DATAFLOW COMPUTER ARCHITECTURE: RESEARCH AND GOALS

by

Arvind
Kim P. Gostelow

Technical Report #113

Department of Information and Computer Science
University of California, Irvine
February 6, 1978

*This work is supported by NSF Grant UCI Dataflow Computer Architecture Project MCS76-12460.

Z
699
C3
no. 113

Dataflow Computer Architecture

ABSTRACT

The primary objective of the proposed research is to define and evaluate an architecture for a computer system comprising large numbers of small processors, thereby taking new advantage of LSI technology. Our premise, however, is that such a machine cannot be successful if based upon the usual von Neumann concepts of sequential control and the memory cell. Instead, we adopt the principles of dataflow as a more appropriate semantic base, since dataflow places no constraints on the order of execution other than the arrival of data. Such a basis appears very well suited to a technology disposed towards distributed processing.

We have developed so far both a high-level dataflow programming language and a base machine language into which programs are compiled for execution. We also have devised an interpreter for the base language capable of more highly asynchronous operation than other dataflow systems. It is the emulation of this interpreter that is the goal of the machine proposed here.

We feel that success in this work would be significant since such machines might not only speed the execution of programs, but may also allow the development and application of significant reliability and fail-soft techniques.

A second objective of the proposed work is to collect the experiences of others in using the particular high-level dataflow language we have devised, to improve it, and to determine how well people are able to program in dataflow.

I. The Objective, Significance, and Research Methods to be Employed

1. Objective

The primary objective of the proposed research is to continue the definition and evaluation of an effective architecture for a general-purpose dataflow computer composed of large numbers (hundreds or perhaps even thousands) of small LSI processors, thereby taking new advantage of LSI technology. We also propose to study associated system problems with the eventual goal being a specification of a complete dataflow computer system.

2. Basis and Significance of the Proposed Work

2.1 Basis

Many other proposals (e.g., the Holland machine, Illiac IV, Hydra, Hypercube) have been forwarded in an attempt to synthesize a single large machine from several small processors which cooperate in a concerted effort on a single computation. Almost all of these proposals (some exceptions are [Chamberlin-71, Dennis & Misunas-74, Rumbaugh-77, Sonnenburg & Irani-74]) have failed to recognize that such a machine, only recently made possible by new advances in technology, must itself be founded upon new principles of computation. This is the fundamental premise upon which this proposal stands. We claim that the problems involved in utilizing the new technology are not related simply to providing a proper interconnection mechanism, or to designing a machine which, for example, can efficiently manipulate arrays. Rather, the problems are due to one of the fundamental premises of computer design: the von Neumann model of computation and its basic principles of a (centralized) sequential control and direct programmer manipulation of memory cells. In place of the von Neumann model, we have adopted the opposing principles of <u>dataflow</u> [Dennis-73, Karp & Miller-66, Rodriquez-69, Bährs-72]:

1. Operations execute when and only when the operands required become available (asynchrony).

2. Computation is based on the values produced rather than on where those values are kept (functionality, i.e., the absence of side-effects).

By adopting these principles, we can realize the asynchronous execution of programs without the need for parallel programming constructs (e.g., <u>parbegin</u> - <u>parend</u>) or program analysis of any kind.

Our approach has been to develop simultaneously a base machine language and a higher-level dataflow programming language called ID (for <u>I</u>rvine <u>D</u>ataflow) [Arvind, Gostelow, & Plouffe-76]. Operationally, programs are written only in ID. They are then compiled into the base language (Section II - Semantic Basis of the Machine, and Appendix) and executed on a dataflow computer (Section III - The Proposed Machine). The proposed machine emulates an interpreter [Arvind & Gostelow-77a] capable of far greater concurrency of execution than has been possible in other dataflow systems. It is the definition and evaluation of an architecture for this machine that is described in Section III and which is the primary subject of this proposal. As a final note on our approach, we feel we have now developed a clean semantic basis by simultaneous development of both the higher-level and base dataflow languages, and as opposed to most other approaches to machine design, realization of the base language interpreter has become the goal of the architecture, rather than just a resulting effect.

