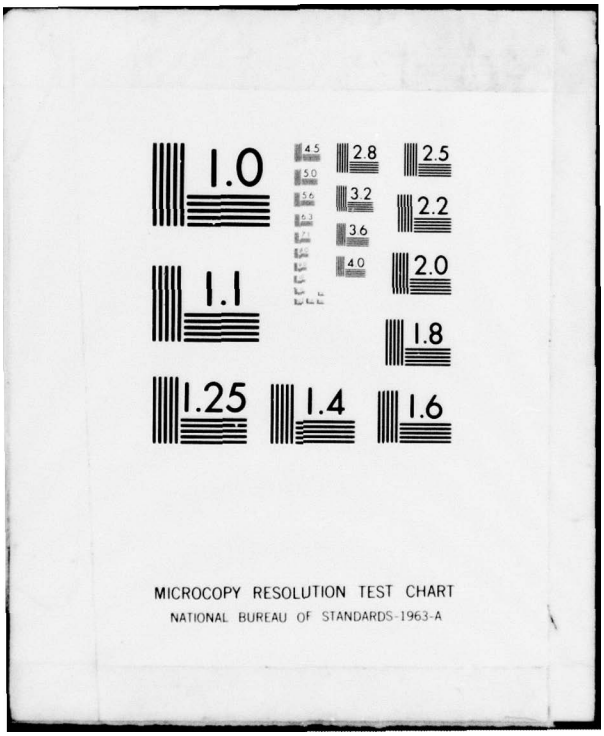
AD-A080 174 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
DATA FLOW TECHNIQUES IN SINGLE-USER MULTIPROCESSOR SYSTEMS.(U)
DEC 79 B P BOESCH
UNCLASSIFIED AFIT/GE/EE/79-7 NL

1 OF 2
AD
A080174


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LEVEL II

UNITED STATES AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/EE/79-7

DATA FLOW TECHNIQUES IN SINGLE-USER MULTIPROCESSOR SYSTEMS

THESIS

AFIT/GE/EE/79-7 Brian P. Boesch
Capt USAF

Approved for public release; distribution unlimited

(14) AFIT/GE/EE/79-7

(6) DATA FLOW TECHNIQUES IN SINGLE-USER MULTIPROCESSOR SYSTEMS.

(9) Master's THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air Training Command

in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

by

(10) Brian P. Boesch, B.S.

CAPT USAF

Graduate Electrical Engineering

(12) 103

(11) December 1979

Preface

In this report, I attempted to determine if it was possible to design, evaluate, and test a data flow processor within the course of one Masters Thesis. To this end, I was only partially successful. There is no quantitative analysis of the effectiveness of data flow. Qualitatively, the test cases indicate that dataflow processors constructed from conventional microprocessors may not be practical. The conclusions section of this report describes several ideas that may lead to practical data flow microprocessor systems. Hopefully, other students and researchers can use the tools developed in this effort to perform quantitative estimates of the utility of data flow processors.

I would like to express my gratitude to the people who made my work possible. First, Kathy, my wife, who was foolish enough to marry me while I was working on this thesis, and caring enough not to divorce me before I finished. Secondly, Dr. Lamont, my advisor, whose vigilence rescued me from several pitfalls. Finally, the technicians in the lab who were always very responsive in manufacturing any equipment I needed to complete my work.

Brian P. Boesch

Contents

List of Figures

List of Tables

Abstract

Problems associated with performing computation tasks in single user microprocessor environments are discussed. Data flow methods are investigated as a means of applying multiprocessors to this environment. An event driven simulation of a variable architecture data flow processor is developed, using UCSD Pascal(TM). A trial data flow multiprocessor using 8080 based microprocessors is developed and tested. Using the simulator and information gathered from the trial multiprocessor, several situations typical to single user systems were tested. Maximal levels of parallel processing are presented for a variety of situations.

DATA FLOW TECHNIQUES IN SINGLE USER MULTIPROCESSOR SYSTEMS

I. Introduction

Large scale computers have traditionally been considered useful for many computing tasks. Today however, the cost of large systems is becoming increasingly prohibitive when compared to the falling cost of microcomputers. In addition, as language development progressed, it became more and more practical to use microcomputers to perform computational tasks, making it desirable to shift from large timeshared machines to small processor systems. Unfortunately, microcomputers have limited processing speed (Ref 7). This severely limits the types of tasks which can be solved using such systems. Tasks such as interactive procedures and time critical calculations often can not be performed by microcomputers within time constraints.

To solve highly computational tasks using microcomputers, it would be desirable to have a practical way to increase the power of a microcomputer system incrementally by adding processors. This has been done in many large multiuser systems to increase computational power, and there have been several proposed large multiprocessing systems constructed using microprocessors (Ref 13, 17, 18). But multiprocessing has rarely been used with small microprocessor systems.

The problem of multiprocessing in the typical small microcomputer

system is inherently more complex than the problem of multiprocessing in large systems. This is due to the way in which the systems are employed rather than the hardware used to implement them. It is analogous to putting more than one worker in a manufacturing plant. Both workers can work in parallel so long as there are two separate jobs to be done. If the job function cannot be divided between the workers, then one will stand idle while the the other works. In a computer system the same relationship holds. If the job can be broken into independent tasks, then two or more processors can be employed effectively. If not, then only one processor can be efficiently used.

In large systems, there are generally several users simultaneously using the system. Each of the users' work is effectively independent of the others. Therefore, it is a relatively simple matter to schedule these independent tasks in a multiprocessor. Microprocessor systems, on the other hand, typically are used in small dedicated applications where a computer serves one user. To effectively multiprocess, the computer system must divide the user's work between processors. That is often difficult because the typical user considers his work to be comprised of a sequence of logical tasks done one at a time, not as a set of actions that may be done in parallel.

Developing effective multiprocessing on a single user microprocessor, therefore, reduces to partitioning the logical tasks requested by the operator into a set of noninterfering (Ref 8:39) subtasks.

One method of partitioning is to divide the work into functional areas. For example, one processor operates the printer, one the keyboard, another the screen, and finally one processor is the

"computer" actually performing the intended task. This approach works nicely in a system that is input/output bound. If, however, the system is compute bound, assigning peripheral functions to other processors will simply complicate the system with little or no improvement.

Another method is to allow the user to specify independent program areas that can be executed simultaneously. This approach has merit in that it is relatively easily implemented on most computer systems and it will often provide significant parallelism if the programmer takes full advantage of the machine. The user, however, will often continue with obvious inefficiencies in a program simply because he has "always done it that way". Determining if a section of a user task is independent of another section is a complicated task; systems programmers whose primary function is to program multiple independent processors often err in determining independence. Therefore, to expect a general user to partition his work into independent sections is impractical!

The remaining method of decreasing the primary task execution time is by allowing the processor to determine those sections of the users task that can be executed in parallel. One way of representing this is called data flow processing (Ref 9). The task is considered as a directed graph where the arrows represent movement of data and the nodes represent data transforms. Each transform is completely independent of all others because it depends solely on the data on its inbound arcs. When done with its computation, the node places the result value on the outbound arcs for transfer to other nodes. This independence allows the individual transforms to be executed arbitrarily as soon as all of their inputs are present. Further,

transforms can also be allocated arbitrarily to processors of a multiprocessor system.

Until recently, data flow has been considered primarily as a mathematical tool to expand the processing capability of large processing systems. Compilers, when performing code optimization and register allocation, often generate graphical representations of the interdependence of data within a program to take advantage of inherent parallelism in their computers.

Currently, there is much interest in developing machines that can directly execute a data flow representation of a problem. Such a machine would avoid the need to approximate data flow execution on conventional computers. There have been several designs of large data flow processors proposed. For example, Rumbaugh has proposed one architecture that will execute data flow (Ref 21). Lawrence Livermore Laboratories has done studies on the practicality of data flow computer systems that would theoretically be capable of many times the throughput of large conventional systems (Ref 1).

Though most of the data flow research seems directed at the large processor, it would appear that data flow concepts could be applied to the single user scenario described previously. In such a system, the intent would be to use parallelism to eliminate the higher cost of fast processors.

Statement of Problem

The purpose of this effort is to investigate data flow techniques as they apply to the single user multiprocessor system. The goals are to assess the practicality of developing a single user data flow

processor using current technology. Factors bearing on this goal are cost of the system, flexibility, and ease of use.

Scope

Toward these goals, the specific scope of this investigation shall be:

1. to investigate current data flow technology and to formulate ways in which it may be applied to the single user computer system;
2. to develop a simulation of a data flow processor to allow evaluation of different architectures and processing speeds;
3. to develop an executing data flow multiprocessor, using existing technology (Because of the widespread availability of microprocessors, the processor shall be developed using an 8080 based microcomputer development system);
4. to formulate, using information gathered from the above tasks, an assessment of the practicality of microcomputer based data flow processors.

Assumptions

The performance of computer systems is difficult to measure, because it is often more dependent on the user environment than on the processor design (Ref 14:1-5). To make any statements about the relative worth of one option over another, it is first necessary to define the environment in which the system will operate. In this investigation, the following shall be defined as the operating environment of the system:

1. Single user- To date, multiprocessor systems have been used

in a variety of large multi-user and timeshared systems (Ref 5: chapters 22, 36, and 37). Because there are several distinct tasks happening concurrently in such systems, it is possible to isolate independent tasks for separate processors to perform simply by having processors always execute on different users' programs. The intent of this investigation is to determine ways to separate seemingly indivisible tasks into independent subtasks; therefore, only one major task will be allowed to directly operate in the system at any given time.

2. Small- The size shall be limited to a small system. The motivation for this investigation is to allow incremental expansion of microprocessor systems to minimize cost. To propose the use of large computer systems would defeat this intent. There are no firm guidelines defining small and large computer systems! For want of a better criterion, any relatively low cost computer affordable by a small business or laboratory shall be considered a small system.
3. Laboratory environment- This requirement is primarily to define the type of work to be performed. The primary function of the computer is numerical anlyisis not database manipulation or text editing. The emphasis on numerical processing is meaningful in the context of a laboratory where computers are used as "number crunchers" to perform repeated calculations on the numerical results of experiments.
4. Non real-time - Multiprocessing can often be used to advantage in realtime systems, but such systems are

inherently different enough from the thrust of this investigation to exclude them from discussion. The intent of this statement is to prohibit highly time dependent functions such as radar scheduling or process control. In such systems, the tasks are usually very time critical. Often, there is a deadline time by which a task must be completed. Rescheduling of tasks, or subtasks, is made extremely difficult by such deadlines. For example, in a nonrealtime system, execution time of a series of tasks may be decreased by delaying one large task. In real-time environments, such delay of a critical task may be unacceptable even though the entire workload could complete earlier. Such constraints do not exist in the normal single user system. The user may have a desire to have his work done as soon as possible but there are no specfic task deadlines.

5. Reconfigurable- The number of processors available at any given time should be hidden from the user. To the user, a three processor system should simply be "faster" than a two processor system. The intent here is not so much to allow rapid rearrangement of the system as to require program independence from any specific hardware parameters. For example, a program written for a two processor system should run, without modification, on both a one processor and a ten processor system.

Standards

The criterion on which the performance of the trial system shall

be judged is cost per unit of performance. The intent is to investigate data flow as a cost effective means for implementing multi-microprocessors. If data flow multi- microprocessors cost more per unit of performance than other techniques, they are not cost effective. At the same time, however, it should be remembered that the cost of hardware is rapidly decreasing with time. Therefore, though this investigation will make an assessment on the relative cost of data flow processors using contemporary technology, results should be reassessed in light of any significant hardware cost changes.

Order of Presentation

This report will discuss current usage of data flow processing with particular emphasis on multiprocessors, then describe development of a variable architecture data flow simulator and hardware implementation of a data flow processor. Specifically, Section II will define data flow and discuss its advantages and problems. Section III will describe the design, development and initial test of a data flow simulator. Section IV describes the development and implementation of a data flow multiprocessor. Finally, Sections V and VI show results obtained and conclusions drawn from this investigation.

II. Data Flow, an Overview

Data flow is a rather simple concept that has been in use since the early days of electronic computing. As early as 1940, John von Neumann had formulated the concepts of "neuron nets". In the past decade, Carl Adam Petri developed the concept of "Petri Nets" (Ref 19) which show critical timing relations in both hardware and software systems. A Petri Net defines a graph of dependent events, but does not show either control of the events or data movement. Program graphs, another method of expressing parallel events, allows data and control separatly to flow through the graph(Ref 15). Data flow(Ref 9), a further extension, allows only data movement; control can be expressed in terms of the presence or absence of data. The following paragraphs will describe the use of data flow and problems associated with them.

What is a data flow program?

A data flow program is a directed graph (digraph). Each node is a transform. Each arc represents the flow of data from one node to another. A data flow system may be defined either to allow only one value to occupy an arc, or it may be defined to allow values to queue on the inputs to the destination nodes. The transforms at nodes can be any form of data manipulation from as simple as an integer add to as complicated as a fast fourier transform. If only one value may occupy an arc, then a node must have all of its inputs present and all of its outputs empty to execute. If values may queue, the only requirement for execution is that the node cannot start processing information until all of its inputs are present. (Because the intent of this report

is to maxize parallelism, the first definition will not be considered further as it arbitrarily reduces the number of concurrently executing functions). When executed, a node consumes the data values on its input arcs, performs its transform, and then passes the result along some or all of its output arcs. This concept may be better described through a simple example of a data flow program.

Example Data Flow Program. Consider a data flow processor with the set of transforms (nodes) shown in Table I. Each node shown in the table takes the indicated number of inputs, and produces the result by applying its arithmetic operator to those inputs. Now consider the following Pascal assignment statement:

```
A := (B / C)  + (C * D)
```

The corresponding data flow program is shown in Figure 1(a). Note that the multiply and the divide are dependent only on values generated outside of this expression. Because they do not directly or indirectly depend on each other for their results, those two operations can be done in parallel.

If in this example, the independent variables B, C, and D were given the values 9, 11 and 13 respectively, the sequence of events in a data flow processor would be as follows. Figure 1(b) shows the input values waiting on the input arcs of the multiply and divide nodes. If the multiply node executed first, it would read its input arcs, perform the multiply, and send the product down its output arc to the addition node. The value would then wait on the right input of the addition node (Figure 1(c)). The add node cannot begin yet because it has only one of its two inputs. Next, suppose that the divide node was

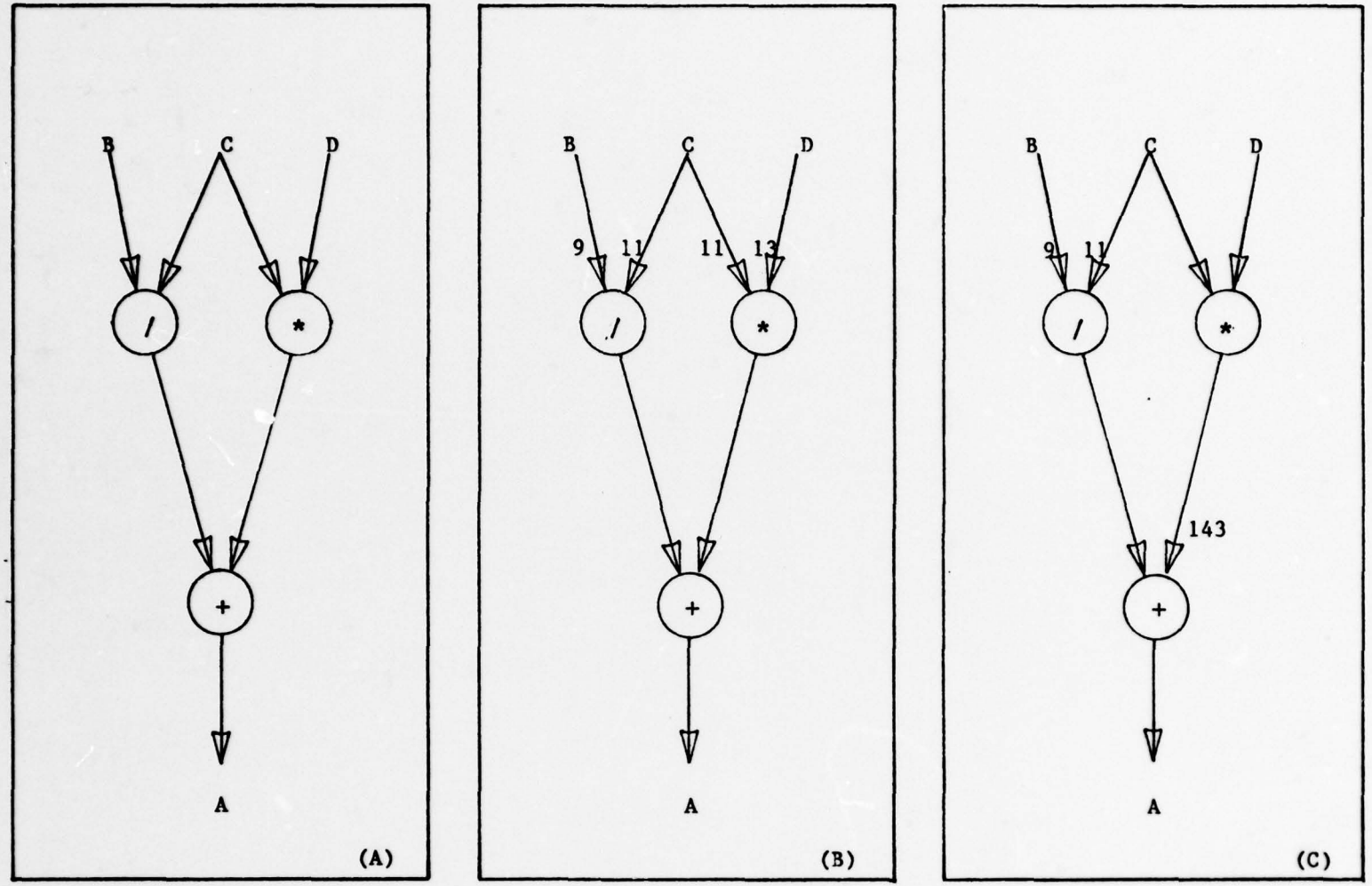


Fig 1. Example Data Flow Program

executed. It would remove the values from its input arcs and put the integer quotient on the output arc. The value would pass to the left input of the addition node and wait there for the addition node to begin execution. The add node would then execute, read its input arcs, produce the sum, and place it on its output arc.

