Dataflow: Overview and simulation

Steven I. Benjamin

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
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
Date: 3/25/88

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

Approved by: Dr. Peter Lutz
Date: 3/25/88
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
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Data Flow vs. Control Flow</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Data Flow History</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>MIT Dennis Machine</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>MIT Arvind Machine</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Manchester Machine</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Utah DDM1 Machine</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Toulouse Machine</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>EDDY Machine</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>Trapesoid Rule Compilation</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>Machine Organisation</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>System Process Diagram</td>
<td>27</td>
</tr>
<tr>
<td>12</td>
<td>The config File</td>
<td>28</td>
</tr>
<tr>
<td>13</td>
<td>The commandTable File</td>
<td>30</td>
</tr>
<tr>
<td>14</td>
<td>Instruction Store Data Structure</td>
<td>31</td>
</tr>
<tr>
<td>15</td>
<td>Token Data Structure</td>
<td>32</td>
</tr>
<tr>
<td>16</td>
<td>Token Store Data Structure</td>
<td>33</td>
</tr>
<tr>
<td>17</td>
<td>Operation Packet Data Structure</td>
<td>34</td>
</tr>
<tr>
<td>18</td>
<td>Instruction Set Summary</td>
<td>35</td>
</tr>
<tr>
<td>19</td>
<td>I-structure Store Data Structure</td>
<td>36</td>
</tr>
<tr>
<td>20</td>
<td>Dataflow Graph Sequence Construct</td>
<td>41</td>
</tr>
<tr>
<td>21</td>
<td>Dataflow Graph Decision Construct</td>
<td>42</td>
</tr>
<tr>
<td>22</td>
<td>Dataflow Graph Repetition Construct</td>
<td>43</td>
</tr>
</tbody>
</table>
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
1. P = X + Y
2. Q = P / Y
3. R = X * P
4. S = R - Q
5. T = R * P
6. RSLT = S / T

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.
Figure 2. Data Flow History

MIT

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
<td>J. Dennis</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>J. Rumbaugh, D. Misunas</td>
<td>Dataflow Graphs</td>
</tr>
<tr>
<td>71</td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stanford

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>Tesler/Enea</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td></td>
<td>Single Assignment</td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Irvine

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>P. Kosinsky</td>
<td>Architecture</td>
</tr>
<tr>
<td>74</td>
<td>Arvind, K. Gostelow</td>
<td>Languages</td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>Arvind, D. Brock</td>
<td>Simulator</td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Utah

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>A. Davis</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Toulouse

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>J. M. Nicolas, J. C. Syre</td>
<td>Machine lang. architecture</td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Manchester

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>J. Gurd, I. Watson</td>
<td>Compiler</td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>D. Comte, A. Plas, N. Hildi, J. C. Syre</td>
<td>Simulator</td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Impl.

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>B. Pelois</td>
<td>processor MIMD running</td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data Flow

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td></td>
<td>Prototype Machine</td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Prototype Machine

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td></td>
<td>VAL Compiler</td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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]:
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.
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.
CE (Computing Element)

- Input Queue (IQ)
- Output Queue (OQ)
- Agenda Queue (AQ)
- Atomic Processor (AP)
- Atomic Storage (ASU)
- Switch

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

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 addresses 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.
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, structure 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:

\[
s = \frac{(f(a) + f(b))}{2} \\
x = a + h \\
\text{for } i = 1 \text{ to } n - 1 \\
\quad s = s + f(x) \\
\quad x = x + h \\
\text{end for} \\
\quad s = s \times h
\]

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

\[ \frac{1}{2} + h \]

\[ \frac{s}{2} + L \left( x < n - 1 \right) \]

\[ f + 1 \]

\[ \times h \]
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.

