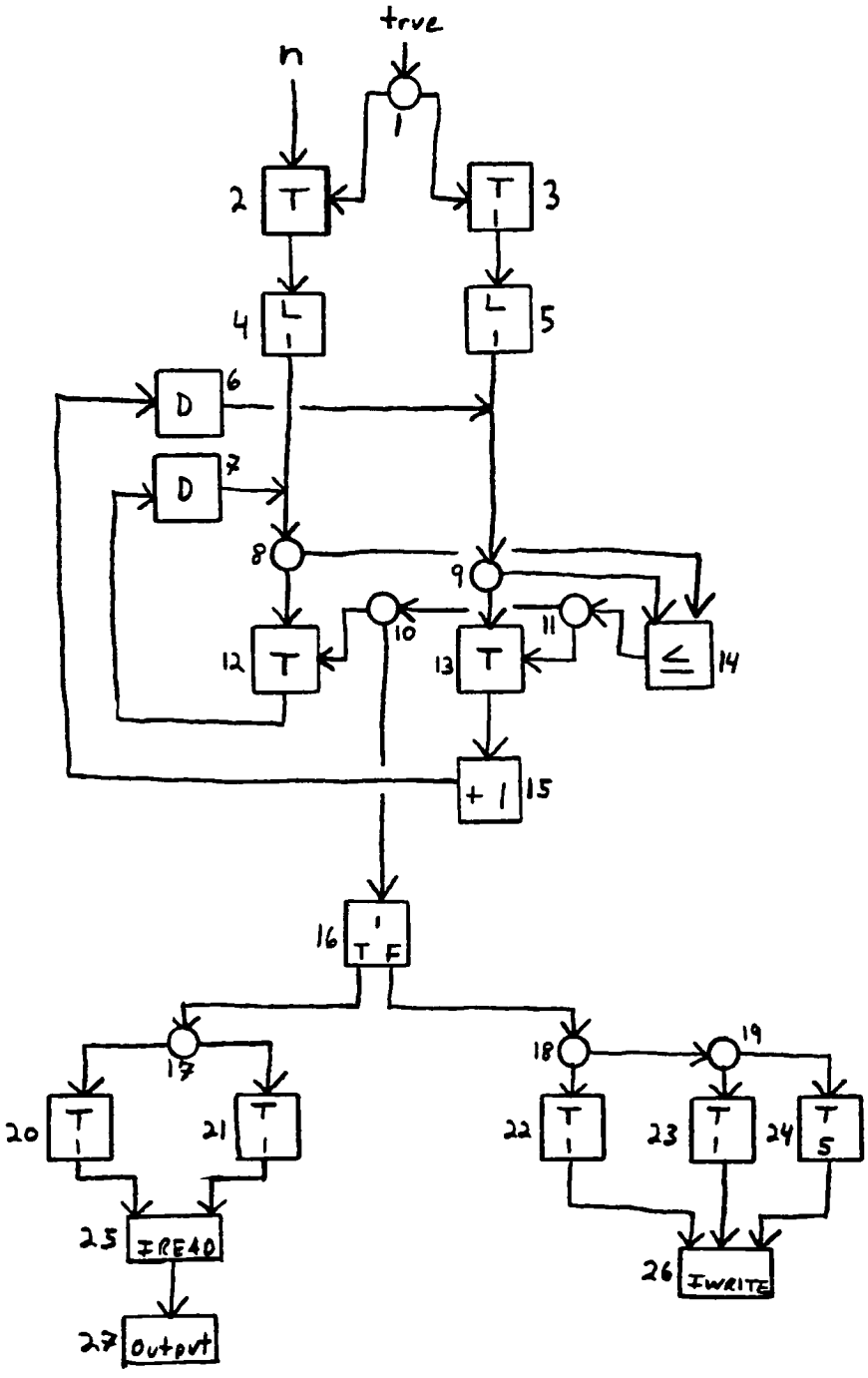
```
value   node/port
1         1 1
2         5 1
4         4 1
3         7 1
```