In this simplified example of a data flow program operations were taken one at a time to allow clear explanation of the steps involved. In actuality any node which has all of its inputs available can begin at any time. Therefore, the divide and the multiply could operate simultaneously. Also, after the multiply and divide finish more inputs can be introduced at the top of the graph. These inputs will move through the graph as nodes become free to process them, thereby setting up a kind of pipeline.

Existing Data Flow Techniques

Data flow processors can be designed along two drastically different approaches. In one approach, each node has its own processor. In the other, all nodes draw on a central pool of processors, and return the processor to the pool when completed. The first approach may be practical for solving extremely large problems in a minimum of time, but it does not meet this investigation's requirement of reconfigurability. The second approach shall be used here.

The multiprocessor pooled approach does meet the requirement of reconfigurability because the number of processors does not affect the number of nodes possible, only the maximum number that may be executing at one time. For example, if a multiprocessor has nine processors and

there are twelve tasks available to be computed simultaneously, then all nine processors will begin immediately. The remaining tasks will begin when a processor is free. On the other hand, in a twenty processor computer with the same twelve tasks available for simultaneous processing, all twelve will begin simultaneously while eight of the processors wait for task. In each case, data flow allows the maximum possible parallelism consistant with the hardware and the program.

Clearly, in a physical implementation of a data flow processor using the pooled processor approach, the operation does not necessarily take place as soon as all inputs are present. Rather, it takes place as soon as a processor is available after all inputs are present. There may be some precedence of operators in a given implementation, or nodes may be queued as they are available for execution. Independence of nodes is critical, because a node could be executed after a node enabled later than it. For example, two nodes could be enabled in close succession, but when assigned to processors the second to be enabled might be the first to begin execution. Without independence, execution out of order could cause problems.

Problems with Data Flow Techniques

Though data flow appears, on the surface, to be the salvation for the single user parallel processing system, there are problems. Data flow programs may be indeterminate, data structures are poorly handled by data flow processors, and data flow programs are difficult to understand.

Determinacy

Definition of determinacy in compter programs.

A determinate computer program will always produce the same output given the same input. External events not changing the program's inputs will have no effect on its outputs.

Nondeterminism in data flow. One may think that independence of operators in data flow totally protects the user from nondeterminism in programs. It does not! Using the constructs of data flow, namely arcs and nodes, it is very possible to develop programs that exhibit nondeterministic behavior.

Examples of nondeterminism. The most obvious case of nondeterminism can be seen in Figure 2. This example shall use the same operators as the previous example (Table I). Assume that initially the three inputs: A, B, and C have the values 4, 5, and 7 respectively. The inputs to the conditional cause the true output to be taken. An input is left "hanging" on the input arc of node 2. That input cannot be used by its node because the node did not receive its second input from the conditional. This hanging input will simply wait on the input arc until at some later time the false conditional output is exercised. At that time, the hanging input will be confused for the current input. This case of indeterminacy can be eliminated by good programming practice. All inputs to any nodes between a conditional and the corresponding merge node must come through the conditional. If some inputs needed by the true case are not needed by the false case then they should be disposed of after the conditional.

Another case of indeterminacy can be shown in Figure 3(a). In

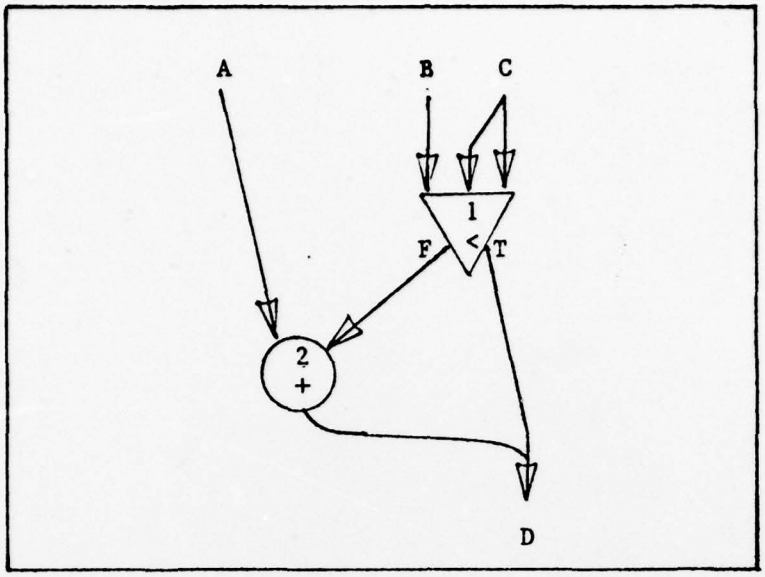


Fig 2. Non-determinism Example One

TABLE I

Set of Transforms for Example Dataflow Processor

Transform	Inputs	Meaning
+	2	Add the values of the two inputs, producing their sum for output.
-	2	Subtract right input value from left input value, giving their difference for output.
*	2	Multiply the values of the two inputs, producing their product for output.
/	2	Divide the right input value into the left input value, giving their quotient for output.
<	n	Compare inputs 1 and 2 then pass inputs 3..n to outputs 1,3,5...n-1 for test=true, outputs 2,4,6..n for false

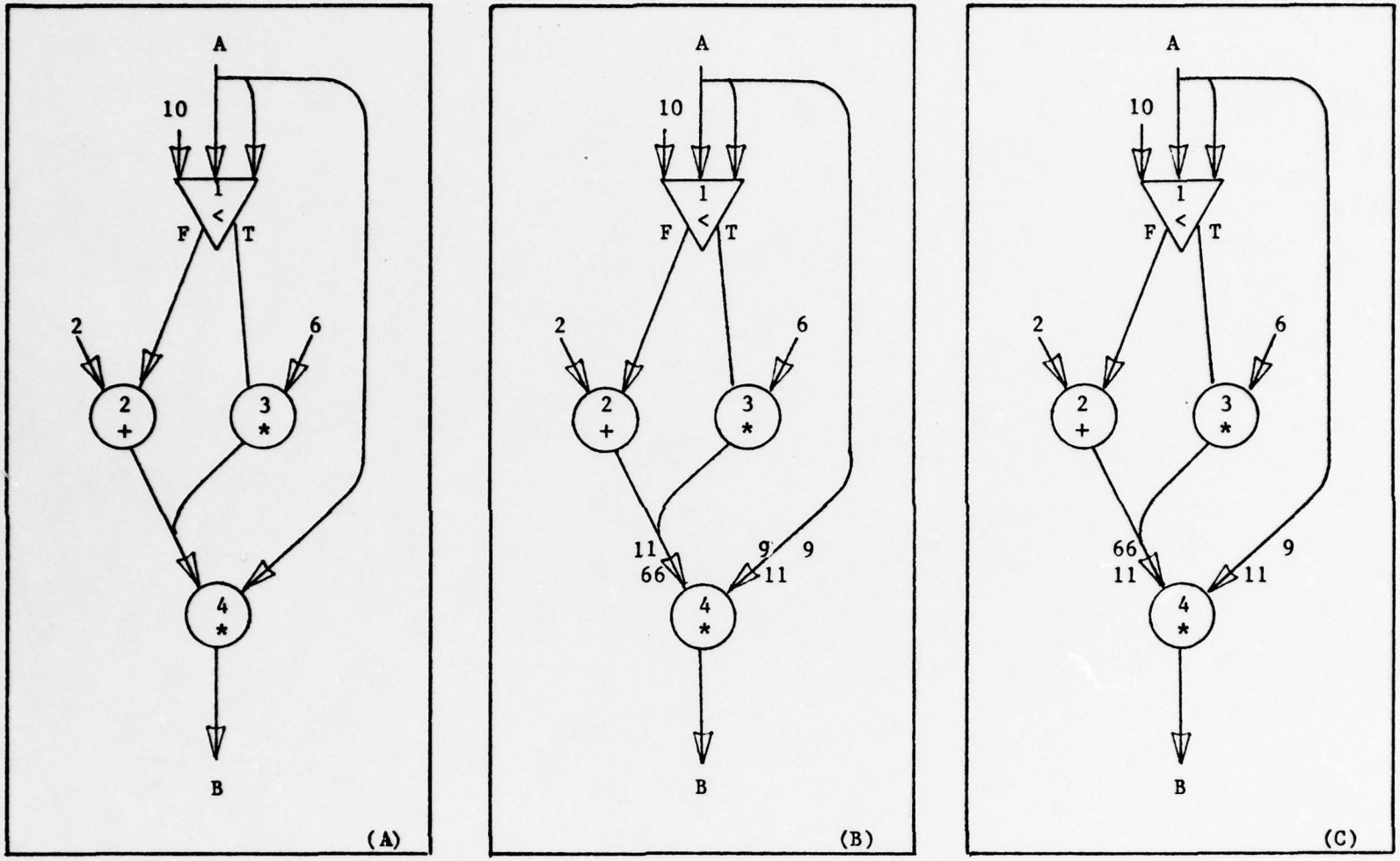


Fig 3. Non-determinism Example Two

this example, we will assume that the second input will come into the system while the system is still processing the first input. Assume further, that inputs to a node queue on the input arcs to that node. The input stream will be 11 and 9. The 11 will go through the conditional true branch to the multiply, and the 9 will then go through the conditional to the add. If the multiply operator finishes before the add then the system will appear as shown in Figure 3(b). If the addition finishes before the multiply then the system will appear as shown in Figure 3(c). Clearly, when the multiply at node 5 operates on the data from the two cases, the results will be different. In this example of indeterminacy, the values of the data flow program can be changed by timing differences when passing through conditionals.

Difficulty of Data Structures

Data structures also can cause problems. Conceptually, if not physically, the data to be used by a node in a data flow program is passed directly to the node by the arc. Physically, passing values through the data flow program is practical with simple data types such as real and integer numbers, but not in general. Record structures and arrays are large data structures that cannot be efficiently moved through the graph. For example, the overhead alone of moving a 50,000 character record from node to node would defeat any possible savings in multiprocessing speed.

The concept of single use data structures allows some flexibility (Ref 13). These structures, once created, are never changed. Rather they are "modified" by replicating the structure with the changes incorporated. If the structure cannot be changed by nodes to which it

is passed, then the structure may be passed by passing pointers. In addition, the same occurrence of the data structure may be passed to several destination nodes. Though this technique solves many of the problems of passing large data structures, it does not eliminate them. For example, using the single use technique, minor changes to a large structure result in replication of the entire structure.

Difficulty of Understanding

As seen in the examples, the level of complexity of a data flow program is significantly higher than that of the corresponding Pascal program. As more complicated programs are expressed in data flow, the graphs become even more complex. In addition, programmers are not familiar with data flow notation because it is a departure from their programming experience. These factors contribute to make it substantially harder to write a data flow program than an equivalent Pascal program.

Improvements

Though some of the problems discussed previously are inherent in data flow, most can be eliminated through one change. Instead of programming directly in the data flow notation, program in an existing HOL, and translate the source into data flow notation. It has been proven that all ALGOL like programs can be mechanically translated to data flow (Ref 24). This approach will have the following effects: decreased errors, increased transportability, and increased programmer acceptance.

Decreased Errors. Data flow is not inherently more error prone than other programming practices. However, humans cannot easily

understand the maze of arcs and operators present in even a simple flow graph. This inability to understand promotes errors. By hiding the data flow nature of the processor from the programmer, the use of an HOL allows the programmer to use a media with which he is familiar.

At the same time, determinacy of the data flow program can be assured by compiling from an HOL. The compiler could be constructed so that it will only produce deterministic programs from the source code. This appears feasable as determinacy arrises from the cases described above. If the compiler specifically avoids the above cases it should be possible to prove determinacy of the generated data flow programs.

Transportability. There is a wealth of existing software today, most of it is written in one or another HOL. If a compiler existed that translated from an HOL to data flow notation, existing software could be used in data flow processors without costly rewriting.

Programmer Acceptance. Most programmers of today have been taught to program using one or more of the standard HOLs, such as Pascal, ALGOL, or FORTRAN. Data flow is significantly different from all of these. Though new techniques may be more efficient there is a strong tendency for programmers to use techniques with which they are familiar. By hiding the flow nature of the machine, the programmer will be able to continue programming in a language he is familiar with while having the advantage of data flow.

Summary

Though there are problems with the use of data flow, it does allow expression of algorithms so that the system, rather than the programmer,

can determine where parallel execution is possible. Because the nodes wait for their inputs and are independent after all inputs are present, it would be relatively simple to develop a multiprocessor based on current microprocessor technology that would be able to execute data flow programs. The remainder of this report investigates the amount of parallelism possible in such a data flow microprocessor.

III. SIMULATION

To evaluate data flow in the single user scenario it is necessary to have a data flow processor. Initially, it was felt that a processor could be designed for evaluation. However, on examination of the number of design options available it became clear that choice of a specific set of design options would be arbitrary and have significant effect on the performance of such a processor. It was decided, therefore, to first develop an event driven data flow simulator. The simulator would be designed to allow rapid change of those design constraints, giving a basis for selection of options on a real data flow processor. To insure that the simulator could be easily modified, it will be strictly implemented in software, and written with modularity and modifiability in mind rather than efficiency. The design development and initial test of the simulator will be described in the following paragraphs.

Design of the Simulator

In this section, the initial design choices involved in the data flow simulator will be discussed.

General Design Goals. The simulator is intended to be a test bed to make the choice of design options in a data flow processor easier. The purpose here define a set of features that will make the simulator both simple to design, easy to use, and adaptable to new methods of implementation as they are developed. The following represent features desirable in a test be to meet the above goals:

1. Must be easy to modify.
2. Must have convenient means of inputting test data flow program.

3. Must be heavily instrumented, to allow monitoring of the internal workings of the test data flow processor (such as movement of data, transfer of control, and ocurrance of overheads).
4. Must be realistic. That is, must not place artificial restrictions on the system. For example, if a data flow processor can perform additions, then the simulator must perform additions. In addition, when the same data flow program is run on the simulator and a real data flow processor, they should produce identical reaults. Realism in this context does not mean real time nor that the simulator will be as fast as an actual data flow processor.
5. Must be responsive. The simulator should provide reasonably fast response. It should be reliable, in that it will repeatably produce identical results.