[Diagram of LINK actor with tokens T and T replicated and distributed to separate output tokens T and T]

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

[Diagram of OPERATOR actor with tokens T and T, function f, and output token T]

\[ f = \text{negation, } \sqrt{}, \text{abs, } +, -, /, * \]

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

[Diagram of DECIDER actor with tokens T and T, predicate p, and output token T]

\[ p = <, >, =, <=, >= \]
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.

\[
\text{while } (p(x)) \\
\quad x = f(x) \\
\text{end while}
\]

\[
\begin{align*}
\text{Initial Token} & : |x| \rightarrow |1s| \quad x = \text{data} \\
& \quad 1 = \text{iteration} \\
& \quad s = \text{sending address} \\

\text{Add new context to tag} & : |x| \rightarrow |1c| \rightarrow |1s| \quad c = \text{code block} \\

\text{Increment initiation value} & : \\
& \quad 1 : |x| \rightarrow |2c| \rightarrow |1s| \\
& \quad n : |x| \rightarrow |n+1c| \rightarrow |1s| \\

\text{Reset initiation value} & : |x| \rightarrow |1c| \rightarrow |1s| \\

\text{Remove context added by L} & : |x| \rightarrow |1s|
\end{align*}
\]
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 instruction 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 xo at selector i.

The IWRITE operation appends the value v to I-structure xo 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

1. Assemble Instruction

2. Read Data

3. Build Token Set

4. Build Operation Packet

5. Execute Operation Packet

- Token Set Queue
- Operation Packet Queue
- Token Queue
- I-structure Store

Input

Output
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 mnemonic, 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

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

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

<table>
<thead>
<tr>
<th>Activity Name (Tag)</th>
<th>LIFO queue</th>
<th>Destination</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>data na</td>
<td>i s</td>
<td>i s/c</td>
<td>i s/c</td>
</tr>
</tbody>
</table>

data 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

<table>
<thead>
<tr>
<th></th>
<th>max instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ec cc</td>
<td>ec cc</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>V</td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tc</th>
<th>p..p</th>
<th>Token Set Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>na</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Token Set Activity Name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Token Set Activity Name</td>
</tr>
</tbody>
</table>

- **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

<table>
<thead>
<tr>
<th>d</th>
<th>p</th>
<th>d</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>q</th>
<th>q</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>q</th>
<th>q</th>
<th>q</th>
</tr>
</thead>
</table>

d data
p presence bit
q deferred read request
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.

Sequence

```
function
```

```
function
```

Decision

```
predicate
```

```
predicate
```

Repetition

```
predicate
```

```
function
```

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.

\[
\begin{align*}
x &= y + z \\
v &= u - m \\
w &= x/v
\end{align*}
\]

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 $f(y)$

\[
\text{If } x = y \text{ then } f(y) \text{ else } g(y)
\]
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

(1 is the code block number)
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

mainline
\begin{verbatim}
x
y
A
x, y, A1-addr
\end{verbatim}

\begin{verbatim}
A1
z
\end{verbatim}

\begin{verbatim}
x y A1-addr
PBEG
x y
+
A1-addr
PEND
z
\end{verbatim}
4.1.5. I-structures

The IREAD, IWRITE, and IFREE instructions read, write, and reallocate 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

(1 is the code block number)
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**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>comment; characters from # to end of line are skipped</td>
</tr>
<tr>
<td>a</td>
<td>add subprogram context; invoke subprogram</td>
</tr>
<tr>
<td>al</td>
<td>remove subprogram context; distribute result tokens</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value</td>
</tr>
<tr>
<td>add</td>
<td>addition</td>
</tr>
<tr>
<td>and</td>
<td>boolean and</td>
</tr>
<tr>
<td>const</td>
<td>store instruction constants</td>
</tr>
<tr>
<td>d</td>
<td>increment loop iteration counter</td>
</tr>
<tr>
<td>debug</td>
<td>output token values at particular point in graph</td>
</tr>
<tr>
<td>d1</td>
<td>reset loop iteration counter to 1</td>
</tr>
<tr>
<td>div</td>
<td>division</td>
</tr>
<tr>
<td>eq</td>
<td>equal</td>
</tr>
<tr>
<td>fgate</td>
<td>propagate token if control signal is false</td>
</tr>
<tr>
<td>ge</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>gt</td>
<td>greater than</td>
</tr>
<tr>
<td>halt</td>
<td>halt program</td>
</tr>
<tr>
<td>ifree</td>
<td>reinitialize i-structure cell</td>
</tr>
<tr>
<td>iread</td>
<td>read i-structure cell</td>
</tr>
<tr>
<td>iwrite</td>
<td>write i-structure cell</td>
</tr>
<tr>
<td>l</td>
<td>add loop context</td>
</tr>
<tr>
<td>l1</td>
<td>remove loop context</td>
</tr>
<tr>
<td>le</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>link</td>
<td>duplicate token</td>
</tr>
<tr>
<td>lt</td>
<td>less than</td>
</tr>
<tr>
<td>mul</td>
<td>multiplication</td>
</tr>
<tr>
<td>ne</td>
<td>not equal</td>
</tr>
<tr>
<td>neg</td>
<td>negate value</td>
</tr>
<tr>
<td>not</td>
<td>boolean not</td>
</tr>
<tr>
<td>or</td>
<td>boolean or</td>
</tr>
<tr>
<td>output</td>
<td>output token value(s)</td>
</tr>
<tr>
<td>pbeg</td>
<td>initiate procedure</td>
</tr>
<tr>
<td>pend</td>
<td>terminate procedure</td>
</tr>
<tr>
<td>sqrt</td>
<td>square root</td>
</tr>
<tr>
<td>sub</td>
<td>subtraction</td>
</tr>
<tr>
<td>switch</td>
<td>choose token path depending on control signal</td>
</tr>
<tr>
<td>tgate</td>
<td>propagate token if control signal is true</td>
</tr>
<tr>
<td>trace</td>
<td>trace simulator execution</td>
</tr>
</tbody>
</table>
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{0}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
ddebug 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 fl .. 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.
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**  
\[ \text{addr} \rightarrow \text{dest} \]
- addr - address in the graph
- dest - result token destination address and port