2.2 Significance

Even though LSI technology has made available to us tremendous capacity for computation, no one has yet been able to realize this potential within a single machine. The reasons, we feel, are inherent in the usually unquestioned principles of the von Neumann approach. Dataflow provides a fundamentally distinct direction for development. One significant result of developing such a machine can be seen by studying the time-complexity of the programs it executes. In Section II we show, for example, an algorithm that requires $O(n^3)$ time on a sequential machine, but may execute as fast as $O(n)$ time on the dataflow machine (as long as $O(n^2)$ processors are available).

Also, a particular advantage of dataflow is its inherently functional nature due to the absence of side-effects, and a correspondingly modular structure [Friedman & Wise-76]. Such properties are of interest in many areas of computer science. For example, much of the movement towards structured programming can be viewed as a drive towards a more functional and less procedural semantics. Program verification and proof of formal properties also appear less complex when only functional modules are involved [Guttag-77, Arvind & Gostelow-77b, Ashcroft & Wadge-77]. We feel such points are important, even though they are not well understood.

Finally, we expect that a machine with large numbers of small asynchronous processors will be well-equipped to provide new and more sophisticated error handling, reliability, and protection capability than might be possible in a more conventional system. Such would be due in part to the dataflow basis, the existence of a pool of similar units, and the incorporation of these features into the lowest machine levels where the many processors can provide the capacity to absorb the overhead such facilities would require. These points are expanded upon in Section III - The Proposed Machine.

3. Method

Our primary objective is to devise an effective architecture for a machine comprising large numbers of small processors. To do this we noted that a new semantic basis was necessary, and that dataflow, we felt, could provide that basis. Towards that end we have developed the high-level dataflow language ID and a low-level base machine language into which ID programs are compiled. (Some examples appear in the following section.) The language allows for highly asynchronous execution by automatically unfolding loops and by allowing simultaneous execution of distinct invocations of the same operation. Here an operation may be any function, for example, an addition, a function application, or even a loop. The primary manifestation of such unfolding and simultaneous operation invocation is that even simple programs can make demands for large numbers of small tasks. These tasks would allow a machine, with sufficient and properly controlled processor resources, to allocate space (the processors) rather than time for program execution. During the research period we propose to successively refine and evaluate an architecture beginning with a design presented in Section III. We propose to evaluate each refinement by incorporating it into a simulator* and making performance measurements on a set of real programs**. We hope to determine, for example, what a good interprocessor communication system would be. Currently we are using a simple but flexible ring bus. Would a Wittie [Wittie-76] or some other system be preferrable? Or perhaps a Pierce-ring [Pierce-72] system? Also, scheduling of tasks is important. How might this be done? These and other questions are discussed later in this proposal.

We also propose to investigate the ability of people to program in dataflow by considering the experiences of both undergraduate and graduate students who will be writing ID programs. We hope to determine the basic ability of people to think in dataflow, and the suitability of ID in both its syntax and semantics. Finally, we plan to modify the language (or to reject it altogether) based in part on these experiences.

----

*The simulator is currently 3000 lines of SIMULA code and has been operational for about 6 months on the campus computing facility PDP-10.

**We have a compiler that translates ID programs into the base language for direct input into the simulator.

II. Semantic Basis of the Machine

1. Introduction

In this section we give an example of an ID program along with its translation into the base language. We also discuss the execution of this program on an ideal dataflow computer. We hope to demonstrate that it is possible to program in a high-level dataflow language, and that ID programs are capable of generating demands for large numbers of processors. It should be stated explicitly that the semantics of an ID program are defined by the base language translation. Programs have the same meaning if they produce the same compilation.

We wish to emphasize here that ID is a complete programming language and includes facilities for resource handling via dataflow monitors [Arvind, Gostelow, & Plouffe-77] and for programming with streams; ID is also extensible and incorporates programmer-defined data types. Because of limited space, only some of the fundamental programming constructs will be demonstrated here; nevertheless the execution behavior of ID programs and the demands placed by ID programs on machine resources will be evident even in the small examples.

This section is thus an introduction to dataflow and discusses material already well-developed by our group. Section III following proposes the architecture we wish to investigate.

2. Elementary Programming in ID

ID is a block-structured expression-oriented single-assignment language. This subsection briefly explains the four fundamental kinds of ID expressions -- blocks, conditionals, loops, and procedure application -- by giving examples of each and their translation into the base language.