Use of Software Engineering. When trying to meet the above goals, it is necessary for the designer to define how the system will be used. Interestingly, the software engineering technique has almost the same name as the processor: Data Flow Analysis (Ref 12:47-62). In this technique, the flow of data to and from all elements of a system is shown. The level of detail is then further defined by expanding the flows of data and the functions operating on them. Functionally, the data flow diagrams used in software engineering bear the same relationship to data flow programs executed by a data flow processor, as a natural language such as English bears to a programming language such as Pascal. The data flow diagram (DFD) used in software engineering is intended to be interpreted by humans. It does not

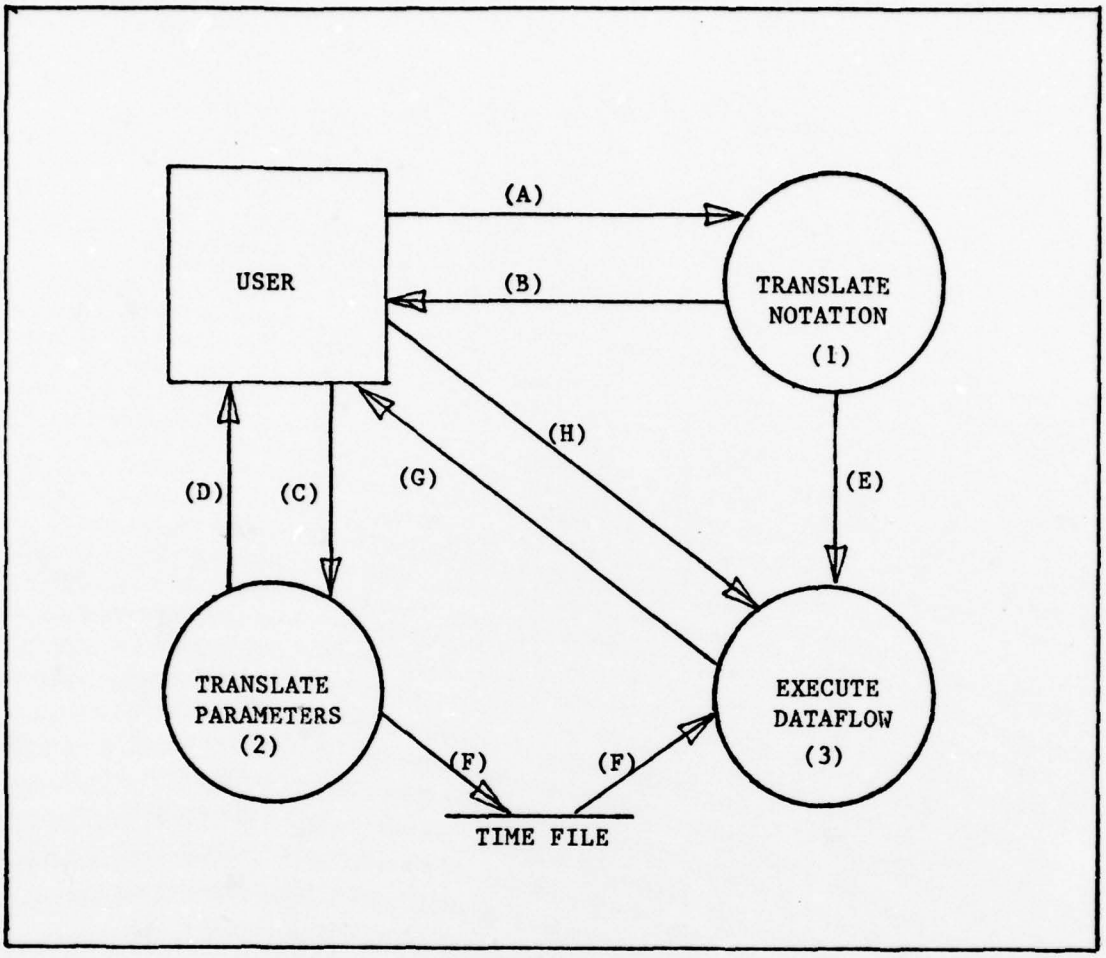


Fig 4. Top Level Data Flow Diagram

TABLE II

Data Dictionary for Simulator DFD

In this table, uppercase words represent data dictionary elements. Lowercase words are descriptive of the meaning of data.

Data Element Name	Definition
(A). NOTATION	= {TOKENS}
(B). ERRORS	= Errors detected in Parsing
(C). MACHINE PARAMETERS	= Text machine parameters
(D). PROMPTS	= Prompt Strings for User
(E). INIT-STATE	= {ENABLED NODES} + {PROGRAM STEP} + {VALUES}
(F). PARAMETERS	= Machine readable parameters
(G). OUTPUTS	= Results of the execution
(H). REPORTS	= Summary of statistics
(I). DISCONNECTED GRAPH	= Nodes and inputs (no arcs)
(J). TOKENS	= Tokens of the dataflow notation
(K). PROGRAM STEP	= One node, its inputs, and its destinations
(L). ENABLED NODES	= Nodes, and time completed
(M). VALUES	= Numerical value
(N). DEST-VALUES	= VALUE + destination
(O). TIMES	= Execution time for an operator

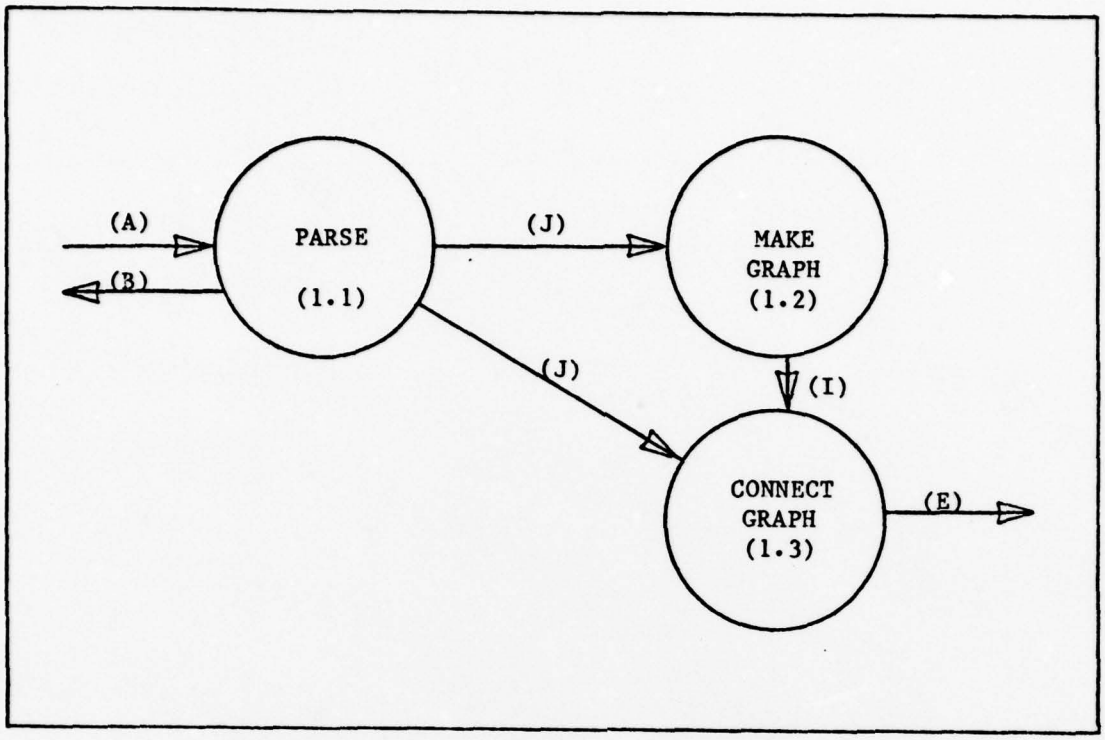


Fig 5. Subordinate DFD of TRANSLATE NOTATION

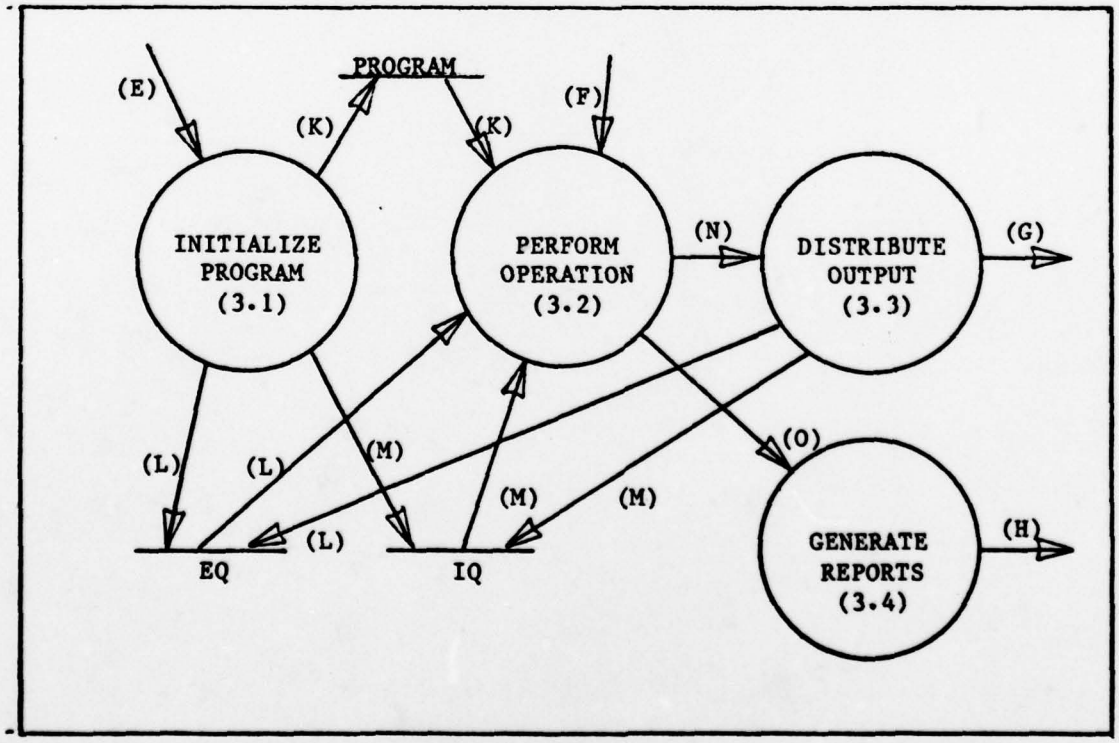


Fig 6. Subordinate DFD of EXECUTE DATAFLOW

pretend to be an exact statement of an algorithm; rather it is a guide describing the movement of data within a system. The data flow program on the other hand, will be executed by a machine. There can be no room for interpretation in the meaning of a data flow program, or the results would be unpredictable. The following discussion, shows the general organization of the simulator.

Figure 4 shows the top level of interaction in the data flow simulator. Table II is the corresponding data dictionary. To read the diagrams, each node of the DFD corresponds to a function that needs to be performed either by the human using the system or by the simulator. Arcs of the DFD represent the flow of data to and from the nodes. Each arc is numbered; the number corresponds to an entry in the data dictionary which describes the information following that path.

The DFD shown in Figure 4 alone is not sufficient to describe the actions taking place within the simulator, therefore, the more complicated nodes are further divided. Figures 5 and 6 show the breakdown of the "TRANSLATE NOTATION" and the "EXECUTE DATAFLOW" nodes respectively. All nodes not further subdivided into other nodes are then described in structured English (Ref 12:117-214). That means, described in a clear algorithmic fashon. The following is the structured English definition of the nodes in Figures 4, 5, and 6:

```
TRANSLATE PARAMETERS-
   Prompt user for old execution times file.
   Open and read file.
   While not done do
        Display operation time values.
        Prompt user for parameter.
        Read value.
   Write new execution times file.
```

```
PARSE-
   Repeat
        Read a char from file
        If delimiter then return token and delimeter.

MAKE GRAPH-
   For each node:
         Allocate space for node.
         Link into node chain in alphabetical
                sequence.
         Determine number of inputs.
         Generate linked list of inputs.

CONNECT GRAPH-
   For each node:
         For each input:
              If there is a literal value for this
                 input then set the input to that
                 value.
         For each destination:
              Locate destination node and input.
              Add to destination linked list.

INITIALIZE PROGRAM-
   Set up user specifyable parameters:
         Number of processors, debug options,
         list file names, program file name.
   Read in the program.
   Set up queue of enabled nodes.

PERFORM OPERATION-
   While there are free processors and there are nodes
   in the enabled node queue:
         Remove one node from enabled node queue.
         Calculate completion time of node.
         Place in correct time sequence in processor
              list.
   Remove the next node from the processor list.
   Update clock to reflect completion time.
   Do the node's operation.
   Keep statistics.

DISTRIBUTE OUTPUT-
   For each destination:
         Place correct value in input queue.
         If the queue was empty then
             decriment the destination node's wait
             count.
             if the count is zero then queue the node
             in the enabled node queue.
```

GENERATE REPORTS-
Print out summary reports from statistics.

Design Choices.

Design Choices. A series of relatively high level decisions must be made to meet the goals of the system. In the following discussion, each of the choices is directly addressed to one of the design goals. As in all engineering efforts, there is always compromise of cost, schedule and performance. The decisions selected here are not necessarily the optimal given unlimited funds and support. They are, rather, selected to maximize performance of the system in light of the practical constraints placed on its development. When, such a non-optimal decision is selected, the reasons for it will be stated.

Easy to Modify. Use UCSD-Pascal as the high order language to program the system. To meet the goal of modifyability, the system must be implemented in an efficient programming environment. Assembly language is not practical for the main simulation because the programmer has the burden of direct machine interface. Relative programming efficiency of Pascal over assembly is very high. The Pascal supports recursive programs and sophisticated data structures which are difficult to implement in assembly. BASIC and FORTRAN were rejected because neither of them offer the sophistication of the Pascal programming environment. In addition, BASIC code is generally difficult to modify because short variable names cause programmer confusion. FORTRAN would be a minimally acceptable language even though it does not contain the programming flexibility desirable for this task. It is a very

common language which most engineers know. Therefore, programs written in FORTRAN are likely to be better understood by a wider audience. However, when this project began there was no FORTRAN compiler available on the Intel Single Board Computer (SBC), therefore, the suitability of FORTRAN is academic. UCSD-Pascal supports the facilities mentioned above and has several other advantages. It supports structured programming. It is widely available on a number of mini and microcomputers. It contains an excellent programming development facility including program libraries, separate module compilation, and an interactive screen oriented editor for easy program modification.

Responsive. Use Intel single board computer (SBC) as development system. The primary computer available, the Control Data CYBER, does not meet the level of convenience needed for this project. Although a much more powerful computer than the SBC, it is a time shared system. During peak load times, the turnaround time can be excessive. It also has limited availability during weekends and at nights, and there are restrictive storage limitations imposed by the number of users. The SBC, being a dedicated system is available whenever the user is present. The Intel Multibus allows multiprocessors to share the same system bus without undue interferance, and there are several similar systems (The loboratory has two Intel SBC 80/20s, an Intel Series II which uses the same instruction set, and a PDP LSI-11 whic also can execute UCSD Pascal programs) that can be used in the event

of a hardware failure in the SBC system. A further benefit of the small system is that it is a "hands on system" the user can halt execution of a program at will and resume execution at a later time, or he can simply observe the operation of the system under different circumstances.

Easy to Input Programs. Data flow programs are directed graphs. It is impractical to input a digraph directly into a computer. Therefore, it is necessary to develop a notation for expressing data flow programs in machine readable form. The simplest method to express a graph is a connectivity table. This is the basis of the notation. It seems likely that some extensions over a traditional connectivity table will be necessary but these should be minimized.

Heavily Instrumented and Realistic. Use an event driven simulation. There are several possible ways to model the data flow processor. The difference lies chiefly at the level at which the modeling is to be done. For example, the simulation could model the actual movement of signals from gate to gate, or the model could have loosely shown the system as a whole without attending to the specific tasks performed by a processor. Because it is the intent of this study to evaluate the best architecture for data flow processors (within the constraints listed in assumptions), an event driven simulation appears best. In such a simulation, execution of each node of a data flow program would constitute an event. The simulator will keep records as each node is processed and update a pseudo clock. The pseudo

clock will show the amount of elapsed time that the simulator estimates a real data flow processor would take. The model contains all of the functional operations of the processor without necessarily retaining all of its structure. Because the simulator functionally models a data flow processor, changing it to represent any specific architecture is simplified.

Data Flow Notation. The notation developed for the simulator is intended to be a concise method for expressing the connections within a data flow program. It is intended to be easy to write and to understand, both for the programmer and for the parser in the simulator.