OUTPUT FILE

```
  18.814817
```

4.5.5. I-Structure Example

```
mainline
  input(n)
  for i = 1 to n
    output("v=",icell(1,1))
  end for
  icell(1,1) = 5
end mainline
```


true
n
1
2
T
3
T
4
L
5
L
6
D
7
D
8
9
10
11
12
T
13
T
14
≤
15
+ 1
16
T F
17
18
19
20
T
21
T
22
T
23
T
24
T
5
25
IREAD
26
IWRITE
27
output

```
# Sample Program Five
#   I-structures

# load initial values into Istructure

link      1              2 2      3 2
tgate     2              4 1
tgate     3              5 1
l         4       1      8 1
l         5       1      9 1
d         6              9 1
d         7              8 1
link      8              14 2     12 1
link      9              14 1     13 1
link      10             12 2     16 2
link      11             10 1     13 2
tgate     12             7 1
tgate     13             15 1
le        14             11 1
add       15             6 1
switch    16             17 1     18 1
link      17             20 2     21 2
link      18             22 2     19 1
link      19             23 2     24 2
tgate     20             25 1
tgate     21             25 2
tgate     22             26 1
tgate     23             26 2
tgate     24             26 3
iread     25             27 1
iwrite    26             0 0      #bit bucket
output    27      1      v=

# instruction constants

const             1      3 1
const             1      15 2
const             1      16 1
const             1      20 1
const             1      21 1
const             1      22 1
const             1      23 1
const             5      24 1
```