2.1 Block Expressions

To evaluate the two roots of a quadratic we can write the following list of expressions or program:

```
( (-b + sqrt(b↑2 - 4*a*c))/(2*a),
      (-b - sqrt(b↑2 - 4*a*c))/(2*a) )
```

However, it is often more convenient for the ID programmer instead to identify and to reference certain partial results, as in the following functionally equivalent block expression, the compilation of which is shown in Figure 1:

```
( x ← sqrt(b↑2 - 4*a*c);
  y ← 2*a
  return (-b+x)/y, (-b-x)/y)
                                              (1)
```

To define some terminology, an assignment statement assigns a variable as the name of the output of an operator (any box in Figure 1); note that assignment is not itself an operator. Variables are used to specify the interconnections among the operators. Assignment statements in a block are separated by semicolons and can always be commuted without affecting the result of the expression. The inputs to a block are those variables referenced but not assigned within that block. The return clause is the last item in a block and specifies the ordered outputs of that block.

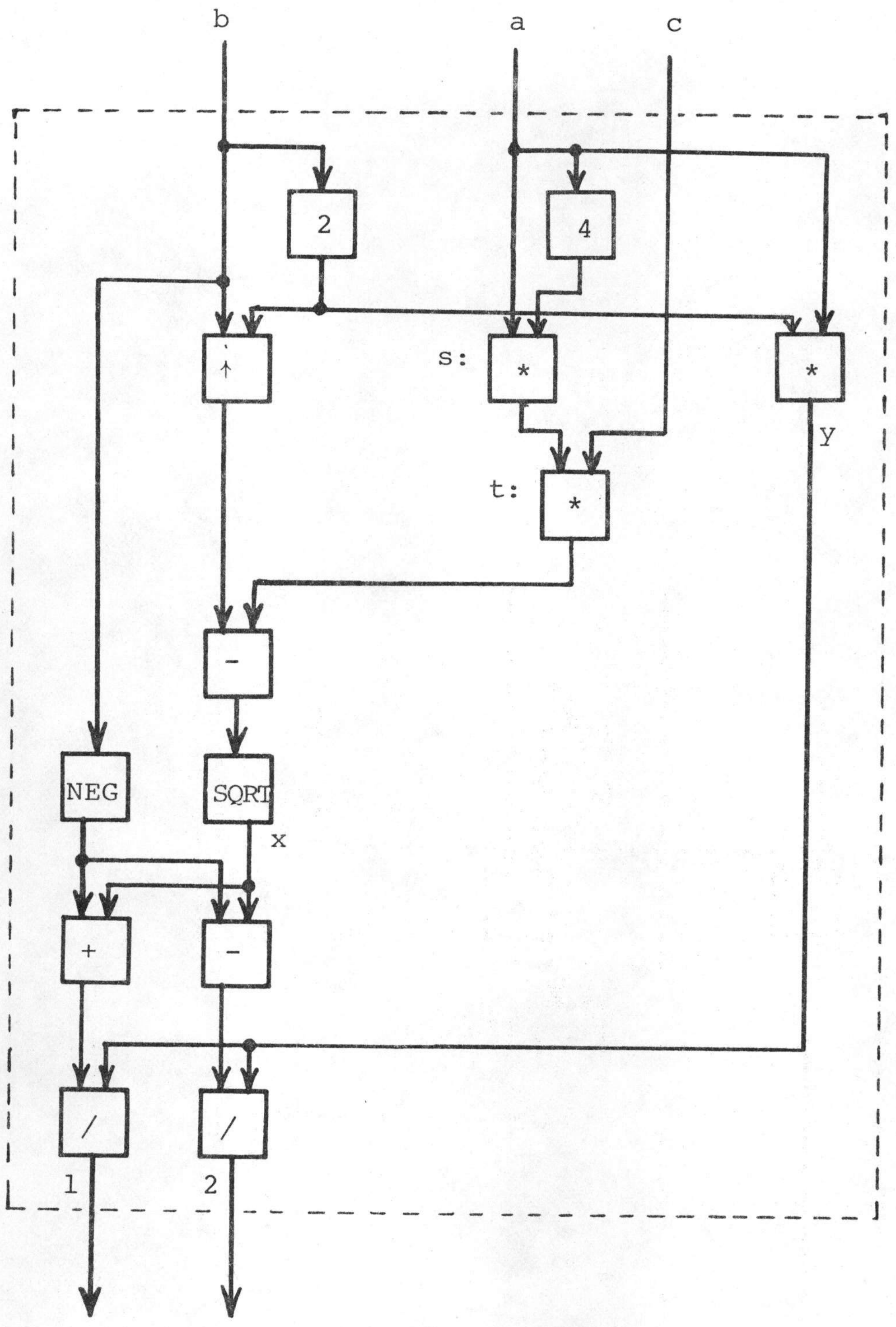


Figure 1

Compilation of the block expression (1)

A variable can be assigned exactly once within its context. This single-assignment rule makes the connection shown in Figure 2 illegal and guarantees that once defined, an instance of a variable never changes in value. This is a concrete implementation of the second principle of dataflow (functionality) for it removes the need for two processes to synchronize the updating of memory since there are no memory cells to update. As a final point of syntax, variable names are scoped to the most recent instance of assignment within the same or an encompassing block, thus allowing variable names to be reused in distinct contexts.