To concisely define the syntax of the notation, Bacus Naur Form (BNF) will be employed (Ref 3:125-129). The BNF of the data flow notation is contained in Table III. The notation is blank sensitive, meaning that, blanks cannot be imbedded within tokens. Otherwise, the notation is completely free format. Figure 7 gives an example of both a data flow program in graphical form and in the specified notation. Note that each node in the flow program is given a distinct name, which is the same name that will be used for the node in the flow notation. Each node is completely described by a single statement, a string of characters ending in a semicolon. The following paragraph defines the meaning of fields within the statement:

First, the node name is a string of alphanumeric characters. The operation type follows; it is a single character with meaning shown in Table IV. Next come two integers, the first indicates how many inputs the node will have, the second defines the number of inputs to wait for

TABLE III

Bacus Nauer Form Definition of Data Flow Notation

Token		Definition
<flowprog>	::=	<null> | <dataflow> <flowprog>
<dtatflow>	::=	<flohead> <litlist> $ <outdest> ;
<flohead>	::=	<floname> <operator> <minputs> <nwait>
<litlist>	::=	<null> | <inum> : <ival> <litlist>
<outdest>	::=	<null> | <floname> : <inum> <outdest>
<operator>	::=	+ | - | * | / | ^ | < | > | = | # |M | R | C | O | I
<inum> <ninputs> <nwait>	::=	<pos integer>
<ival>	=	<integer>
<floname>	::=	<letter> | <floname> <alphanumeric>

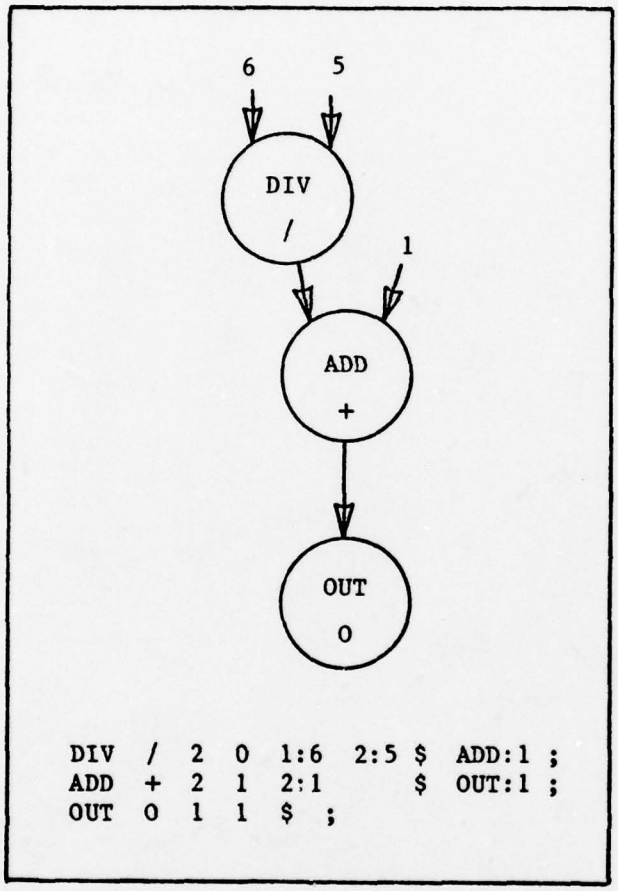


Fig 7. Example of Data Flow Notation

TABLE IV

Operators Defined in the Simulator

Op	Name	Definition
+	add	takes 2 inputs and operates
-	sub	passing result to all outputs
*	mpy	
/	div	
^	exp	
C	literal	passes literal value to all outputs
>	if gt	Compare inputs 1 and 2 then
<	if lt	pass inputs 3..n to outputs
#	if ne	1,3,5...n-1 for test=true,
=	if eq	outputs 2,4,6..n for false
R	replicate	Pass sole input to all outputs
M	merge	Wait for all inputs, then pass all to their respective destinations. Input 1 to output 1, input 2 to output 2, etc. Note: this is not the same as the merge operation is usually defined. (Ref 5:93)
I	input	Read single value from console output to all destinations
O	output	Output all inputs to list device

before beginning execution. If all of the nodes inputs were supplied by incoming arcs, the two integers would be the same. If one or more of the inputs were literals, that is constant values permanently attached to the input of a node, then the wait count is decreased by one for each literal input. Next, optionally, is the definition of literal inputs: input number, colon, input value. A dollar sign separates the input definitions from the output arc definitions. Lastly, for each output arc: (there may be none) the destination node name, colon, input number.

Using the above notation, any data flow program that can be expressed graphically may be input into the simulator. The simulator provides syntax checking during the input of the program to aid in the proper generation of the program. It also will output the state of the data flow program including all input values and arcs before execution. Optionally, the simulator will output diagnostic messages during and after the execution of the data flow program. The diagnostics include but are not limited to: state of all nodes, inputs present at nodes, number of processors busy, and amount of time used by the simulated machine.

Structure of the Simulator. Using the software engineering DFD produced for the simulator, afferent (input) and efferent (output) sections were isolated. The preliminary structure chart is shown in Figure 8. From this structure chart the major subareas of the simulator were defined.

The final structure of the simulator is shown in Figure 9. It can be seen that there are two afferent branches: one for input of the simulated machines parameters, and one for input of the data flow

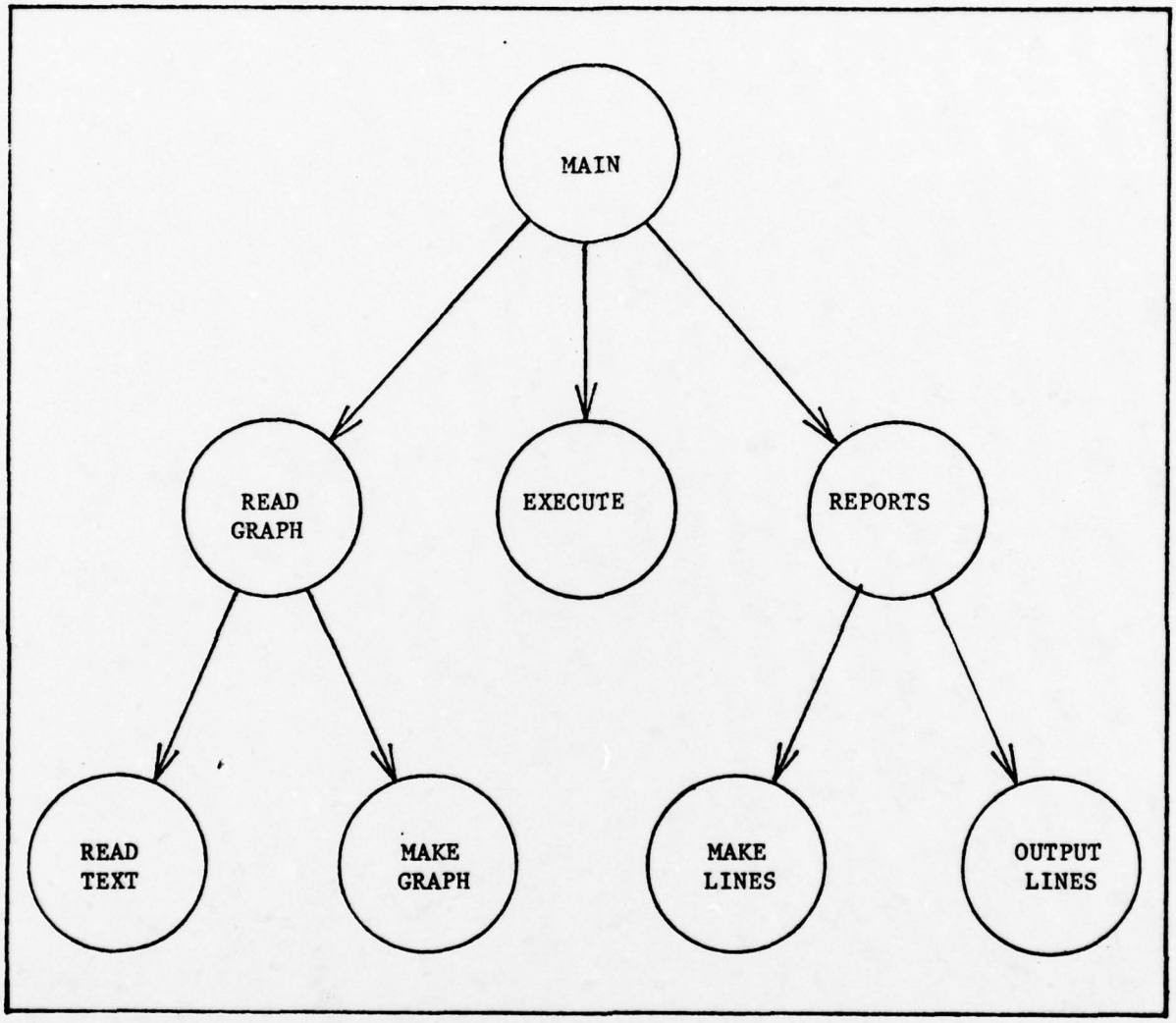


Fig 8. Preliminary Structure Chart of Simulator

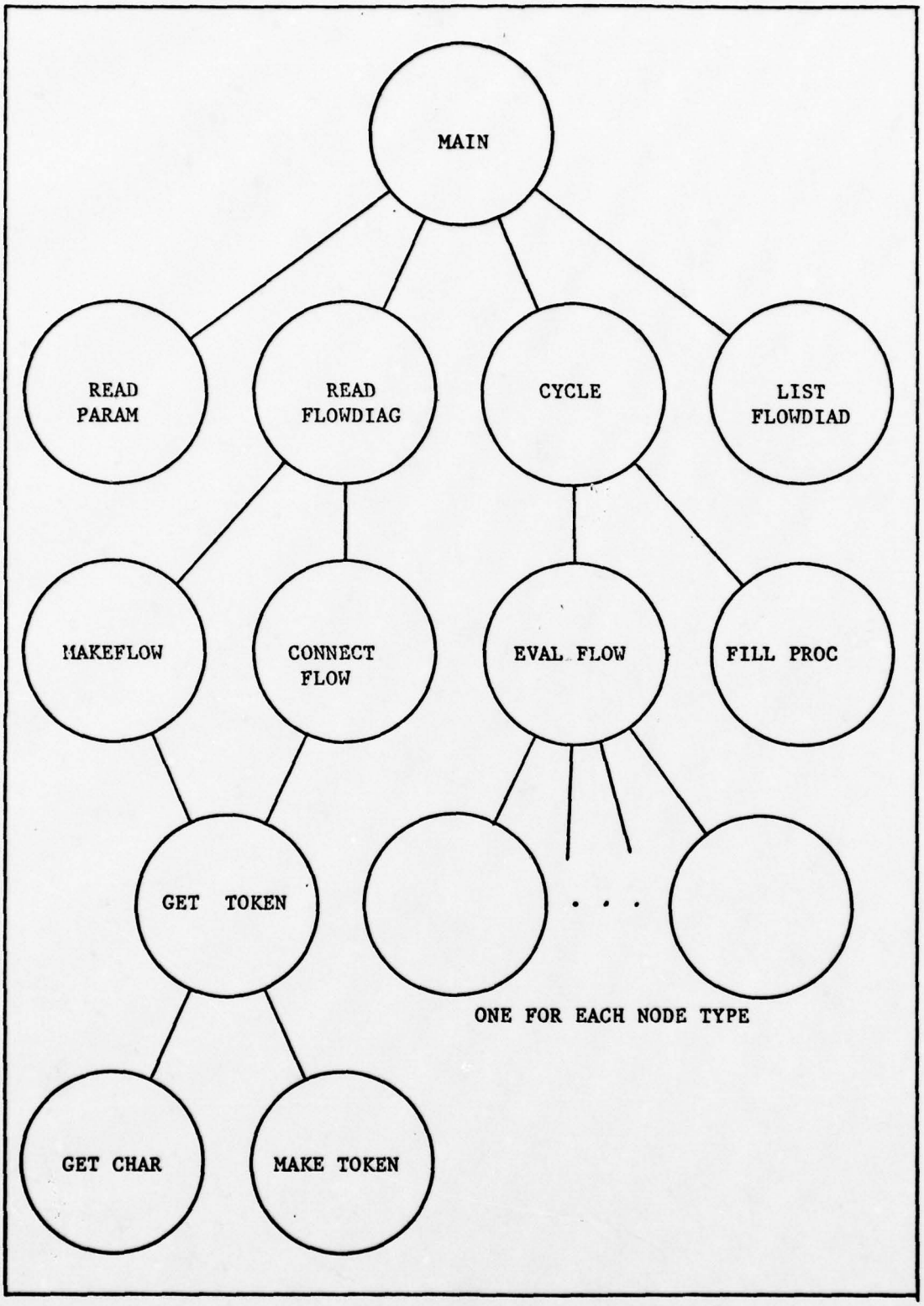


FIG 9. Final Structure Chart of Simulator

program. There are deviations from the traditional efferent branch because it is necessary to output signifigant amount of diagnostic messages to the user during operation of the simulator. Because of this output the efferent branch becomes muddled with the central transform, the simulation itself. There also is a true efferent branch that produces the final report of the execution of the simulator.

Implememtation of the Simulator

The discussion of implementation will be broken into three sections: data structures, logical organization of the simulator, and physical organization. The dicotomy between logical and physical organization is caused by software limits of the UCSD Pascal implementation (Ref 23:155). In a traditional Pascal program, all procedures must be internal to the main program. This is both impractical and impossible on microcomputers. The slow compilation time and limited central memory forces separate module compilation. The simulator is therefore divided roughly along functional boundaries into separately compilable modules (UNITS); this allocation of procedures to UNITS that is discussed in the section on physical organization.

Data Structures. Pascal allows the use of a variety of different data structures in the implementation of a Pascal program. Data defined as static in the Wirth Pascal Manual (Ref 25) will not be discussed here because it is clearly defined in the data declarations within the code. Those data structures that are dynamic, by Wrirth's definition, are described here because their use depends on where new invocations of then are called in the code. There are six major data

```
    FLOW = RECORD
           FLONUM:INTEGER;{sequential number of flow}
           FLONAME: STRING;{name of this node}
           NFLOW: ^FLOW;{pointer to next node
                        in alphabetical list}
           NENABLED  : ^FLOW;{pointer to next enabled
                             queue, if not enabled then
                             NENABLED is NIL}
           OP   :OPERATOR;
           NINPUT:INTEGER;{number of inputs to this node}
           NWAIT :INTEGER;{number of inputs needed to
                          activate the node}
           INS  :^INPUTS;{pointer to list of inputs}
           OUTS :^OUTPUTS;{pointer to list of outputs}
           END;

INPUTS  =  RECORD
           IQF,IQL: ^INVALS;{pointers to input value queues}
           NIN  :^INPUTS;{pointer to next input of node}
           END;

INVALS  =  RECORD
           VFULL:FULLTYPES;{type of value}
           VALUE:INTEGER;{value of this input}
           NINVAL: ^INVALS;{pointer to next queued value}
           END;

OUTPUTS =  RECORD
           FLOW :^FLOW;{ptr to destination node}
           INUM:INTEGER;{number of the destination input}
           INPTR:^INPUTS;{direct pointer to destination}
           NOUT : ^OUTPUTS;{pointer to next output of node}
           END;

PROCESSCHAIN = RECORD
               NPRO : ^PROCESSCHAIN;{pointer to next
                      processchain entry}
               PFLO : FLOWPTR;{pointer to executing node}
               TIM  : REAL;
               END;
```

Figure 10. Connection Between Data Structures in the Simulator Database

structures used in the simulator. They are organized as queues, stacks, alphabetic lists, or time ordered lists. Figure 10 shows in general the connections between the various data structures. Each structure is described in detail below:

FLOW. This is an alphabetically linked list of nodes in the data flow program. Each record of the chain contains the node name, its number(a sequential number assigned in the order nodes are found in the notation), the operator, the number of inputs expected and the number remaining before this activation. It also contains pointers to its list of inputs and outputs(see INPUTS and OUTPUTS).

PROCESSCHAIN. This is the time ordered list of currently executing nodes. The chain cannot be longer than the number of processors in the simulated system.

INPUTS. This is the chain of inputs for each node. Position in the chain indicates the input number. Each record contains pointers to maintain the queue of input values(INVALS) for that input.

INVALS. This is the queue of input values for each INPUTS record.

OUTPUTS. This is the chain of outputs for each node. Position in the chain indicates the output number. The number is signifigant for some operation types such as conditionals, which use position in this list to determine if this is a true or a false branch. Records contain the destination node pointer, input number, and a direct pointer to the corresponding INPUTS record(this is redundant but saves

execution time in the simulator).

ENABLQ. This is the queue of enabled nodes. This is used to make the next assignment to the PROCESSCHAIN when a processor becomes free.