Find absolute value of port 1.

**ADD**  
\[ \text{addr} \rightarrow \text{dest} \]
- addr - address in the graph
- dest - result token destination address and port

Add port 1 to port 2.

**DIV**  
\[ \text{addr} \rightarrow \text{dest} \]
- addr - address in the graph
- dest - result token destination address and port

Divide port 1 by 2.

**MUL**  
\[ \text{addr} \rightarrow \text{dest} \]
- addr - address in the graph
- dest - result token destination address and port

Multiply port 1 by port 2.

**NEG**  
\[ \text{addr} \rightarrow \text{dest} \]
- addr - address in the graph
- dest - result token destination address and port

Multiply port 1 by -1.

**SQRT**  
\[ \text{addr} \rightarrow \text{dest} \]
- addr - address in the graph
- dest - result token destination address and port

Find square root of port 1.

**SUB**  
\[ \text{addr} \rightarrow \text{dest} \]
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

FGATE addr  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).
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.

<table>
<thead>
<tr>
<th>Input file</th>
<th>Output file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>addr/port</td>
</tr>
<tr>
<td>2</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Example 2.

<table>
<thead>
<tr>
<th>Input file</th>
<th>Output file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>addr/port</td>
</tr>
<tr>
<td>-2</td>
<td>1 1</td>
</tr>
</tbody>
</table>
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

```plaintext
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
```
# Sample Program One
# Nested loop

# I loop code block

<table>
<thead>
<tr>
<th>link</th>
<th>gate</th>
<th>l</th>
<th>d</th>
<th>link</th>
<th>gate</th>
<th>l</th>
<th>d</th>
<th>link</th>
<th>gate</th>
<th>l</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td></td>
<td>5</td>
<td>1</td>
<td>4</td>
<td></td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>8</td>
<td>1</td>
<td>4</td>
<td></td>
<td>9</td>
<td>1</td>
<td>6</td>
<td></td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td>9</td>
<td>5</td>
<td></td>
<td>8</td>
<td>1</td>
<td>1</td>
<td></td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>7</td>
<td>8</td>
<td>6</td>
<td></td>
<td>9</td>
<td>1</td>
<td>7</td>
<td></td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>12</td>
<td>1</td>
<td>9</td>
<td></td>
<td>14</td>
<td>2</td>
<td>10</td>
<td></td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>12</td>
<td>1</td>
<td>14</td>
<td></td>
<td>1</td>
<td>7</td>
<td>10</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>13</td>
<td>1</td>
<td>14</td>
<td></td>
<td>1</td>
<td>10</td>
<td>17</td>
<td></td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>13</td>
<td>1</td>
<td>18</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>13</td>
<td>1</td>
<td>16</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td></td>
<td>13</td>
<td>1</td>
<td>17</td>
<td></td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>14</td>
<td>1</td>
</tr>
</tbody>
</table>