Values in the base language are carried by tokens that flow along lines. According to the first principle of dataflow, an operator executes when and only when all its required input tokens are present. It does so by absorbing those tokens as input, computing a result, and producing an output token that carries that result as its value. Execution of an operator is illustrated in Figure 3a. Figure 3b shows that whenever a token encounters a fork while traversing a line, the token is replicated and follows all outbranches of the fork. In this way a single result may be sent asynchronously as input to many different operators. Note that the order of execution of enabled operators is unimportant since there are no races, i.e., computation is determinate [Arvind & Gostelow-77b, Patil-70].

In a von Neumann machine, the operators address the data; in dataflow, the data addresses the operators. That is, each token actually comprises two fields: value and activity (address).

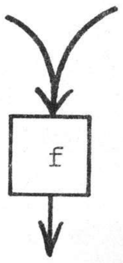


Figure 2

An impossible connection

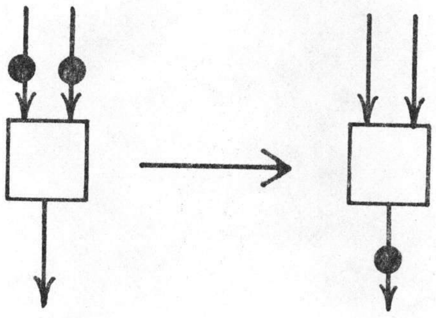


Figure 3a

Execution of a dataflow operator

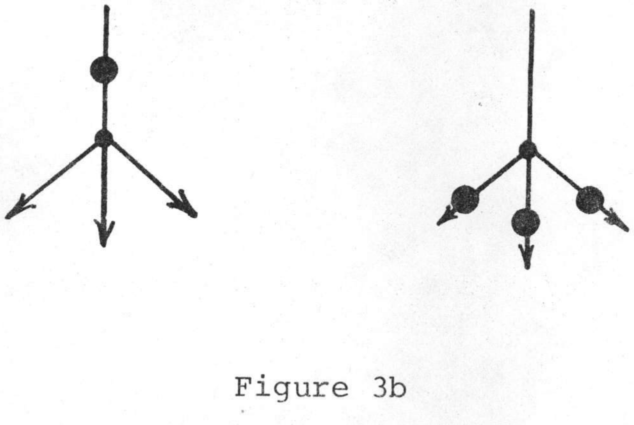


Figure 3b

A fork

The activity field does not physically specify a location, rather it too is composed of two parts: activity name and opcode. The activity name portion uniquely identifies an instance of execution of the destination operator, termed on *activity*, while the opcode simply specifies what primitive is to be executed. Thus if the program of Figure 1 were placed in the body of a loop, the token moving from operator s to operator t on each iteration of the loop would specify the same opcode, but distinct activity names. The important point is that each execution of each operator becomes independent of all other executions of every other (and the same) operator because it has a unique name. Since no memory cells or side-effects exist, it is possible for many instances of the same variable and executions of the same operator to exist at the same time. The method of generating activity names is a simple and mechanical process and is detailed in the Appendix.