Dynamic storage recovery. Many of the above data structures are allocated and deallocated dynamically during the execution of the simulation. UCSD Pascal does not support the Pascal DISPOSE statement. Therefore, the Pascal system has no way to deallocate a structure once created. To meet the dynamic needs of the simulation a set of stacks of currently unused structures is maintained. When a type of structure is needed the stack is first checked, if it has a record on it then that record is given to the requestor. If the stack is empty then a NEW() element is created. To return a record, it is simply pushed on the appropriate stack. Most unused record stack pointers start with the letters FREE then the name of the stacked record. For example, some stack pointers are: FREEENABLQ or FREEINVAL.

Logical organization of the Simulation. The procedure calling chart of Table V shows each procedure and all subordinate procedures. In the table there are two kinds of procedures noted: procedures and segment procedures. Segment procedures are a feature of UCSD Pascal which allows the swapping of code to and from disk. The use of segment procedures is identical to that of conventional procedures, even though their implementation is different. This feature signifigantly reduces the core consumption of a program without adding the complexity of overlays.

TABLE V

Procedure Call Chart

Procedure	Calls
MAIN	HWTEST, CYCLE, READEXTIME, READFLOWDIAG,LISTFLOWDIAG
HWTEST (SEGMENT)	HWDEBUG
CYCLE (SEGMENT)	GETPROC, RETPROC, FILLPROCESSORS, EVALFLOW, LISTFLOWDIAG
GETPROC	none
RETPROC	none
FILLPROCESSORS	OPEXTIME, DEQUEUEENABLEDFLOW
EVALFLOW	ARITHGP, LITGP, IFGP, REPGP, MERGEGP, IOGP
OPEXTIME	none
DEQUEUEENABLEDFLOW	none
ARITHGP	POPINVAL, SETINPUT, RESETFLOW
LITGP	POPINVAL, SETINPUT, RESETFLOW
IFGP	POPINVAL, SETINPUT, RESETFLOW
REPGP	POPINVAL, SETINPUT, RESETFLOW
MERGEGP	POPINVAL, SETINPUT, RESETFLOW
IOGP	POPINVAL, SETINPUT, RESETFLOW
POPINVAL	none
SETINPUT	QUEUEENABLEDFLOW, PUSHINVAL
RESETFLOW	PEEKINVAL, QUEUEENABLEDFLOW
LISTFLOWDIAG	none
READEXTIME	none
READFLOWDIAG	MAKEFLOW, CONNECTFLOW
MAKEFLOW	GETLTOK, FINDINPUT, FINDFLOW, LISTFLOWDIAG
CONNECTFLOW	GETLTOK, FINDINPUT, FINDFLOW, LISTFLOWDIAG

The following is a discussion of the major procedures within the simulator:

MAIN- The main program body. This section does initialization of the various queue pointers and record keeping variables. It also initializes the files for input. It sequences the reading of the data flow program and its execution.

HWTEST- Produces a compressed file containing the data flow program for use by the hardware software simulation.

CYCLE- This routine causes one event in the simulator to occur. One event is the termination of a node and the assignment of as many processors as is possible within the limits of precidence relations and number of processors.

GETPROC- Gets a processor activation record from free storage.

RETPROC- Returns a processor activation record to free storage.

FILLPROCESSORS- Attempts to assign a node of the graph to all unused processors. It then puts the now busy processor activation records in their proper place in the PBUSY list, such that, all records before it in the list represent nodes that will complete before this one. Note: because the processors are in fact simulated there is no distinction between processors here, only count of the number busy.

OPEXTIME- A function used to determine the execution time for a particular operation type.

DEQUEUEENABLEDFLOW- Returns the next enabled node on the enabled node queue.

QUEUEENABLEDFLOW- Puts a node on the enabled node queue.

EVALFLOW- This routine causes the actual operation of the node

to take place. It uses several routines directly: ARITHGP, LITGP, IFGP, REPGP, MERGEGP, and IOGP. Each of these routines corresponds directly to one of the groups of instructions available in the data flow notation.

POPINVAL- Returns the next input value from a particular input of a particular node. The module name is misleading because the inputs are queued not stacked.

PEEKINVAL- Similar to POPINVAL except that PEEKINVAL does not remove the value from the input. It is used to see if there is an input queued without altering the value of the input.

PUSHINVAL- Puts a value into the queue of inputs for a particular input for a particular node.

SETINPUT- Uses PUSHINVAL to queue a value on a nodes input. Also, if this is the first input in the queue, it decrements the node's wait count by one. If the wait count reaches zero PUSHINVAL then queues the node for execution using QUEUEENABLEDFLOW.

LISTFLOWDIAG- Traces through the data flow program listing all nodes, all arcs, all input values currently on node inputs, and all enabled nodes.

READEXTIME- Reads in the file of execution times to be used in calculating the execution times for data flow programs.

READFLOWDIAG- This routine directs the parsing of the data flow notation. Reading the notation is a two pass process.

MAKEFLOW- Pass one of the notation input. This routine makes all of the nodes and their inputs.

CONNECTFLOW- Pass two of the notation input. Connects all arcs

within the data flow program.

GETLTOK- Low level routine. Returns the next token from the input stream.

FINDINPUT- This routine given the node name and input number returns a direct pointer. This direct pointer saves considerable time during execution of the program.

FINDFLOW- This routine given a node name, returns a pointer to the node.

OPEXTIMES- This is a separate main program. It generates the file of operation execution times.

Physical Organization. UCSD Pascal allows separate compilation of procedures in a structure called a UNIT. To make the simulator practical to compile it was necessary to use a set of UNITS. Code for each unit should be relatively independent of all other code. Therefore, the following breakdown of units was chosen:

Main Program- contains the initialization, interface to hardware data flow implementation (discussed in Section III), and high level control for the simulator.

EXECFLO- contains procedures that perform the actual execution of the data flow program, record keeping procedures, and error detection.

DFPROGS- contains the procedures to read the flow notation, make the linked lists which are the internal machine representation of the data flow programs, and to list the internal data flow program representation in printed form.

UTILITIES- contains routines to manipulate text files including the lexical scanner of the data flow notation.

Initial Results of Simulation

The simulator was tested using several simple data flow programs. Initially, the programs were chosen for their ease of data flow coding. Then the complexity of the programs was increased. The simulator showed that several processors could be applied to a single task. The number of tests was limited, but it could be seen that the application drives the number of processors used by a data flow program. The question remained: was the effective processing power of the data flow processor similar to that of a comparable single processor or was the overhead associated with queueing and dequeueing tasks and values consuming the additional processing power available.

Summary

The simulator was designed to allow evaluation of data flow; it does this through an event driven simulation of the execution of a data flow processor. The simulator can be configured to operate with an arbitrary number of processors and arbitrary node execution times. Its design is, also, intended to simplify changes to the data flow architecture.

Some speculation at this point provides a basis for the remainder of this report. During its early testing, the simulation showed that the use of data flow techniques can, in theory, produce improvements over conventional processors of similar speeds. Can a data flow processor be designed using today's technology, using several microcomputers, and can that processor compete favorably with conventional processors using the same technology? Section IV discusses the development of a data flow processor using microcomputers.

IV. Hardware/Software Implementation

Purpose of a Hardware Implementation

The intent of implementing a data flow processor using a current single chip computer is to determine if current technology can easily be adapted toward data flow processing. The simulation provides a detailed examination of the inner workings of the simulated data flow processor but it does not provide assurance that such a processor is physically realizable. In addition, the implementation of a data flow processor gives significant insignt into the amount of processing necessary to perform the steps involved in data flow processing. In that respect, the hardware implementation provided a means of estimating parameters used in the simulation.

Because the intent is to show feasability and to estimate execution time, many of the features that would be found in a practical processor will be omitted in this hardware implementation. Specifically, the number of operations available at the nodes is signifigantly decreased. The operations selected were the minimum necessary to demonstrate a data flow processor. The following discussion describes the design and implementation of the hardware data flow processor.

Design of the Data Flow Processor

The discussion of the data flow processor design will be broken into three sections: the development system, the data structures of the

simulator, and the operation of major modules.

Development System. The development system selected for the hardware implementation must provide the general features described in chapter II for the test bed development system and it must provide multiprocessors. The only system available is a multiprocessor comprised of Intel Single Board 80/20 series computers. This system was selected for a number of reasons: it supports multiprocessor access to a common memory, it supports interlock functions, a single processor version used in the development of the simulator, and two processors were available to construct a multiprocessor.

To use the computers in a multiprocessor configuration, relatively few changes were necessary. There are three types of interference that can occur between multiple processors: mutual exclusion of the buss, mutual exclusion of memory, and mutual exclusion of tables. All three had to be addressed in development of the multiprocessor. The SBC line of Intel microcumputers was designed to operate with multiple processors on the same system buss therefore two of the three types of exclusion are procvided by the hardware.

Buss Exclusion. This means that when one processor needs the system buss to perform input, output or memory operations, other processors will be prevented from interfering. The Intel computers have two means of buss arbitration: parallel and serial. In the parallel, a processor wanting access to the buss sends a request to the buss arbiter. The arbiter determines using its priority scheme which request should be granted. In the serial method all masters on the buss are arranged in order of their priority. When a processor wants

access to the buss checks the buss busy line (BUSY). If the buss is not busy then it raises its BPRO line telling all lower priority masters that the buss is needed then it checks the next higher master on the buss. If the next higher master has not requested the buss before the next falling clock edge then the processor has access to the buss. It then raises the BUSY line telling all masters that the buss is in use. Because the serial approach is a daisy chain, there is a maximum number of masters that can be served at a particular buss clock rate. If there are too many masters or the clock rate is too high, the signals will not have time to propagate through the daisy chain. Though the serial method is limited in the number of masters, it is simple to use. The number of masters that can be used in the SBC 80/20 system could be up to three before the delay in the daisy chain would cause problems. Because the system used here will have no more than three masters the serial approach meets its needs.

Memory Exclusion. Memory exclusion here means the ability to lock other processors out of memory for several instructions. In the Intel system there is a command called a buss override. This command causes the buss interface circuit in the SBC 80/20 to gain command of the buss as described above. When it has command of the buss, however, the circuit is not to release the buss. All other processors are therefore locked out of the buss. This command allows a processor to for example safely access and update common data structures. No other processor can interfere because none of

the other processors can access the buss. When the critical phase is completed the processor can issue a buss release and allow other processors access to the buss again.

Table Exclusion. Excluding all other processors from memory will clearly exclude them from the tables, but if one processor was making a series of updates to a table it would be wasteful to prevent all other processors from doing anything during that period. Exclusion at the level of tables can be simply implemented in software by the following algorithm:

1. Excluding all processors from memory
2. Check a flag indicating if the table is in use.
   If it is in use: release memory, wait a while, and go to 1.
   If it is not in use go to 3.
3. Set table busy flag.
4. Release memory.

The table may then be released by simply setting the flag to not busy.

Structure of the Hardware Flow Processor. The basic structure of the hardware flow processor is virtually identical to the structure of the execution section of the simulator. The implementation is quite different! In the simulation all data structures and module interfaces were selected to maximize flexibility and minimize the time needed to modify code for changes to the system. Though flexibility is desirable in the hardware implementation, it has low priority. There are fewer node types in the hardware flow processor, data structures are simplified, and arrays are used instead of linked lists.

Implementation of the Hardware Flow Processor

To visualize the hardware data flow processor, the structure diagram is shown in Figure 11. The diagram shows the major components of the system. The implementation of the flow processor may be divided into the hardware, data structures, and Pascal host software, software specific to processor 1, software specific to processor 2, the base data flow processor software, and the memory organization.

Hardware. The flow processor uses two Intel SBC 80/20 computer boards and a disk controller. The hardware system is a multiprocessor using two Intel SBC 80/20 computers, a dual floppy disk, and an ADM-3A terminal. There are some modifications necessary to allow multiprocessors to share the same buss. Processor 1 is the same processor as is used in the simulation, processor 2 is added.

There is only one modification to be made to processor 1. The on board random access memory is reconnected so that it appears in locations F800H to FFFFH. This is accomplished by jumpering wire wrap pin 117 to pin 121.

The changes to processor 2 are more substantial, though still relatively minor. The multibus needs a buss clock and a system clock. Both SBC 80/20s naturally provide this clock, therefore, the buss clock on the second processor must be disabled. The modifications are:

Remove jumper from pins 110 to 111. (Buss clock)

Remove jumper from pins 111 to 112. (System clock)

In addition, the standard SBC 80/20 read only monitor(ROM) monitor is removed from the processor 2. It is replaced by a program to allow the processor to take commands from processor 1. Figure 12 shows the program that is located in processor 2's ROM. This program

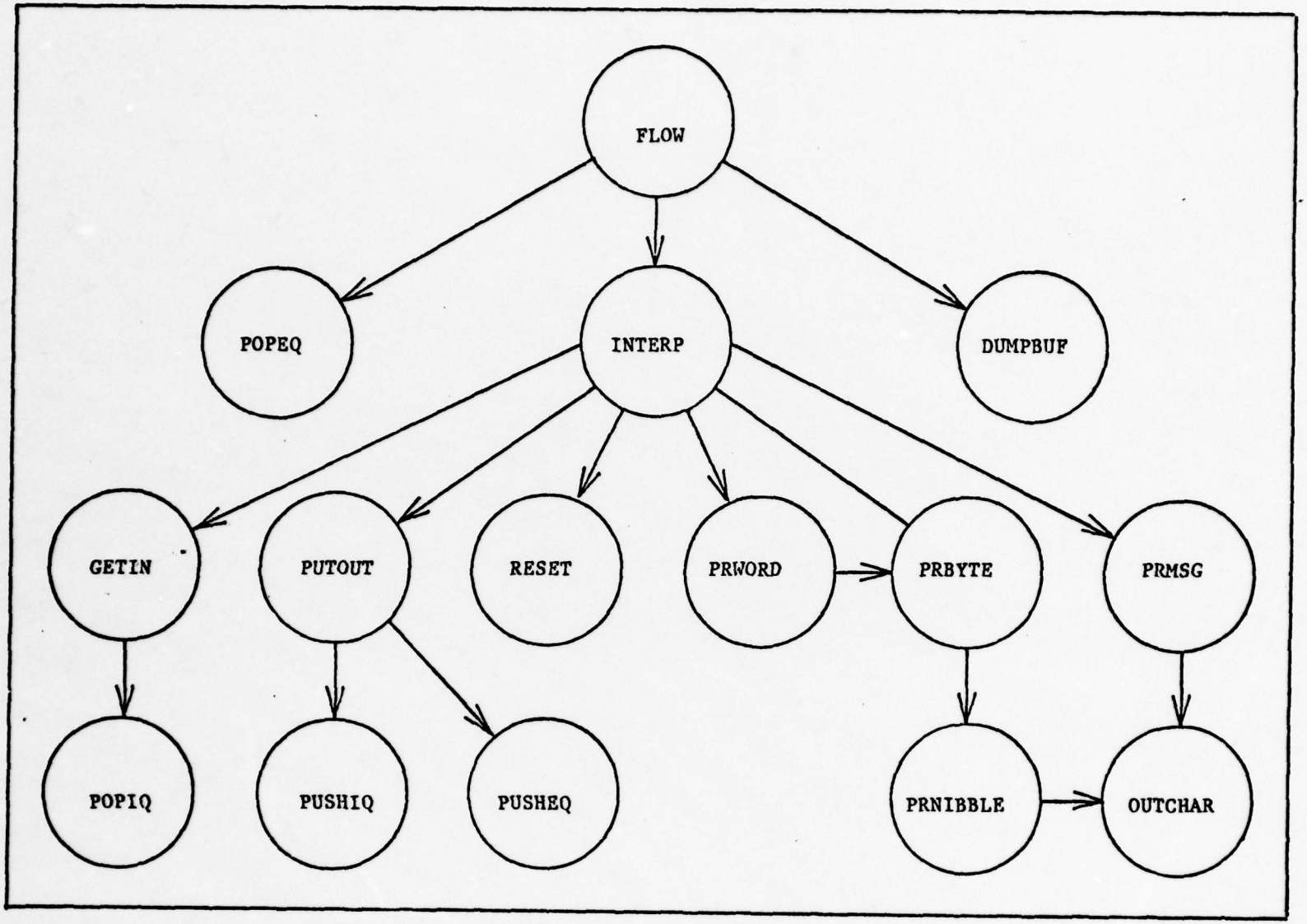


FIG 11. Structure Chart of Hardware Implementation

```
        ;THIS PROGRAM RUNS ON THE SECOND PROCESSOR IN
        ; THE DATA FLOW MULTIPROCESSOR SYSTEM
        ;
        ; IT CONTINUALLY CHECKS LOCATION 0F7FDH FOR
        ; A COMMAND TO BEGIN EXECUTION
        ;
        ; IT THEN READS THE ADDRESS IN LOCATIONS
        ; 0F7FEH AND 0F7FFH AND JUMPS TO IT.

ROM     .EQU    0
ASSIGH  .EQU    0F7FDH  ;LOCATION WHERE ASSIGNMENT WILL BE
LOC     .EQU    0F7FEH  ;LOCATION WHERE JUMP ADDRESS IS

        .ORG    ROM
        LD      A,128   ;SHORT WAIT LOOP
LOOP
        DEC     A
        JP      NZ,LOOP

        LD      A,(ASSIGN)
        CP      1               ;CHECK THAT ASSIGNMENT IS
                                ;FOR THIS PROCESSOR
        JP      Z,GOTONE         WE HAVE BEEN ASSIGNED
        LD      A,10            ;SMALL WAIT COUNT
        JP      LOOP

GOTONE
        LD      HL,(LOC)         GET JUMP ADDR
        XOR     A               ;ZERO ASSIGNMENT TO TELL
                                ;ASSIGNING PROCESSOR WE GOT
                                ;IT.
        LD      (ASSIGN),A
        JP      (HL)            ;JUMP TO ASSIGNED LOCATION
        .END
```

Fig 12. ROM Program in Processor 2

periodically checks location F7FCH. If the location has the value 1 (meaning that auxiliary processor 1 is to receive the message) the processor reads the address from location F7FDH, sets location F7FCH to zero, and jumps to the address read. When the task to which processor 2 has been assigned is complete, it simply jumps to location 0 (In the ROM program), and waits in the loop for another assignment.