# I to J loop connection

<table>
<thead>
<tr>
<th>link</th>
<th>gate</th>
<th>l</th>
<th>d</th>
<th>link</th>
<th>gate</th>
<th>l</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>19</td>
<td>2</td>
<td>2</td>
<td>20</td>
<td>19</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>21</td>
<td>1</td>
<td>20</td>
<td></td>
<td>22</td>
<td>1</td>
</tr>
</tbody>
</table>

# J loop code block

<table>
<thead>
<tr>
<th>l</th>
<th>d</th>
<th>link</th>
<th>gate</th>
<th>l</th>
<th>d</th>
<th>link</th>
<th>gate</th>
<th>l</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>2</td>
<td>25</td>
<td>1</td>
<td>22</td>
<td>2</td>
<td>26</td>
<td>1</td>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>25</td>
<td>1</td>
<td>25</td>
<td>2</td>
<td>31</td>
<td>2</td>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>29</td>
<td>1</td>
<td>28</td>
<td>2</td>
<td>30</td>
<td>1</td>
<td>29</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>32</td>
<td>1</td>
<td>31</td>
<td>2</td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>2</td>
<td>37</td>
<td>2</td>
<td>34</td>
<td>2</td>
<td>37</td>
<td>2</td>
<td>35</td>
<td>2</td>
</tr>
</tbody>
</table>

# J to K loop connection

<table>
<thead>
<tr>
<th>link</th>
<th>gate</th>
<th>l</th>
<th>d</th>
<th>link</th>
<th>gate</th>
<th>l</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>36</td>
<td>36</td>
<td>2</td>
<td>37</td>
<td>36</td>
<td>38</td>
<td>1</td>
</tr>
</tbody>
</table>

# 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
tle  47  44 1
link  48  49 1  50 1
output  49  1  k=
add  50  40 1

# instruction constants

cost  3  2 1
cost  1  3 1
cost  1  17 2
cost  3  19 1
cost  1  20 1
cost  1  34 2
cost  3  36 1
cost  1  37 1
cost  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
j= 3.000000
k= 1.000000
4.5.2. **Recursive Factorial Example**

```plaintext
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
```
# Sample Program Two
#  Recursive factorial

# main program

a  1  1  2  4
al 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
al 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=  362880.00000
4.5.3. *Fibonacci Series Example*

```plaintext
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
```
# Sample Three
# Fibonacci 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
tgate  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
INPUT FILE

<table>
<thead>
<tr>
<th>value</th>
<th>node/port</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1</td>
</tr>
<tr>
<td>9</td>
<td>2 1</td>
</tr>
</tbody>
</table>

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
Sample Program Four
# Trapesoid 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
al  15  1  18  1
al  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
al  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. L-Structure Example

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

<table>
<thead>
<tr>
<th>value</th>
<th>node/port</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1</td>
</tr>
<tr>
<td>10</td>
<td>2 1</td>
</tr>
</tbody>
</table>

**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
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
**# Sample Program 6**