INPUT FILE

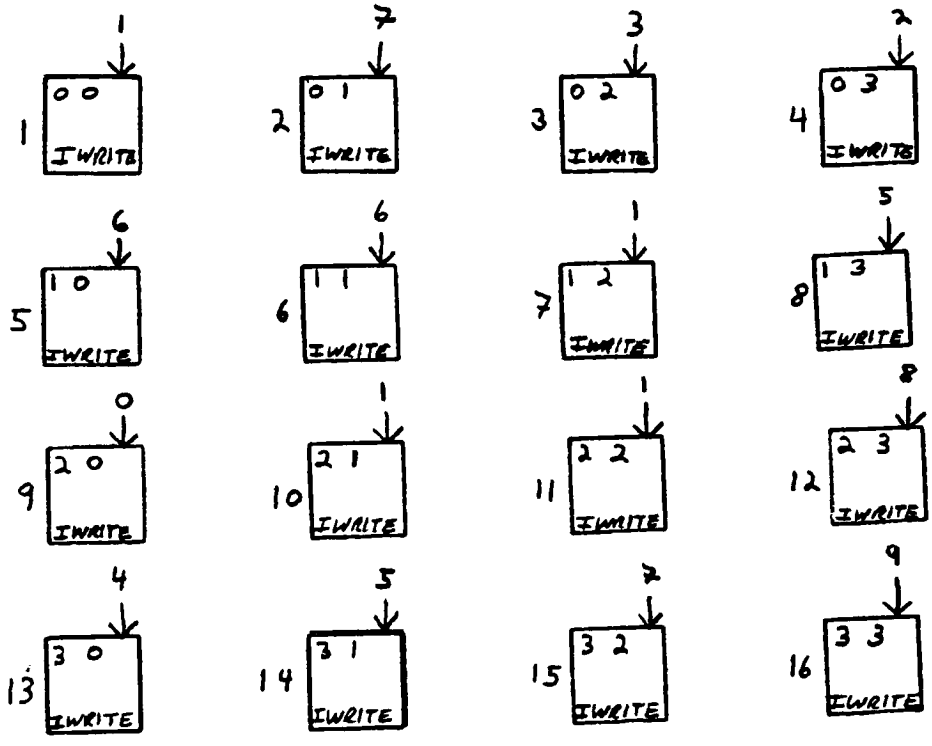
value	node/port
1	1 1
10	2 1

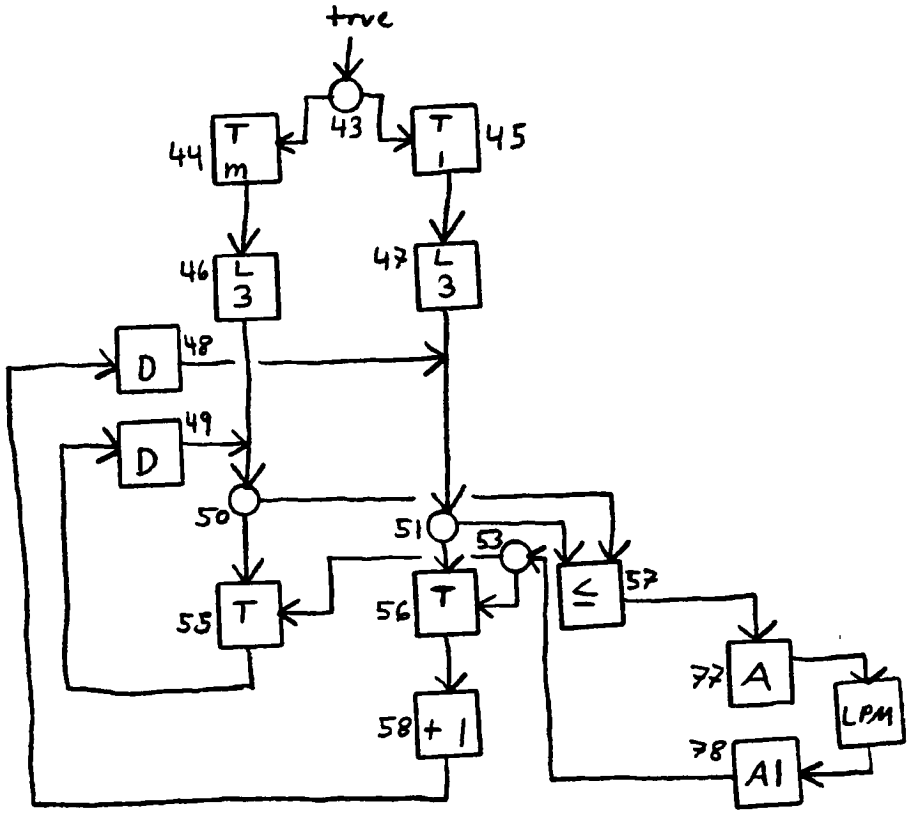
OUTPUT FILE

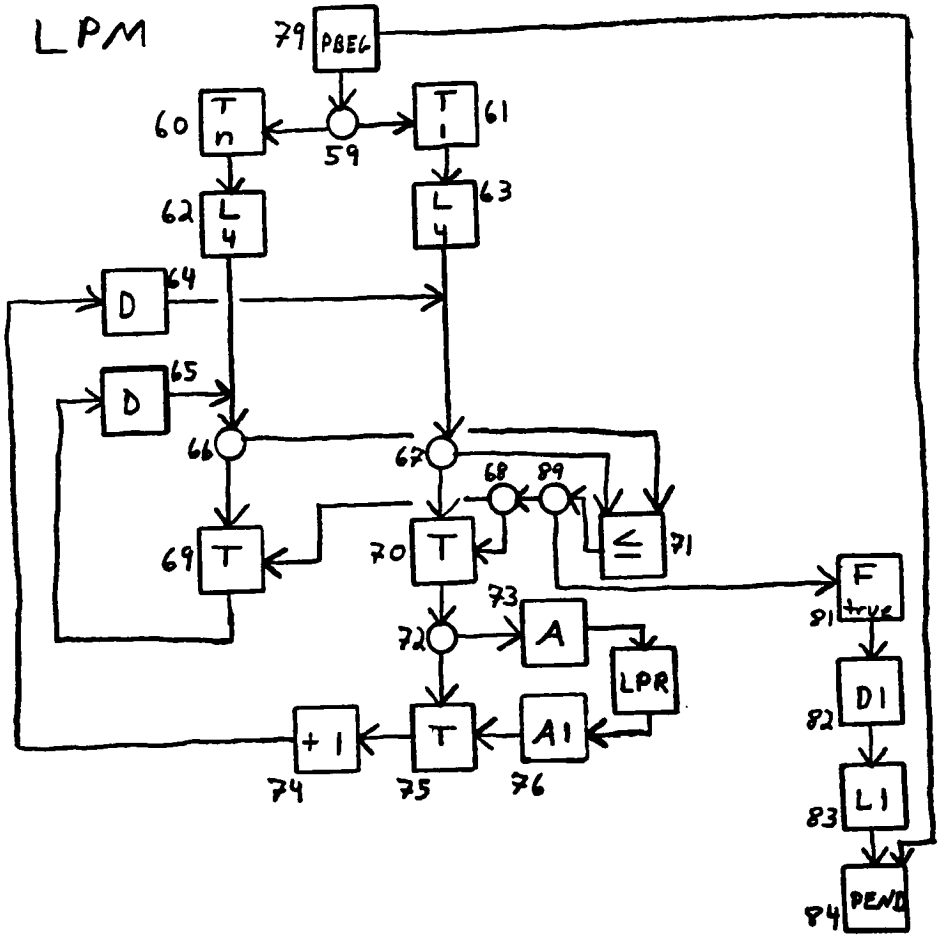
```
v=  5.000000
v=  5.000000
v=  5.000000
v=  5.000000
v=  5.000000
v=  5.000000
v=  5.000000
v=  5.000000
v=  5.000000
v=  5.000000
```