This mail box method of processor assignment is used to allow multiple processors to be present in the system without interferance to UCSD Pascal or to CPM. The unused processors benignly execute their wait loops waiting for a command. When ready to test a multiprocessor application, the user program must set up a program for the second processor to execute, and then command it to begin execution.

Data Structures. In the data flow processor, all execution information is in one large array of 16 bit words. The information is essentially identical to the information in the simulation data structures, except that the data is organized in a convenient format for retrieval from assembly language. The array is broken into sections of 32 words each. The first secton is the enabled node queue, after that each section is either a node or an input queue. The formats for the queues and the node are shown in Table VI.

A second array is used as a queue for output awaiting the serial port. Both processors insert information into the queue but only processor 1 retrieves it to send to the console.

Pascal Host. The Pascal host is a Pascal program, running on processor 1, which interfaces an assembly or set of assembly programs to the UCSD Pascal P-machine. In this case the host: reads in the compressed data flow notation made by the simulator, makes the 32 word

TABLE VI

Data Structure for Hardware Implementation

ENABLEQ

Byte	Meaning
0-1	Pointer to next free space in queue
2-3	Pointer to next node pointer
4-64	Pointers to enabled nodes

NODE

Byte	Meaning
0-1	Number of inputs needed for this node to be enabled
2-3	Number of inputs needed minus those already present
4	Operator
8-35	List of addressed of input queues
36-63	List of output destinations

INQUEUE

Byte	Meaning
0-1	Pointer to node
2-3	Next empty location in queue
4-5	Next value to be removed
6-63	Queued values

sections (representing queues and nodes), assigns processor 2 to the data flow program, then jumps to the routine which puts the flow processor software into processor 1's RAM, and begins execution of the data flow multiprocessor.

Software Specific to Processor 1. The intent in developing the data flow processor was to have the processors be totally autonomous. That goal is not totally achievable using the SBC 80/20 series computers. The serial output ports are physically associated to specific processors. Therefore, for all of the systems output to be sent to the console terminal, all output functions had to be consolidated on one processor. Processor 1 has one routine not found in processor 2; that routine is DUMPBUF. It is executed once for each dataflow executed. It checks the output queue and the serial port. If there are characters to be written and the port is ready to receive another character, then it sends the next queued character to the port.

Software Specific to Processor 2. In the actual execution of data flow programs, software in processor 2 is identical to software in processor 1 with the ommission of the DUMPBUF routine described above. To use processor 2, however, it is necessary to assign it using the mailbox assignment scheme described previously. The routine FLOW2, when called by processor 1 from the Pascal host program, assigns processor 2 to the execution of the data flow processor software. FLOW2 also contains the code which allows processor 2 to copy the data flow processor program into its RAM.

Base Data Flow Processor. This is the software that executed the nodes of the flow program. When called by the processor specific software, the routines that comprise the base data flow processor

retrieve the next enabled node from the enabled flow queue and execute it. All memory and table interlocks are performed in this section to assure that, when executing critical sections of code, that all other processors are locked out. Routines that comprise the base data flow processor are:

OUTCHAR- This routine puts one character in the output queue. It performs buss and memory exclusion to prevent interference by other processors, but it does not interlock the queue to prevent other processors from inserting characters between successive OUTCHAR calls.

INTERP- This is the top level procedure of the data flow processor. It takes a pointer to an enabled node and executes it.

All of the routines whose names end in "OP" actually perform the operation indicated. Operations are identical to those of the simulator (Table III).

ADDOP- Performs the "+" operator.

SUBOP- Performs the "-" operator.

MULOP, DIVOP, GTOP, NEOP, INOP- Stubs for node operator types that have not been implemented in this version.

COP, ROP- Perform the "C" and "R" operators. Though the usage of the constant operator and the repeat operator are different, their implementation is identical.

LTOP- Performs the "<" operator.

EQOP- Performs the "=" operator.

MOP- Performs the "M" operator.

OUTOP- Performs the "O" operator.

DISTSAME- This routine distributes outputs where all destination nodes receive the same value. It uses the current nodes output list to distribute the outputs.

DISTCOND- This routine is used to distribute outputs from a conditional. If the conditional is true it sends inputs 3,4,5 .. n+2 to destination nodes 1,3,5 .. n/2 respectively. If the conditional is false it sends the inputs to destinations 2,4,6 .. n/2+1 respectively.

PRWORD- Prints one 16 bit word in hexadecimal in the output queue. Uses PRBYTE. Assumes that the word to be printed is in the HL register pair.

PRBYTE- Prints one 8 bit byte in hexadecimal in the output queue. Uses PRNIBBLE. It assumes that the byte in in the A register.

PRNIBBLE- Prints one four bit hexadecimal digit in the output queue. Assumes that the nibble to be printed is in the low four bits of the A register.

PRMSG- Prints an ASCII message in the output queue. It assumes the message follows the call and is terminated by a null character (0H). It returns control to the instruction following the message.

BLOCKIO- Waits for the output buffer to be free. When the buffer is free, BLOCKIO sets the buffer busy flag to assure that no other processor can gain access to the output buffer.

FREEIO- Releases control of the output buffer.

GETIN- This routine, when called, retrieves the queued value

(if any) of the pointed to input. It updated the pointer to point to the next input. It also maintains a count of inputs that still have queued values; if all inputs have queued values even after removing the needed inputs, then the node may be reenabled at once.

PUTOUT- This routine places a value in the input queue of a node; if the input queue was empty, it decrements the nodes wait count; and if the node's wait count reaches zero PUTOUT enables the node and queues it n the enabled node queue.

POPEQ- Retrieves the pointer to the next enabled node from the enabled node queue.

PUSHEQ- Puts a node in the enabled node queue.

POPIQ- Retrieves a value from an input queue.

PUSHIQ- Places a value in an input queue.

PEEKIQ- Returns the next value (if any) in an input queue, but does not remove the value from the queue.

Memory Organization. To ease implementation of the hardware data flow processor, memory is divided into functional areas. This can be seen in Table VII which shows the allocation of funcional blocks of memory to physical addresses. Note that most of the memory is common to both processors, but there are some areas that belong to one or the other processor alone. Specifically, each processor has its own RAM memory from locations F800H to FFFFH. This is where the machine code of the data flow processor resides. Puting the code into the individual processors' RAM has two major advantages over using the common memory: it minimizes buss traffic, because accesses to internal RAM does not require access to the system buss; and it allows the

TABLE VII

Memory Organization of Hardware Implementation

ADDRESS	PROCESSOR 1	PROCESSOR 2
0-0FFFH	Switchable boot ROM	ROM Program (see Fig 12)
1000H-DD00H	Pascal System	not used
DD00H-DDFFH	Shared Output buffer for both processors	
DE00H-DFFFH	not used	not used
E000H-F7FCH	Shared Data Flow Program Memory	
F7FDH-F7FFH	Shared Processor assignment mailbox	
F800H-FFFFH	Processor 1 Dataflow Processor Code	Processor 2 Dataflow Processor Code

processors' code to occupy the same address space without having to be reentrant code.

Initial Hardware Implementation Results

The processor was demonstrated on a data flow program similar to the looping program of Figure 13 (the only change was to replace multiplies nodes with addition nodes). The processor could be run in either a single or dual processor mode. Execution time for the program could not accurately be measured because the program was largely output bound. But when a debug option was enabled to indicate the processor that executed a particular node, neither processor appeared to be preferred. On subsequent executions of the program, the processor that performed a particular node changed, while the program results remained the same. This indicates that the processors performed the first node that became enabled.

The results described above are not intended to prove the correct operation of the hardware data flow processor, rather, they are intended to show that the basic functions of the processor have been implemented and operate.

Summary

The hardware implementation was intended to provide assurance that development of a data flow processor from existing microprocessors was possible and to estimate the amounts of overhead associated with executing data flow programs. Though it met those initial goals, it is useful only to demonstrate data flow, because its design was not optimized for efficient performance. In Section IV the lengths of some of the processor's routines will be used as an estimate of the execution times for those functions.

V. Results

The results documented in this Section were gathered by executing the simulation in a variety of configurations using parameters estimated from existing conventional processors, and from the hardware implementation. The purpose of this section is to compare the performance of the simulated data flow processor with that of a conventional processor of similar instruction speed. To that end, the following sections will discuss the processor against which the data flow processor will be compared, the execution times of data flow nodes, overheads associated with transfer of data, and the performance of both the data flow processor and the conventional processor in three trial programs.

Comparison of Dataflow against Conventional Processors

A Basis for Comparison. To be able to discuss specific cases, it is necessary to describe the way the tests were run. A hypothetical computer based on the UCSD P-machine was developed as the basis against which the performance of the data flow simulation could be judged. The computers execution times were selected by empiracly determining the execution time of UCSD Pascal running on an 8080 based microcomputer. This was done by executing a simple looping program with the operator whose time was to be measured, performed in the loop. The execution time of the operator is:

$$E = (TO-TL)/R$$

Where:

E = time of the operator
TO= time to execute the loop with the operator
TL= time to execute the loop without the operator
R = the number of loops performed

The execution times of the hypothetical processor is then compared with the simulated results.

The following is a summary of the execution times derived from observing the execution of UCSD Pascal:

```
multiply,divide = 1.900 msec
add,subtract,
all compares    = 0.450  msec
assignment      = 0.350  msec
input,output    = 1.000  msec
```

Execution Times for the Dataflow Processor.

The data flow processor has a slightly different set of instructions than the hypothetical processor. It, for example, has no assignment operation. But, it does have overhead associated with both reading and queueing inputs and outputs. In addition, it has merge and literal operations which have no direct correspondence to a conventional processor. These two operators do nothing more than relay values. Thus almost all of their execution time can be expressed in terms of the input and output overheads associated with them. In each, however, there in still a small amount of overhead associated with decoding the instruction. The following was selected as an estimate of the time to decode the instruction. Execution times follow:

```
merge    = 0.050  msec
constant = 0.050  msec
```

The execution times for all other operations above were chosen to be identical with that of the UCSD Pascal system, with the exception of the assignment operator, because that function is not envoked explicitly in dataflow. This is because each operator implicitly

specifies the destinations of its results. Because the assignment is handled implicitly as part of each operation, there must be an additional parameter in the execution time of a node in addition to the time to perform the operation.

The additional overhead is associated with every queueing and dequeueing of an input. The hardware implementation was used to estimate these overheads. The times to store a value into a queue or to remove a value from a queue were estimated by counting the number of instructions used in that section of the harware implementation, and multiplying by an estimated 8080 instruction execution time of 3 microseconds. Using that technique, the time to input a value into a queue or to remove a value from a queue is 200 microseconds.

Example Program Executions

The following three examples were chosen to be representative of the types of problems that will occur in a laboratory computer system. They were not selected to show all of the features of the data flow processor nor were they intended to be an exhaustive set of possible applications.

Comparison 1: Simple Looping Program. Figures 13,14, and 15 respectively, show this program in Pascal, graphical data flow, and notation form. Basically, the program takes the numbers between 1 and 20 and performs a few simple arithmetic operations on each. In so simple a Pascal program one would tend to think that there is no parallelism possible. That is not the case! The simulation was run on this program with one to five processors; results are shown in Table VIII. Also shown, an estimated execution time for the base

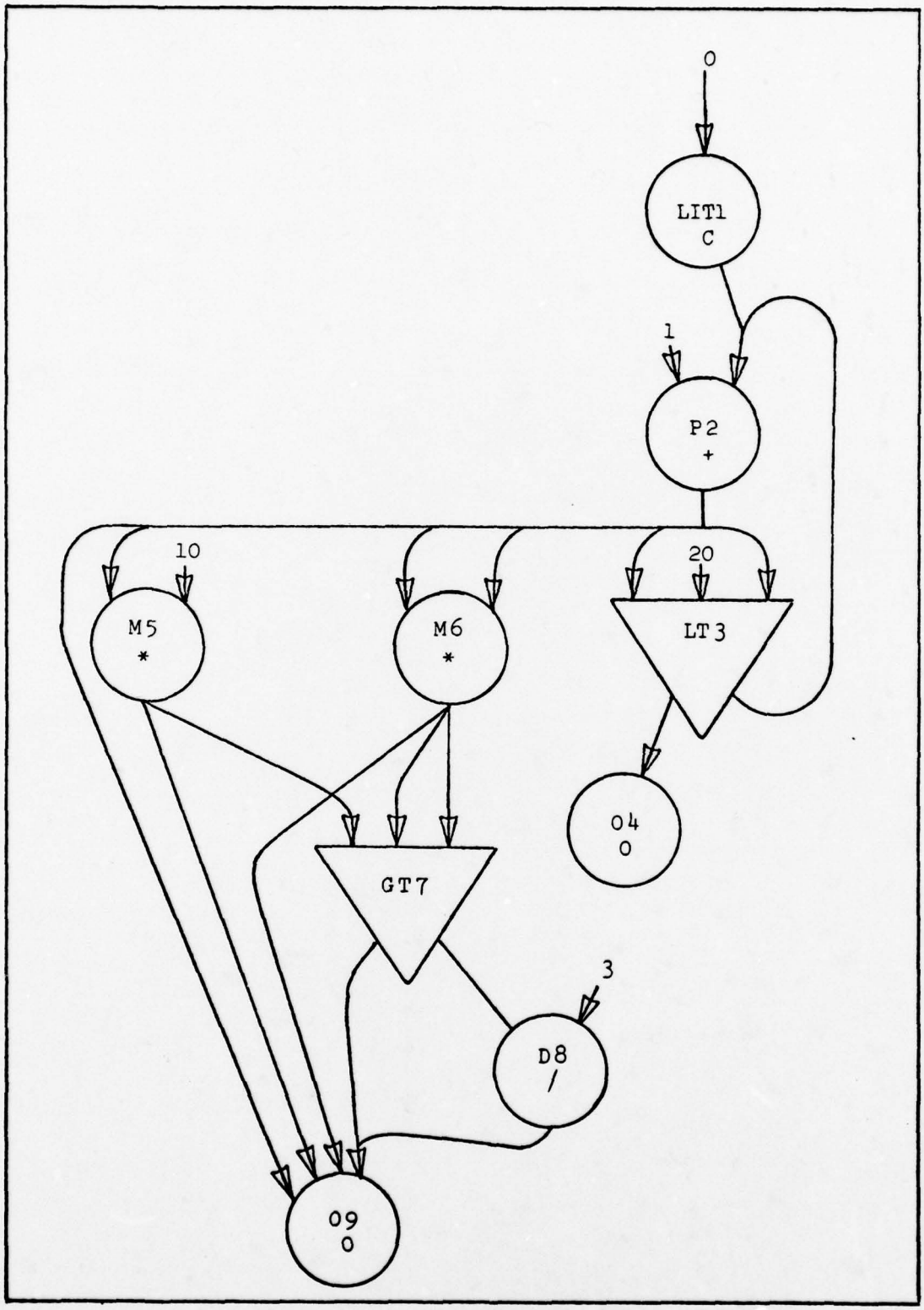


Fig 13. Comparison 1: Loop (Graphical)

```
FOR I := 1 TO 20 DO
        BEGIN
        J := I*10;
        K := I*I;
        IF J < K THEN   L := K DIV 3
                ELSE    L := K;
        WRITELN(I,J,K,L);
        END;
```

Fig 14. Comparison 1: Loop (Pascal)

```
LIT1    C  1 0  1:0             $ P2:2;
P2      +  2 1  1:1             $ LT3:1 LT3:3 M6:1 M6:2 M5:1 09:1;
LT3     <  3 2  2:20            $ P2:2 04:1;
04      0  1 1                  $ ;
M5      *  2 1  2:10            $ GT7:1 09:2;
M6      *  2 2                  $ GT7:2 GT7:3 09:3;
GT7     >  3 3                  $ D8:1  09:4;
D8      /  2 1  2:3             $ 09:4;
09      0  4 4                  $ ;
```

Fig 15. Comparison 1:Loop (Notation)

TABLE VIII

Simulated Execution Times for Loop Program

Number of Processors	Execution Time
5 or more	80.1 msec
4	83.6 msec
3	100.0 msec
2	143.1 msec
1	281.1 msec
Comparable Conventional Processor	189.0 msec

conventional processor developed above. As can be seen the conventional processor outperformed the single processor data flow machine. However, as the number of processors increased the data flow processor surpassed the performance of the conventional processor. Note also, that there is a diminishing return on the number of processors. In this case, although there was some performance increase up to the limit of the number of processors that could be used in the application, when the fourth and fifth processors were added there was negligable improvement.