**# Laplace Transform**

**# Set up Istructure cell indices**

<table>
<thead>
<tr>
<th>iwrite</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0</td>
</tr>
<tr>
<td>4</td>
<td>0 0</td>
</tr>
<tr>
<td>5</td>
<td>0 0</td>
</tr>
<tr>
<td>6</td>
<td>0 0</td>
</tr>
<tr>
<td>7</td>
<td>0 0</td>
</tr>
<tr>
<td>8</td>
<td>0 0</td>
</tr>
<tr>
<td>9</td>
<td>0 0</td>
</tr>
<tr>
<td>10</td>
<td>0 0</td>
</tr>
<tr>
<td>11</td>
<td>0 0</td>
</tr>
<tr>
<td>12</td>
<td>0 0</td>
</tr>
<tr>
<td>13</td>
<td>0 0</td>
</tr>
<tr>
<td>14</td>
<td>0 0</td>
</tr>
<tr>
<td>15</td>
<td>0 0</td>
</tr>
<tr>
<td>16</td>
<td>0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>const</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 1</td>
</tr>
<tr>
<td>0</td>
<td>1 2</td>
</tr>
<tr>
<td>0</td>
<td>2 1</td>
</tr>
<tr>
<td>1</td>
<td>2 2</td>
</tr>
<tr>
<td>0</td>
<td>3 1</td>
</tr>
<tr>
<td>2</td>
<td>3 2</td>
</tr>
<tr>
<td>0</td>
<td>4 1</td>
</tr>
<tr>
<td>3</td>
<td>4 2</td>
</tr>
<tr>
<td>1</td>
<td>5 1</td>
</tr>
<tr>
<td>0</td>
<td>5 2</td>
</tr>
<tr>
<td>1</td>
<td>6 1</td>
</tr>
<tr>
<td>1</td>
<td>6 2</td>
</tr>
<tr>
<td>1</td>
<td>7 1</td>
</tr>
<tr>
<td>2</td>
<td>7 2</td>
</tr>
<tr>
<td>1</td>
<td>8 1</td>
</tr>
<tr>
<td>3</td>
<td>8 2</td>
</tr>
<tr>
<td>2</td>
<td>9 1</td>
</tr>
<tr>
<td>0</td>
<td>9 2</td>
</tr>
<tr>
<td>2</td>
<td>10 1</td>
</tr>
<tr>
<td>1</td>
<td>10 2</td>
</tr>
<tr>
<td>2</td>
<td>11 1</td>
</tr>
<tr>
<td>2</td>
<td>11 2</td>
</tr>
<tr>
<td>2</td>
<td>12 1</td>
</tr>
<tr>
<td>3</td>
<td>12 2</td>
</tr>
<tr>
<td>3</td>
<td>13 1</td>
</tr>
<tr>
<td>0</td>
<td>13 2</td>
</tr>
<tr>
<td>3</td>
<td>14 1</td>
</tr>
<tr>
<td>1</td>
<td>14 2</td>
</tr>
<tr>
<td>3</td>
<td>15 1</td>
</tr>
<tr>
<td>2</td>
<td>15 2</td>
</tr>
<tr>
<td>3</td>
<td>16 1</td>
</tr>
</tbody>
</table>
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
al  76  1  75  2
a  77  1  78  79
al  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
dl  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
link 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
### Instruction Constants

<table>
<thead>
<tr>
<th>const</th>
<th>2</th>
<th>60 1</th>
<th># n</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>2</td>
<td>73 1</td>
<td># n</td>
</tr>
<tr>
<td>const</td>
<td>3</td>
<td>44 1</td>
<td># m</td>
</tr>
<tr>
<td>const</td>
<td>0</td>
<td>334 2</td>
<td># icell base address</td>
</tr>
<tr>
<td>const</td>
<td>0</td>
<td>335 2</td>
<td># icell base address</td>
</tr>
<tr>
<td>const</td>
<td>0</td>
<td>336 2</td>
<td># icell base address</td>
</tr>
<tr>
<td>const</td>
<td>0</td>
<td>337 2</td>
<td># icell base address</td>
</tr>
<tr>
<td>const</td>
<td>0</td>
<td>338 2</td>
<td># icell base address</td>
</tr>
<tr>
<td>const</td>
<td>0</td>
<td>346 2</td>
<td># icell base address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>const</th>
<th>1</th>
<th>45 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>1</td>
<td>58 2</td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>61 1</td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>73 2</td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>74 2</td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>81 1</td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>317 2</td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>330 2</td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>331 2</td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>332 2</td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>333 2</td>
</tr>
<tr>
<td>const</td>
<td>2</td>
<td>347 2</td>
</tr>
<tr>
<td>const</td>
<td>8</td>
<td>349 2</td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>354 1</td>
</tr>
</tbody>
</table>
### INPUT FILE

<table>
<thead>
<tr>
<th>value</th>
<th>node/port</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 3</td>
</tr>
<tr>
<td>7</td>
<td>2 3</td>
</tr>
<tr>
<td>3</td>
<td>3 3</td>
</tr>
<tr>
<td>2</td>
<td>4 3</td>
</tr>
<tr>
<td>6</td>
<td>5 3</td>
</tr>
<tr>
<td>6</td>
<td>6 3</td>
</tr>
<tr>
<td>1</td>
<td>7 3</td>
</tr>
<tr>
<td>5</td>
<td>8 3</td>
</tr>
<tr>
<td>0</td>
<td>9 3</td>
</tr>
<tr>
<td>1</td>
<td>10 3</td>
</tr>
<tr>
<td>1</td>
<td>11 3</td>
</tr>
<tr>
<td>8</td>
<td>12 3</td>
</tr>
<tr>
<td>4</td>
<td>13 3</td>
</tr>
<tr>
<td>5</td>
<td>14 3</td>
</tr>
<tr>
<td>7</td>
<td>15 3</td>
</tr>
<tr>
<td>9</td>
<td>16 3</td>
</tr>
<tr>
<td>1</td>
<td>43 1</td>
</tr>
</tbody>
</table>
### OUTPUT FILE