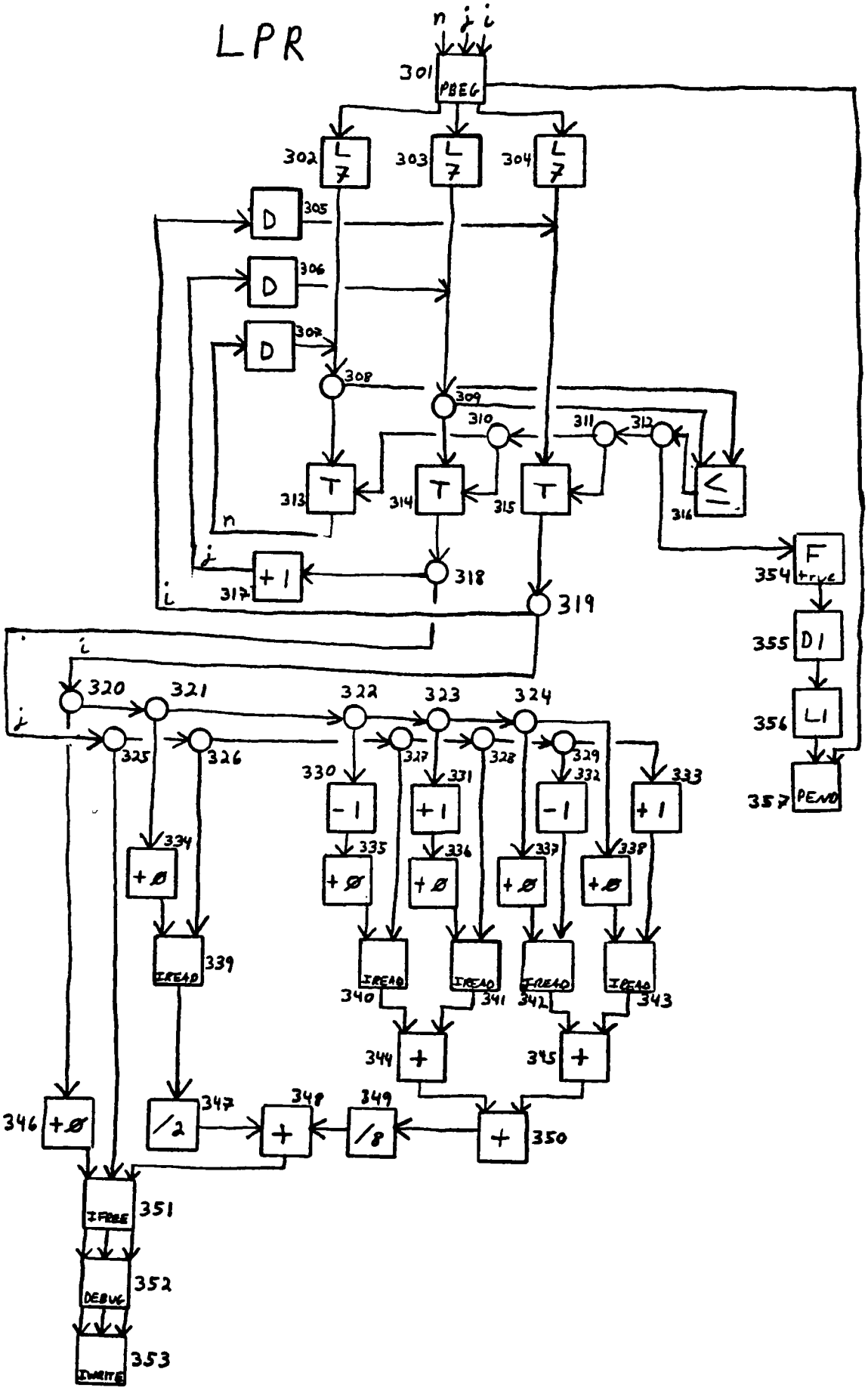
4.5.6. Laplace Transform Example

```
mainline
  input(
    icell(0,0),icell(0,1),icell(0,2),icell(0,3),
    icell(1,0),icell(1,1),icell(1,2),icell(1,3),
    icell(2,0),icell(2,1),icell(2,2),icell(2,3),
    icell(3,0),icell(3,1),icell(3,2),icell(3,3)
    )
  m = 3
  for k = 1 to m
    lpm()
  end for
end mainline

lpm()
  n = 2
  for i = 1 to n
    lpr(i)
  end for
end lpm

lpr(i)
  n = 2
  for j = 1 to n
    icell(i,j) = icell(i,j) / 2 +
      (icell(i-1,j) + icell(i+1,j) +
      icell(i,j-1) + icell(i,j+1)) / 8
    output("<i j v>",i,j,icell(i,j))
  end for
end lpr
```