Comparison 2: Ordinary Differential Equation. This program evaluates a simple ordinary differential equation of the form:

$$\underline{X} = \underline{A} * \underline{X} + \underline{B}$$

The Pascal code and the corresponding data flow notation to perform this function are shown in Figures 16 and 17 respectively. When executed, the results shown in Table IX were produced. Again the estimated time for a conventional single processor is shown for comparison. The single processor was outperformed by the dual processor data flow machine but not by the single processor. Note again, that there is a diminishing return on the number of processors effectively used.

Comparison 3: Equation Evaluation. In this example, processing of a simple fifth order polynomial is compared. Several different methods of connecting the graph were explored. The Pascal at the top of the figures shows the different order of evaluation. In the notation, this is represented by different connections between parts of the graph. Figure 18 A,B,C,D, and E show the equation (parenthesized to show order of evaluation) and the corresponding data flow notation.

```
CONST
        DT =   1;
        K1 =   1;
        K2 =   2;
        B2 =   1;
        M1 =   4;
        M2 =   3;
VAR
        I:INTEGER;
        OFILE:INTERACTIVE;
        TIME,X1,X2,X1D,X2D,X1DD,X2DD:INTEGER;

BEGIN
REWRITE(OFILE,'PRINTER:');

X1 := 5000;
X2 := 4000;
X1D := 0;
X2D := 0;
TIME := 0;

WHILE TIME<  100 DO
        BEGIN
        FOR I := 1 TO 10 DO
                BEGIN
                TIME := TIME + DT;
                X1DD := (-X1*K1-(X1-X2)*K2) DIV M1;
                X2DD := (-(X2-X1)*K2-X2D*B2) DIV M2;
                X1D  := X1D + X1DD*DT DIV 10;
                X2D  := X2D + X2DD*DT DIV 10;
                X1   := X1  + X1D *DT DIV 10;
                X2   := X2  + X2D *DT DIV 10;
                END;

        WRITELN(OFILE,'TIME=',TIME:5,' X1=',X1:5,
                      ' X2=',X2:5,' X1D=',X1D:5,
                      ' X2D=' ,X2D:5,' X1DD=',X1DD:5,
                      ' X2DD=',X2DD:5);
        END;
END.
```

Fig 16. Comparison 2: Differential Equation (Pascal)

```
X1O      R 1 1    $ X1:1 T1:1 M1:1;
X1DO     R 1 1    $ X1D:1;
X2O      R 1 1    $ M1:2 X2:1;
X2DO     R 1 1    $ T2:1 X2D:1;
T1       * 2 1 2:-1       $ P1:1;
P1       + 2 2            $ X1DD:1;
T4       * 2 1 2:1        $ DIV101:1;
T6       * 2 1 2:1        $ DIV102:1;
M1       - 2 2            $ T3:1;
T3       * 2 1 2:-2       $ P1:2 M2:2;
T2       * 2 1 2:-1       $ M2:1;
M2       - 2 2            $ X2DD:1;
T5       * 2 1 2:1        $ DIV103:1;
T7       * 2 1 2:1        $ DIV104:1;
DIV101   / 2 1 2:10       $ X1D:2;
DIV102   / 2 1 2:10       $ X1:2;
DIV103   / 2 1 2:10       $ X2D:2;
DIV104   / 2 1 2:10       $ X2:2;
X1DD     / 2 1 2:4        $ T4:1  IF1:7;
X1D      + 2 2            $ T6:1  IF1:6;
X1       + 2 2            $       IF1:5;
X2DD     / 2 1 2:3        $ T5:1  IF1:10;
X2D      + 2 2            $ T7:1  IF1:9;
X2       + 2 2            $       IF1:8;
TO       + 2 1 2:1        $       IF1:3;
IO       + 2 1 2:1        $       IF1:1 IF1:4;

INIT     R 1 0 1:0        $ TO:1 IO:1 X1DO:1 X2DO:1;
INIT1    C 2 0 1:5000 2:4000 $ X1O:1 X2O:1;
K        0 0 20000 $ ;

IF1      < 10 9 2:10      $ TO:1 RTO:1 IO:1 K:0 X1O:1
                RX1O:1 X1DO:1 RX1DO:1 K:0 RX1DDO:1
                X2O:1 RX2O:1 X2DO:1 RX2DO:1 K:0 RX2DDO:1;

IF2      < 8 6 2:100 4:0 $ TO:1 K:0 IO:1 K:0 X1O:1
                K:0 X1DO:1 K:0 X2O:1 K:0 X2DO:1 K:0;

RTO      R 1 1            $ IF2:1 IF2:3 OUT:1;
RX1O     R 1 1            $ IF2:5       OUT:2;
RX1DO    R 1 1            $ IF2:6       OUT:3;
RX1DDO   R 1 1            $             OUT:4;
RX2O     R 1 1            $ IF2:7       OUT:5;
RX2DO    R 1 1            $ IF2:8       OUT:6;
RX2DDO   R 1 1            $             OUT:7;

OUT      0 7 7 $ ;
```

Fig 17. Comparison 2: Differential Equation (Notation)

TABLE IX

Simulated Execution Times for ODE Program

Number of Processors	Execution Time
7 or more	2705.8 msec
6	2705.8 msec
5	2707.8 msec
4	2735.0 msec
3	2797.0 msec
2	3005.8 msec
1	5257.8 msec
Comparable Conventional Processor	3406.2 msec

```
Pascal:

      X2 := X*X;
      X3 := X*X2;
      X4 := X2*X2;
      X5 := X*X4;
      ANS := ((((X*A)+(X2*B))+(X3*C))+((X4*D)+(X5*E)));

Notation:

INIT     C  7 0  1:3 2:3 3:-2 4:2 5:-5 6:1 7:3 $
         X:1 S01:1 A1X:2 A2X2:2 A3X3:2 A4X4:2 A5X5:2;
X        R  1 1          $ X2:1 X2:2 X3:2 X5:2 A1X:1 OUT:1;
X2       *  2 2          $ X3:1 X4:1 X4:2      A2X2:1;
X3       *  2 2          $                     A3X3:1;
X4       *  2 2          $ X5:1                A4X4:1;
X5       *  2 2          $                     A5X5:1;

A1X      *  2 2          $ S01:2;
A2X2     *  2 2          $ S012:2;
A3X3     *  2 2          $ S0123:2;
A4X4     *  2 2          $ S45 1;
A5X5     *  2 2          $ S45 2;

S01      +  2 2          $ S012:1;
S012     +  2 2          $ S0123:1;
S0123    +  2 2          $ S012345:1;
S45      +  2 2          $ S012345:2;
S012345  +  2 2          $ OUT:2;

OUT      0  2 2          $ ;
```

Fig 18(A). Comparison 3: Expression Evaluation (Order of Evaluation #1)

```
Pascal:

      X2 := X*X;
      X3 := X*X2;
      X4 := X2*X2;
      X5 := X*X4;
      ANS := ((((((X*A)+(X2*B))+(X3*C))+(X4*D))+(X5*E)));

Notation:


INIT     C  7 0  1:3 2:3 3:-2 4:2 5:-5 6:1 7:3 $
         X:1 S01:1 A1X:2 A2X2:2 A3X3:2 A4X4:2 A5X5:2;
X        R  1 1           $ X2:1 X2:2 X3:2 X5:2 A1X:1 OUT:1;
X2       *  2 2           $ X3:1 X4:1 X4:2      A2X2:1;
X3       *  2 2           $                     A3X3:1;
X4       *  2 2           $ X5:1                A4X4:1;
X5       *  2 2           $                     A5X5:1;

A1X      *  2 2           $ S01:2;
A2X2     *  2 2           $ S012:2;
A3X3     *  2 2           $ S0123:2;
A4X4     *  2 2           $ S01234:2;
A5X5     *  2 2           $ S012345:2;

S01      +  2 2           $ S012:1;
S012     +  2 2           $ S0123:1;
S0123    +  2 2           $ S01234:1;
S01234   +  2 2           $ S012345:1;
S012345  +  2 2           $ OUT:2;

OUT      0  2 2           $ ;
```

Fig 18(B). Comparison 3: Expression Evaluation (Order of Evaluation #2)

```
Pascal:

      X2 := X*X;
      X3 := X*X2;
      X4 := X*X3;
      X5 := X*X4;
      ANS := ((((X*A)+(X2*B))+(X3*C))+((X4*D)+(X5*E)));

Notation:

INIT     C  7 0  1:3 2:3 3:-2 4:2 5:-5 6:1 7:3 $
         X:1 S01:1 A1X:2 A2X2:2 A3X3:2 A4X4:2 A5X5:2;
X        R  1 1          $ X2:1 X2:2 X3:2 X4:2 X5:2 A1X:1 OUT:1;
X2       *  2 2          $ X3:1                A2X2:1;
X3       *  2 2          $ X4:1                A3X3:1;
X4       *  2 2          $ X5:1                A4X4:1;
X5       *  2 2          $                     A5X5:1;

A1X      *  2 2          $ S01:2;
A2X2     *  2 2          $ S012:2;
A3X3     *  2 2          $ S0123:2;
A4X4     *  2 2          $ S45:1;
A5X5     *  2 2          $ S45:2;

S01      +  2 2          $ S012:1;
S012     +  2 2          $ S0123:1;
S0123    +  2 2          $ S012345:1;
S45      +  2 2          $ S012345:2;
S012345  +  2 2          $ OUT:2;

OUT      O  2 2          $ ;
```

Fig 18(C). Comparison 3: Expression Evaluation
(Order of Evaluation #3)

```
Pascal:

      X2 := X*X;
      X3 := X*X2;
      X4 := X*X3;
      X5 := X*X4;
      ANS := ((((((X*A)+(X2*B))+(X3*C))+(X4*D))+(X5*E)));

Notation:

INIT     C  7 0  1:3 2:3 3:-2 4:2 5:-5 6:1 7:3 $
         X:1 S01:1 A1X:2 A2X2:2 A3X3:2 A4X4:2 A5X5:2;
X        R  1 1          $ X2:1 X2:2 X3:2 X4:2 X5:2 A1X:1 OUT:1;
X2       *  2 2          $ X3:1                 A2X2:1;
X3       *  2 2          $ X4:1                 A3X3:1;
X4       *  2 2          $ X5:1                 A4X4:1;
X5       *  2 2          $                      A5X5:1;

A1X      *  2 2          $ S01:2;
A2X2     *  2 2          $ S012:2;
A3X3     *  2 2          $ S0123:2;
A4X4     *  2 2          $ S01234:2;
A5X5     *  2 2          $ S012345:2;

S01      +  2 2          $ S012:1;
S012     +  2 2          $ S0123:1;
S0123    +  2 2          $ S01234:1;
S01234   +  2 2          $ S012345:1;
S012345  +  2 2          $ OUT:2;

OUT      O  2 2          $ ;
```

Fig 18(D). Comparison 3: Expression Evaluation
(Order of Evaluation #4)

```
Pascal:

      X2 := X*X;
      X3 := X*X2;
      X4 := X2*X2;
      X5 := X2*X3;
      ANS := ((((((X*A)+(X2*B))+(X3*C))+(X4*D))+(X5*E)));

Notation:

INIT     C  7 0  1:3 2:3 3:-2 4:2 5:-5 6:1 7:3 $
         X:1 S01:1 A1X:2 A2X2:2 A3X3:2 A4X4:2 A5X5:2;
X        R  1 1           $ X2:1 X2:2 X3:2       A1X:1 OUT:1;
X2       *  2 2           $ X3:1 X4:1 X4:2 X5:1 A2X2:1;
X3       *  2 2           $ X5:2                 A3X3:1;
X4       *  2 2           $                      A4X4:1;
X5       *  2 2           $                      A5X5:1;

A1X      *  2 2           $ S01:2;
A2X2     *  2 2           $ S012:2;
A3X3     *  2 2           $ S0123:2;
A4X4     *  2 2           $ S01234:2;
A5X5     *  2 2           $ S012345:2;

S01      +  2 2           $ S012:1;
S012     +  2 2           $ S0123:1;
S0123    +  2 2           $ S01234:1;
S01234   +  2 2           $ S012345:1;
S012345  +  2 2           $ OUT:2;

OUT      O  2 2           $ ;
```

Fig 18(E). Comparison 3: Expression Evaluation
(Order of Evaluation #5)

Table X shows the execution times for each of the connectivities and also the performance of a conventional processor.

The order of execution has a distinct effect on the execution times of the multiprocessor data flow processors. As can be seen there is a 20 percent difference in processing times between the fastest and slowest of the four processor configurations. In fact, cases C and D could not use all four processors, the maximum parallelism achieved for them was three processors.

In this example, the conventional processor was considerably more efficient than the single or double processor data flow processor. This can be attributed largely to startup and finish delays in the data flow processor. This effect is quite similar to inefficiencies caused when pipeline processors are starting up or shutting down. Consider another case of Figure 18(B), executed four times in succession, with all input values present at the start of execution. The execution times are shown in Table XI. Here the data flow processor can begin to increase the multiprocessing level. The maximum number of processors used at any time was 12.

Parallelism Achieved

Theoretical Parallelism. During execution of a data flow program, the simulator keeps a record of the maximum number of processors used at one time, by giving the simulated processor a large number of processors, the data flow program will use as many processors as it can. After execution, the maximum number of processors used will be determined by the data flow program. In each of the examples previously discussed, the highest number of processors shown on the

TABLE X

Simulated Execution Times for Expression Evaluation

Number of Processors	Execution time (in msec) A	B	C	D	E
4	18.5	17.5	N/A	N/A	17.5
3	18.5	17.9	21.0	20.0	17.9
2	21.5	20.5	22.1	21.0	20.6
1	34.2	34.2	34.2	34.2	34.2

Comparable Conventional Processor 20.7 msec

TABLE XI

Simulated Execution Times for Expression Evaluation *with Multiple Inputs*

Number of Processors	Execution Time
12 or more	27.6 msec
11	27.6 msec
10	27.8 msec
9	28.2 msec
8	28.9 msec
7	30.0 msec
6	32.1 msec
5	35.0 msec
4	40.9 msec
3	50.5 msec
2	71.6 msec
1	138.4 msec

Comparable Conventional Processor 80.0 msec

execution table is the maximum number of processors usable by the program. The examples show that many processors can be brought to bear on what would normally be considered a single processing task. In the case of the multiple evaluation of the equation, as many as 12 processors were working at one time.

Practical Parallelism. This nebulous term shall be defined as the actual effect of parallelism during execution of a program. Practical parallelism of one would mean executing a program in the same time as the hypothetical processor. Parallelism of three would mean execution in one third the time. To allow a meaningful comparison, the unit of comparison shall be the the hypothetical processor defined above.

From the examples, (Tables VIII, IX, X, and XI) it can be seen that though the maximum number of processors usable was generally very high (5 to 12 processors), the effective parallelism in the cases shown was relatively low (0.58 to 2.9). There appear to be two causes for this: first, the algorithms often did not allow a significant amount of actual parallel execution; and second, there is significant overhead associated with queueing and dequeueing information in the data flow system.

The algorithms used in the examples were selected to represent typical types of operations carried out within a laboratory computer. One of the prime constraints in this investigation is that it consider a typical laboratory workload. Therefore, the inability of the processor to achieve high parallelism appears to represent a potential problem with the processor, not the algorithms tested.