<table>
<thead>
<tr>
<th>ijv</th>
<th>1.000000</th>
<th>1.000000</th>
<th>4.875000</th>
</tr>
</thead>
<tbody>
<tr>
<td>ijv</td>
<td>1.000000</td>
<td>2.000000</td>
<td>2.234375</td>
</tr>
<tr>
<td>ijv</td>
<td>2.000000</td>
<td>1.000000</td>
<td>1.859375</td>
</tr>
<tr>
<td>ijv</td>
<td>2.000000</td>
<td>2.000000</td>
<td>2.886719</td>
</tr>
<tr>
<td>ijv</td>
<td>1.000000</td>
<td>1.000000</td>
<td>4.574219</td>
</tr>
<tr>
<td>ijv</td>
<td>1.000000</td>
<td>2.000000</td>
<td>3.049805</td>
</tr>
<tr>
<td>ijv</td>
<td>2.000000</td>
<td>1.000000</td>
<td>2.487305</td>
</tr>
<tr>
<td>ijv</td>
<td>2.000000</td>
<td>2.000000</td>
<td>4.010498</td>
</tr>
<tr>
<td>ijv</td>
<td>1.000000</td>
<td>1.000000</td>
<td>4.604248</td>
</tr>
<tr>
<td>ijv</td>
<td>1.000000</td>
<td>2.000000</td>
<td>3.601746</td>
</tr>
<tr>
<td>ijv</td>
<td>2.000000</td>
<td>1.000000</td>
<td>2.945496</td>
</tr>
<tr>
<td>ijv</td>
<td>2.000000</td>
<td>2.000000</td>
<td>4.698654</td>
</tr>
</tbody>
</table>

#### 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, Gos-telow 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]

[Arvind, Dertouzos, and Iannucci 1983]

[Arvind and Gostelow 1977]

[Arvind and Gostelow 1982]

[Arvind and Kathail 1981]

[Arvind and Thomas 1980]
Arvind and Thomas, R.E. "I-Structures: An Efficient Data Type for Functional

[Backus 1978]


[Carlson and Hwang 1985]


[Clark 1973]


[Comte, Hifdi and Syre 1980]


[Davis 1978]


[Davis 1979]

[Davis and Keller 1982]


[Davis and Lowder 1981]


[De Marco 1978]


[Dennis 1975]


[Dennis 1977]


[Dennis 1980]


[Dennis 1984]


[Dennis, Boughton, and Leung 1980]

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


[Evans 1982]


[Flynn and Hoevel 1984]


[Gajski, Kuck, and Padua 1981]


[Gajski, Padua, Kuck and Kuhn 1982]


[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 and Watson 1977]


[Ho and Irani 1983]


[Hogenauer, Newbold, and Inn 1982]


[Hwang and Briggs 1984]


[Keller, Lindstrom, and Patil 1979]


[Keller and Yen 1981]

156-161.

[Leler 1983]


[Lerner 1984]


[Litvin 1983]


[Miklosko and Kotov 1984]


[Misunas 1976]


[Misunas 1978]


[Misunas 1979]


[Moore and McKay 1987]


[Myers 1982]


[Schwartz 1980]


[Sharp 1980]


[Shimada, Hiraki, Nishida 1984]


[Srini 1981]


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

[Syre, Comte, and Hifdi 1977]

[Tiberghien 1984]

[Todd 1982]

[Tokoro, Jagannathan, and Sunahara 1983]

[Torsone 1985]

[Treleaven 1979]
[Treleaven 1980]


[Treleaven, Brownbridge, and Hopkins 1982]


[Watson and Gurd 1979]


[Watson and Gurd 1982]


[Woo and Agrawala 1983]