1
0 0
IWRITE
1
7
0 1
IWRITE
2
3
0 2
IWRITE
3
2
0 3
IWRITE
4
6
1 0
IWRITE
5
6
1 1
IWRITE
6
1
1 2
IWRITE
7
5
1 3
IWRITE
8
0
2 0
IWRITE
9
1
2 1
IWRITE
10
1
2 2
IWRITE
11
8
2 3
IWRITE
12
4
3 0
IWRITE
13
5
3 1
IWRITE
14
2
3 2
IWRITE
15
9
3 3
IWRITE
16


true
43
T m
44
T 1
45
L 3
46
L 3
47
D
48
D
49
50
51
53
T
55
T
56
≤
57
A
77
LPM
78
A1
+ 1
58


LPM
79
PBEG
T n
60
T 1
61
59
L 4
62
L 4
63
D
64
D
65
66
67
68
69
T
70
T
89
≤
71
F true
91
73
A
72
LPR
+ 1
74
T
75
A1
76
D1
82
L1
83
PEND
84


LPR
n j i
301
PBEG
302
303
304
L 7
D
305
306
307
308
309
310
311
312
T
313
314
315
≤
316
n
j
i
+1
317
318
319
F
true
354
D1
355
L1
356
PEND
357
i
320
321
322
323
324
325
326
327
328
329
330
331
332
333
-1
+1
+Ø
334
335
336
337
338
339
IREAD
340
341
342
343
+
344
345
346
/2
347
348
/8
349
350
IFREE
351
DEBUG
352
IWRITE
353

```
# Sample Program 6
#   Laplace Transform

# Set up Istructure cell indices

iwrite    1              0 0 # null address
iwrite    2              0 0
iwrite    3              0 0
iwrite    4              0 0
iwrite    5              0 0
iwrite    6              0 0
iwrite    7              0 0
iwrite    8              0 0
iwrite    9              0 0
iwrite   10              0 0
iwrite   11              0 0
iwrite   12              0 0
iwrite   13              0 0
iwrite   14              0 0
iwrite   15              0 0
iwrite   16              0 0

const          0         1 1
const          0         1 2
const          0         2 1
const          1         2 2
const          0         3 1
const          2         3 2
const          0         4 1
const          3         4 2
const          1         5 1
const          0         5 2
const          1         6 1
const          1         6 2
const          1         7 1
const          2         7 2
const          1         8 1
const          3         8 2
const          2         9 1
const          0         9 2
const          2        10 1
const          1        10 2
const          2        11 1
const          2        11 2
const          2        12 1
const          3        12 2
const          3        13 1
const          0        13 2
const          3        14 1
const          1        14 2
const          3        15 1
const          2        15 2
const          3        16 1
```

```
const           3       16 2

# perform Laplace transform m times

# k loop 1 to m

link    43              44 2    45 2
tgate   44              46 1
tgate   45              47 1
l       46      3       50 1
l       47      3       51 1
d       48              51 1
d       49              50 1
link    50              55 1    57 2
link    51              56 1    57 1
link    53              55 2    56 2
tgate   55              49 1
tgate   56              58 1
le      57              77 1
add     58              48 1

# laplace matrix (lpm) transform subroutine

# i loop 1 to n

link    59              60 2    61 2
tgate   60              62 1
tgate   61              63 1
l       62      4       66 1
l       63      4       67 1
d       64              67 1
d       65              66 1
link    66              69 1    71 2
link    67              70 1    71 1
link    68              69 2    70 2
tgate   69              65 1
tgate   70              72 1
le      71              80 1
link    72              73 3    75 1
a       73      3       76      301
add     74              64 1
tgate   75              74 1
a1      76      1       75 2
a       77      1       78      79
a1      78      1       53 1
pbeg    79      1       59 1    84 2
link    80              68 1    81 2
fgate   81              82 1
d1      82              83 1
l1      83              84 1
pend    84      1

# laplace row (lpr) transform subroutine
```