The overhead caused by the queueing and dequeueing values does

significantly effect the performance of the processor. In the examples, the data flow processor was generally slower than the conventional processor until two or three processors was added. Table XII shows the dataflow processor execution times for comparison 2, ODE evaluation, with the assumption that there is no overhead for queueing and dequeueing. The table also has a percentage column indicating the percentage of increase achieved by eliminating the overhead. By reducing the overhead of the data flow processor, a single processor data flow processor has similar execution time to a conventional processor. The addition of the second and third processor immediately give the data flow processor faster execution time.

In the examples tested, the net performance increase as more than four processors were added was not significant. Reducing the overhead allowed the second and third processors to net a significant performance increase over the conventional processor, rather than merely catching up.

Cost Effectiveness of Data Flow

The cost effectiveness of this approach cannot accurately be determined. In the hardware implementation, the cost of a two processor system was approximately twelve percent higher than a single processor system. On the other hand, there are a significant number of intangables that must be considered:

Code density- Data flow notation takes signifigantly more of the machine memory to store than a corresponding conventional processor instruction.

Interpreter size- The interpreter of the hardware implementation

TABLE XII

Simulated Execution Times for ODE Program
(no overhead)

Number of Processors	Execution time	Improvement
7 or more	1397.0 msec	48.4%
6	1397.0 msec	48.4%
5	1400.0 msec	48.3%
4	1440.5 msec	47.3%
3	1442.3 msec	48.4%
2	1587.7 msec	47.2%
1	2962.0 msec	43.7%

was relatively small (less than one thousand bytes) but the implementation did not include many functions necessary for a useable (rather than a demonstration) system. For example: floating point operations, procedure calls, structures, input-output other than simple terminal operations were not installed. If those functions were added the interpreter size would increase dramatically.

Data flow startup/shutdown delays- As seen previously (comparison 3), a data flow processor does best under circumstances where values may be pipelined through a set of calculations. Many tasks are not structured well for such an environment. For example, iterative operations have limited parallelism, if the loop is short, as can be seen in example 1 (Loop), and short independent calculations such as example 3 (Equation evaluation) spend the majority of time starting up and shutting down. These problems can be minimized by grouping together a set of equations which may be executed in parallel, or by evaluating the same equation for multiple inputs at the same time (example 3 multiple inputs).

Difficulty with structures- Many tasks need to use data structures such as tables. Data flow processors cannot efficiently operate on such structures. Other than the potential solution described in Section II, specifically: single use structures, there does not appear to be any practical way to deal with structures.

Interprocessor interferance- Using two 8080 processors sharing an Intel Multibus there was virtually no interferance. This

can be concluded because the buss access time for a memory access is approximately 500ns, while the processors are accessing the buss only rarely (Remember that the code for the data flow processors is executing from on board memory) to fetch the next data flow node or data value. Assuming one operation on shared memory every 15 instructions, each processor is imposing a 1% load on the buss. As the number of processors increase, however, there will be increased buss contention. This contention will slowly result in decreased benefit from additional processors.

The intangables just discussed make it impossible to quantitatively assess the cost effectiveness of the data flow processor at this time. Qualitatively, however, it appears that a data flow processor constructed using an existing microcomputer does not seem to be practical. Because of the high overheads associated with queueing and dequeueing values and maintaining the data flwo structures, the data flow microprocessor is significantly slower than its conventional counterpart. This results in the amount of processing speed increase to be expected per additional processor is minimal. While the cost of the system rises as the number of processors is increased.

Rather than using a conventional microprocessor, a special purpose processor that performs the data flow operations in microcode would very likely be quite practical. The concept of generating a processor for a specialized application is not new. Western Digital Corporation recently released a microcoded three chip processor that directly implements UCSD P-code. The savings in execution time from an interpreted P-machine to the Microengine is reported by Western Digital

as being approximately a factor of 20. If such a technique were used to implement a data flow processor, most of the overheads associated with data flow processing could be eliminated.

VI. Conclusions and Recommendations

Background

The purpose of this effort was investigation of data flow techniques as they apply to the single user multiprocessor system. The goals were to assess the practicality of developing a single user data flow processor using current technology. Factors bearing on this goal are cost of the system, flexibility, and ease of use.

Today, there is much interest in generating a machine that can directly execute a data flow representation of a problem. Because this approach saves the need to approximate data flow parallelism on conventional computers, such a system would theoretically be capable of many times the throughput of conventional systems.

This report described an effort to determine the usefulness of a small single user data flow processor. The constraints placed on the system were that it be reconfigurable, non-realtime, small, single user and operate in the laboratory environment. An adaptable simulator to model dataflow processors was developed, then a hardware implementation of a data flow processor was developed to provide insight into the functions needed in a data flow processor. It also provided a means of determining the delays occurring when an input or output is transmitted from node to node.

Summary of Important Findings

The simulator was found to be a useful tool for evaluating the use of data flow in a computer system. It allowed the parameters of the data flow processor to be easily varied. This flexibility made it

possible to determine the number of processors that could be used by a particular algorithm, while it assessed the effects of varying the execution and overhead times.

The hardware implementation showed that a dual microprocessor can be used to construct a data flow processor. It was also useful as a means of determining the amount of overhead to be expected in data flow execution. It was not intended to be a practical data flow processor. As such, the design was not intended to optimize performance, rather, it was intended to permit evaluation of overhead and delays resulting from queueing and dequeueing of data.

Using the overhead values derived from the hardware implementation and as set of sample execution rates, derived from a conventional processor, three comparisons between data flow processing and conventional processors were performed using the simulator. A data flow single-processor was significantly slower than a conventional processor with the same instruction execution times. When a second processor was added to the data flow processor, it became more competative. Additional processors produced more improvement, however, there was a limit to this improvement, where additional processors produced little or no improvement.

Conclusions

The initial intent of this investigation was to produce a quantitative analysis of data flow performance versus that of a conventional processor. This could not be totally accomplished. The need to develop tools by which an arbitrary data flow processor could be tested (the simulator) and the need to gain actual experience in the

overhead that is incurred in the execution of a data flow program (the hardware implementation) occupied a large portion of the time allocated.

A qualitative assessment of data flow performance in that environment was, however, completed. The results tend to show that building data flow processors using current microprocessor architectures is impractical. The overheads associated with manipulating the queues of data and enabled nodes is simply too great to allow practical competition with conventional processor architectures. For example, a data flow multiprocessor using two to three microprocessors would be necessary simply to be of comparable speed to a conventional processor.

One reason for this is that a large portion of the time to execute a node is spent performing overhead functions such as queue management and distribution of outputs. Making a data flow processor more efficient could be accomplished in either of two ways:

Increase the proportion of time performing operations.

If the time spent executing the function of the node is increased by increasing the complexity of the node's function, the processor will spend a larger part of its time doing work rather than overhead. This can be accomplished by raising the complexity of the functions at the nodes. For example, a processor whose nodes could perform trigonometric and hyperbolic functions rather than simple arithmetic functions could decrease the number of nodes used in some functions. This would in turn decrease the amount of overhead (overhead is related to the number of nodes and their

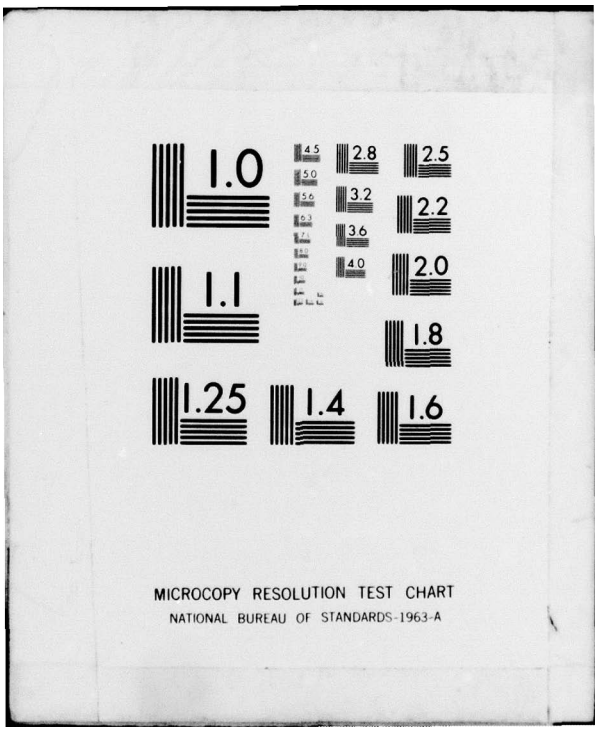
outputs) and increase the program's efficiency. Such functions could be implemented in software if the data flow processor is interpreted (as was the hardware implementation discussed in Section IV) or in microcode (by adding the additional functions to the processors instruction set).

Directly decrease the overhead needed to perform a node. Current microprocessors have no intrinsic facilities for queue management, table exclusion or distribution of the outputs as is needed in data flow processing. If a special microprocessor chip or chip set could be built to directly execute data flow (perform the operations in hardware or in microcode), the overhead could be greatly reduced. Such a processor would have built-in capabilities to interface with other processors, and it would automatically exclude other processors from critical areas of the system when necessary. It would also have the capability to easily enter or remove elements of a queue. In Section IV, the performance of a data flow processor without overheads was discussed. A special purpose dataflow processor would clearly fall closer in performance to those figures than to the figures of a dataflow processor with overhead. The concept of generating a processor for a specialized application is not new. Western Digital Corporation recently released a microcoded three chip processor that directly implements UCSD P-code. The savings in execution time from an interpreted P-machine to the Microengine is reported by Western Digital as being

AD-A080 174 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
DATA FLOW TECHNIQUES IN SINGLE-USER MULTIPROCESSOR SYSTEMS.(U)
DEC 79 B P BOESCH
UNCLASSIFIED AFIT/GE/EE/79-7 NL

2 OF 2
AD
A080174

END
DATE
FILMED
3-80
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

approximately a factor of 20. If such a technique were used to implement a data flow processor, most of the overheads associated with data flow processing could be eliminated.

Recommendations

This area has a significant potential for further study. Three areas seem most promising: development of features to allow easy programming in dataflow notation, further enhancements to the simulator, and design of a microprocessor architecture to directly execute dataflow notation.

Language Development. Development of data flow programs is limited by the programmer's ability to express graphical information. There appear to be two approaches possible to simplifying the programmer's task: first, an interactive graphical programming system that would allow the programmer to express any well formed data flow program visually on a screen; or secondly, to develop a translator from a block structured language, such as Pascal, to a data flow program.

Further Enhancements to the Simulator. To make the simulation more realistic the following features must be added:

Change Conditional Branching. Using the conditionals as defined in this report, determinate programs are difficult to write. If the conditionals and merge operator as defined by Denning (Ref 9:93) were implemented, data flow programs could easily be assured deterministic.

More Data Types. Real, string and general scalar types should be added.

Procedures. Use of procedures to simplify programs has been

used in virtually all programming languages. Rumbaugh (Ref 21) describes the features of a data flow processor with the capability to use recursive procedures.

Data Structures. Because data structures are difficult to manipulate within a data flow machine this would be very useful. It would give the system designer the ability to test different structure manipulation techniques.

Input/Output. The simulator, as implemented, has a very simple input/output capability. It can query the user for a value, and it can output a set of values to the user. A system of file I/O would make evaluation of programs that access stored data possible.

Design of a Data Flow Microprocessor. As discussed previously, a special purpose data flow microprocessor could be the most practical method of implementing a data flow processor. To be practical, such a microprocessor should be capable of manipulating queues directly, rapidly distributing outputs, and be capable of interfacing with several other processors.

Bibliography

1. Ackerman, William B. and Dennis, Jack B. "Val- A Value Oriented Algorithmic Language: Preliminary Reference Manual", Massachusetts: Laboratory for Computer Science, Massachusetts Institute of Technology, June 1979

2. Adams, George and Rolander, Thomas "Design Motivations for Multiple Processor Microcomputer Systems", Computer Design, March 1978

3. Aho, Alphred V. and Ullman, Jeffrey D. Principles of Compiler Design, Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley, March 1978

4. Attwood, William J. "Concurrency in Operating Systems", Computer, October 1976

5. Bell, C. and Newell, Allen Computer Structures: Readings and Examples, McGraw-Hill Books Inc., New York, 1971

6. Boesch, Brian P. "Installation of CPM on an Intel SBC 80/20", Ohio: Laboratory Report for EE687, Air Force Institute of Technology, Wright-Patterson AFB, 1979

7. Burskey, Dave "Microprocessor Data Manual", Electronics Design, Vol. 27, No. 24, November 22, 1979

8. Coffman, Edward G. and Denning, Peter J. Operating Systems Theory, Prentice-Hall Inc. Englewood Cliffs, New Jersey, 1973

9. Denning, Peter J. "Operating Systems Principles for Data Flow Networks", Computer, July 1978

10. Dennis, Jack B. "First Version of a Data Flow Procedure Language", Massachusetts: Laboratory for Computer Science, Massachusetts Institute of Technology, May 1975

11. Dennis, Jack B.; Misunas, David P.; and Leung, Clement K. "A Highly Parallel Processor Using a Data Flow Machine Language", Massachusetts: Laboratory for Computer Science, Massachusetts Institute of Technology, January 1977

12. DeMarco, Tom Structured Analysis and System Specification, Yourdon inc., 1133 Avenue of the Americas, New York, New York 10036, 1978

13. El-Dessouki, Ossama; Hochsprung, Ronald; Green, Peter; and Huen, Wing "A Network Computer for Distributed Processing", Illinois: Illinois Institute of Technology, September 1977

14. Ferrari, Domenico Computer Systems Performance Evaluation, Prentice-Hall Inc. Englewood Cliffs, New Jersey, 1978

15. "Graphical and Functional Models of Parallel Computation", Ohio: Class notes EE 750, Air Force Institute of Technology, Wright-Patterson AFB, 1979

16. Hansen, Per Brinch "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, Vol SE-1, No. 2, June 1975

17. Liebowitz, Burt H. "Multiple Processor Minicomputer Systems- Part 2: Implementation", Computer Design, Vol. 17, No. 11, November 1978

18. Mott, D. R. and Arabadjis, G. "Multimicroprocessors with Queue Memories", New York: GE Heavy Military Equipment Department, Syracuse, New York

19. Peterson, James L. "Petri Nets", Computing Surveys Vol. 9, No. 3, September 1977

20. Rumbaugh, James E. "A Data Flow Multiprocessor", IEEE Transactions on Computers, Vol. C-26, No. 2, February 1977

21. Rumbaugh, James E. "Parallel Asynchronous Computer Architecture for Data Flow Processors", Massachusetts: Project MAC, TR-150, Massachusetts Institute of Technology, May 1975

22. Street, Robert T. "Micro-Multi-Processing", Mini-Micro Systems May 1979

23. UCSD (Mini-Micro Computer) Pascal Version II.0 Institute for Information Systems, UCSD Mailcode C-021 La Jolla, Ca. 92093 March 1979

24. Weng, K. "Stream Oriented Computation in Recursive Dataflow Schemas", Massachusetts: Project MAC, TM-68, Massachusetts Institute of Technology, October 1975

25. Wirth, Niklaus and Jensen, Kathleen Pascal Users Manual and Report, Springer-Verlang, New York, New York, 1974

Vita

Brian P. Boesch was born 12 February 1952 in Niagara Falls, New York. He attended Purdue University, West Lafayette, Indiana, where he received a Bachelor of Science Electrical Engineering and was commissioned a Second Lieutennant in the US Air Force. In 1974, he was assigned to Electronic Systems Division (ESD) of Air Force Systems Command. At ESD, he served as an independent consultant to program offices purchasing large computer software and hardware systems. In May of 1978, he was assigned to Air Force Institute of Technology to receive a Masters of Electrical Engineering.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GE/EE/79-7	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Data Flow Techniques in Single-User Multiprocessor Systems		5. TYPE OF REPORT & PERIOD COVERED MS Thesis 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Brian P. Boesch CAPT, USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT-EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE December, 1979 13. NUMBER OF PAGES 100
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

J. P. Hipps, Major, USAF
Director of Public Affairs

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Multiprocessor
Dataflow
Microprocessor
Microcomputer

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Problems associated with performing computation tasks in single-user microprocessor environments are discussed. Data flow methods are investigated as a means of applying multiprocessors to this environment. An event driven simulation of a variable architecture data flow processor is developed, using UCSD Pascal(TM). A trial data flow multiprocessor using 8080 based microprocessors is developed and tested. Using the simulator and information gathered from the trial multiprocessor, several situations typical to single user

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

systems were tested. Maximal levels of parallel processing are presented for a variety of situations.