j loop 1 to n

pbeg	301	3	302 1	303 1	304 1	357 2
l	302	7	308 1			
l	303	7	309 1			
l	304	7	315 1			
d	305		315 1			
d	306		309 1			
d	307		308 1			
link	308		313 1	316 2		
link	309		314 1	316 1		
link	310		313 2	314 2		
link	311		310 1	315 2		
link	312		311 1	354 2		
tgate	313		307 1			
tgate	314		318 1			
tgate	315		319 1			
le	316		312 1			
add	317		306 1			
link	318		317 1	325 1		
link	319		305 1	320 1		

perform transform

link	320	321 1	346 1
link	321	322 1	334 1
link	322	323 1	330 1
link	323	324 1	331 1
link	324	337 1	338 1
link	325	326 1	351 2
link	326	327 1	339 2
link	327	328 1	340 2
link	328	329 1	341 2
link	329	332 1	333 1
sub	330	335 1	
add	331	336 1	
sub	332	342 2	
add	333	343 2	
add	334	339 1	
add	335	340 1	
add	336	341 1	
add	337	342 1	
add	338	343 1	
iread	339	347 1	
iread	340	344 1	
iread	341	344 2	
iread	342	345 1	
iread	343	345 2	
add	344	350 1	
add	345	350 2	
add	346	351 1	
div	347	348 1	
add	348	351 3	

```
div     349             348 2
add     350             349 1
ifree   351             352 1   352 2   352 3
debug   352     3       353 1   353 2   353 3   <i j v>
iwrite  353             0 0
fgate   354             355 1
d1      355             356 1
l1      356             357 1
pend    357     1

# instruction constants

const           2       60 1            # n
const           2       73 1            # n
const           3       44 1            # m
const           0       334 2           # icell base address
const           0       335 2           # icell base address
const           0       336 2           # icell base address
const           0       337 2           # icell base address
const           0       338 2           # icell base address
const           0       346 2           # icell base address

const           1       45 1
const           1       58 2
const           1       61 1
const           1       73 2
const           1       74 2
const           1       81 1
const           1       317 2
const           1       330 2
const           1       331 2
const           1       332 2
const           1       333 2
const           2       347 2
const           8       349 2
const           1       354 1
```

INPUT FILE

value	node/port
1	1 3
7	2 3
3	3 3
2	4 3
6	5 3
6	6 3
1	7 3
5	8 3
0	9 3
1	10 3
1	11 3
8	12 3
4	13 3
5	14 3
7	15 3
9	16 3
1	43 1

OUTPUT FILE

<i j v>	1.000000	1.000000	4.875000
<i j v>	1.000000	2.000000	2.234375
<i j v>	2.000000	1.000000	1.859375
<i j v>	2.000000	2.000000	2.886719
<i j v>	1.000000	1.000000	4.574219
<i j v>	1.000000	2.000000	3.049805
<i j v>	2.000000	1.000000	2.487305
<i j v>	2.000000	2.000000	4.010498
<i j v>	1.000000	1.000000	4.604248
<i j v>	1.000000	2.000000	3.601746
<i j v>	2.000000	1.000000	2.945496
<i j v>	2.000000	2.000000	4.698654

4.6. Running the Simulator

The simulator software is located on the Atlantis machine in the sib0331 account under the "dataflow" directory. To run the simulator type "dfw" and respond to the prompts.

```
$dfw
Program file name: <assembler program file>
Input file name: <simulator input file>
Output file name: <simulator result file>
```

If the TRACE instruction is included in the assembler program, the trace output will be in the file "trace.lis".

4.7. Errors

An application program may cause the simulator to abort with one of the following errors.

Error 1: unknown command: address = n

The application program contains an unknown assembler command at address n, or a syntax error prior to address n.

Error 2: instruction store overflow

The application program contains an instruction address beyond the range of the instruction store.

Error 3: file not found

The file name given at the prompt could not be found by the simulator.

Error 4: port collision: address = n: port = m

A token was destined for an instruction port that was already occupied by another token.

Error 5: Istructure[i,j] collision

An IWRITE was issued for an i-structure cell that was not empty.

5. Conclusion

Dataflow machines will have their place as special processors for performing highly parallel, non-volatile applications. Serial applications would not benefit from the high concurrency of dataflow machines, and may even have reduced performance as a result of the token communication network overhead. Debugging dataflow programs is difficult and time consuming; maintenance costs for volatile applications would be prohibitive.

The new simulator addresses the problems that limited the previous simulator developed by [Torsone 1985]. The new simulator, having been written in Modula-2, handles real numbers and allows the programming of a broad range of applications. I-structures have been provided for applications that require data structures. The inclusion of an assembler has made it easier to develop and debug application programs.

The new simulator could be enhanced by improving the token tagging scheme. The simulator implements the token tag as a first-in first-out linked list stored as part of the token [Arvind, Gostelow 1982]. As a result some very long tags may occur, especially when applications consist of deeply recursive algorithms. The performance of the simulator deteriorates rapidly as token tag lengths increase.

Another area of improvement involves the reclamation of i-structure cells. Currently, the reclamation of i-structure cells must be handled by the application programmer. This may be improved by implementing a scheme that makes use of a reference count for each cell, and automatically reclaims the cell when its reference count has been decremented to zero [Arvind, Thomas 1980].

A final suggestion for improving the simulator is the expansion of the simulator from a single-ring architecture to a multi-ring architecture [Gurd, Kirkham, Watson 1985].

Some interesting projects that may build upon the simulator include a graphical front-end, which would allow the application programmer to "draw" the dataflow graph directly on the terminal and have it translated into the simulator's assembler language, and the development of a

compiler for a single-assignment high-level programming language.

Bibliography

[Ackerman 1982]

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

[Agerwala, Arvind 1982]

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

[Arvind, Dertouzos, and Iannucci 1983]

Arvind, Dertouzos, M. and Iannucci, R. "A Multiprocessor Emulation Facility", Technical Report MIT/LCS/TR-302, Laboratory for Computer Science, M.I.T., February, 1983.

[Arvind and Gostelow 1977]

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

[Arvind and Gostelow 1982]

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

[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", Technical 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 Neuman Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM* 21, 8 (August 1978), 613-641.

[Carlson and Hwang 1985]

Carlson, W. and Hwang, K. "Algorithmic Performance of Dataflow Multiprocessors", *Computer* 18 (December 1985), 30-40.

[Clark 1973]

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

[Comte, Hifdi and Syre 1980]

Comte, D., Hifdi N., and Syre, J.C "The Data Driven LAU Multiprocessor System: Results and Perspectives", *Proceedings IFIP Congress* (1980), October 1980, 175-180.

[Davis 1978]

Davis, A.L. "The Architecture and System Methodology of DDM1: A Recursively Structured Data Driven Machine", *Proceedings Fifth Annual Symposium Computer Architecture,* (Palo Alto, California, April 3-5) ACM, New York, 1978, 210-215.

[Davis 1979]

Davis, A.L. "A Data Flow Evaluation System Based on the Concept of Recursive Locality", *Proceedings 1979 National Computer Conference* 1979, 1079-1086.

[Davis and Keller 1982]

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

[Davis and Lowder 1981]

Davis, A.L. and Lowder, S.A. "A Sample Management Application Program in a Graphical Data-Driven Programming Language", *Digest of Papers Compcon Spring 81,* February 1981, 162-167.

[De Marco 1978]

De Marco, T. *Structured Analysis and System Specification,* Yourdon inc., New York, New York, 1978.

[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 Group Note 28-1, Feb. 1977, Laboratory for Computer Science, MIT, Cambridge, Mass.

[Dennis 1980]

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

[Dennis 1984]

Dennis, J.B. "Data Flow Ideas for Supercomputers", *Digest of Papers Compcon Spring 84,* February 1984, 15-19.

[Dennis, Boughton, and Leung 1980]

Dennis, J.B., Boughton, G.A. and Leung, C.K.C. "Building Blocks for Data Flow Prototypes," *Proceedings Seventh Annual Symposium Computer Architecture,* May 1980, 1-8.

[Dennis, Fuller, Ackerman, Swan, Weng 1978]

Dennis, J.B., Fuller, S.H., Ackerman, W.B., Swan, R.J., Weng, K-S "Research Directions in Computer Architecture", Technical Report MIT/LCS/TM-114, M.I.T., September 1978.

[Evans 1982]

Evans, D.J. (Editor) *Parallel Processing Systems,* Cambridge University Press, Cambridge, 1982.

[Flynn and Hoevel 1984]

Flynn, M.J. and Hoevel, L.W. "Measures of Ideal Execution Architectures", *IBM Journal of Research and Development* 28, 4 (July 1984), 356-369.

[Gajski, Kuck, and Padua 1981]

Gajski, D.D., Kuck, D.J., and Padua, D.A. "Dependence Driven Computation", *Digest of Papers Compcon Spring 1984,* February 1981, 168-172.

[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 Data Flow", *Proceedings National Computer Conference*

[Gurd, Kirkham, and Watson 1985]

Gurd, J.R., Kirkham, C.C., and Watson, I. "The Manchester Prototype Dataflow Computer", *Communications of the ACM* 28, 1 (January 1985), 34-52.

[Gurd and Watson 1977]

Gurd, J.R., and Watson, I. "A Multilayered Data Flow Computer Architecture", *Proceedings 1977 International Conference on Parallel Processing* (August 1977), 94.

[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.

[Hwang and Briggs 1984]

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

[Keller, Lindstrom, and Patil 1979]

Keller, R.M., Lindstrom G., and Patil, S. "A Loosely-Coupled Applicative Multiprocessing System", *Proceedings National Computer Conference,*

[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 1981,* 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.

[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.

[Miklosko and Kotov 1984]

Miklosko, J. and Kotov, V.E. (Eds.) *Algorithms, Software and Hardware of Parallel Computers,* Springer-Verlag, New York, 1984.

[Misunas 1976]

Misunas, D.P. "Error detection and recovery in a data-flow computer", *Proceedings 1976 International Conference on Parallel Processing* (August1976), 117-122.

[Misunas 1978]

Misunas, D.P. "A Computer Architecture for Data-Flow Computation", Technical Report MIT/LCS/TM-100, Laboratory for Computer Science, M.I.T., March 1978.

[Misunas 1979]

Misunas, D.P. "Report on the Second Workshop on Data Flow Computer and Program Organization", Technical Report MIT/LCS/TM-136, Laboratory for Com-

puter Science, M.I.T., June 1979.

[Moore and McKay 1987]

Moore, J.B., McKay, K.N. *Modula-2 Text and Reference,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[Myers 1982]

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

[Plas, 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.

[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* 8,4 (June 1980), 11-21.

[Shimada, Hiraki, Nishida 1984]

Shimada, T., Hiraki, K., and Nishida K. "An Architecture of a Data Flow Machine and its Evaluation", *Digest of Papers Compcon Spring 84,* February 1984, 486-490.

[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.

[Srini 1985]

Srini, V.P. "A Fault-Tolerant Dataflow System", *Computer* 18 (March 1985), 54-68.

[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.

[Tiberghien 1984]

Tiberghien, J. (Ed) *New Computer Architectures,* Academic Press, Orlando, Florida, 1984.

[Todd 1982]

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

[Tokoro, Jagannathan, and Sunahara 1983]

Tokoro, M., Jagannathan, Sunahara, H. "On the Working Set Concept for Data-Flow Machines", *10th Annual International Symposium on Computer Architecture* (Stockholm, Sweden, June 13-17, 1983), ACM, NY, 1983, 90-97.

[Torsone 1985]

Torsone, C.M. "Simulation of a Data Flow Computer", Master's Thesis,RIT, Rochester,NY, April 1985.

[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, 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.

[Watson and Gurd 1982]

Watson, I. and Gurd, J.R. "A Practical Data Flow Computer", *Computer* (February 1982) 15, 2, 51-57.

[Wirth 1985] Wirth, N., *Programming in Modula-2,* Springer-Verlag, New York, 1985.

[Woo and Agrawala 1983]

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