2.2 *Conditional expressions*

Consider the ID conditional expression

$$(\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ f(x)\ \underline{\text{else}}\ g(x)) \qquad (2)$$

and its base language translation in Figure 4. If the predicate p(x) is true, then a *true* valued token input to the SWITCH causes x to be sent to box f; otherwise x goes to box g. The MERGE operator ($\otimes$) executes when either of the two inputs arrives and simply copies the input to its output. Thus if p(x) is true, the result of the conditional is f(x), else it is g(x). Both f(x) and g(x) must return the same number of ordered results (one or more).

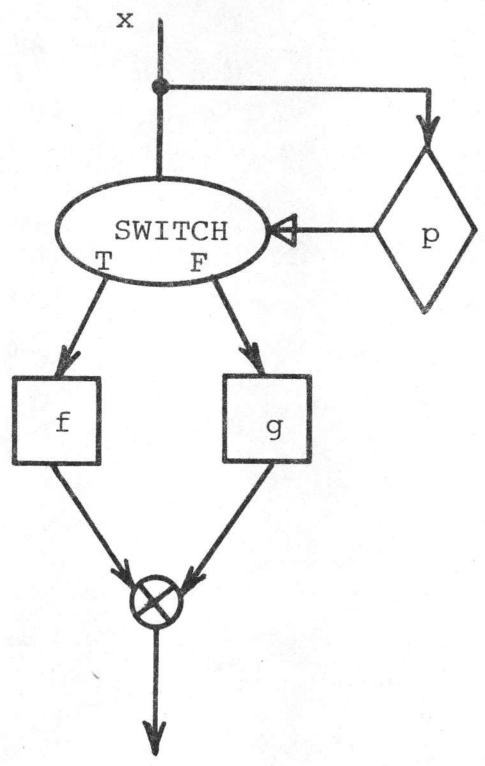


Figure 4

Translation of the _if_-expression (2)

2.3 Loop expressions

All looping constructs in ID are expressions consisting of four parts: an _initial_ part, a predicate to decide further iteration, a loop body, and a list of expressions to be returned as the value of the loop. Consider the loop in the following expression for Simpson's Rule where f is to be integrated from a to b over n intervals of size h:

```
1  (initial s←f(a)/2;
2           x←a+h
3   for i from 1 to n-1 do
4       new x←x+h;
5           y←f(x);
6       new s←s+y
7   return s+f(b)/2)*h                              (3)
```

A loop expression is essentially a set of recurrence relations, where new values of recurrence variables are specified as functions of old values and initial values. In the above example, statements 4 and 6 are recurrence statements where the recurrence variables _new_ x and _new_ s are being computed, both variables having been given initial values. (It is important to notice that y in statement 5 is not a recurrence variable; it is simply a partial result that is referenced in statement 6.) Any reference to a recurrence variable in the body of a loop is to the "old" value of that variable unless the word _new_ precedes the reference. Thus, the x in line 5 does _not_ refer to the value _new_ x computed in line 4. (The value of _new_ x could have been referenced on the right-hand side of line 5 by writing _new_ x instead of just x.)

ID differs from recurrence relations in mathematics only in that a stopping condition must be specified, and the final value(s) of interest must be specified in the <u>return</u> clause of the loop. Changing the order of statements within the loop body does not affect the results (nor the base language translation).

Now let us briefly consider an execution of the above loop expression, compilation of which appears in Figure 5. Suppose function f of line 5 takes a long time to execute. Since the loop predicate $i \le n-1$ does not depend on f(x), the production of n-1 values for x over the range a+h to b will be a relatively fast process. Recalling that each activity is an instance of execution of an operator, we see that instead of accumulating tokens at the x input of box f, many instantiations of f (independent activities) may proceed currently.

We can now briefly explain the operators D, D^{-1}, L, and L^{-1} seen in Figure 5. The D operator changes the activity name (logical destination address) of tokens within a loop for every value which cycles; it does so simply by incrementing a cycle counter position within the activity name. The D^{-1} operator sets that cycle counter back to a 1 -- the same value that the cycle counter begins with at loop initiation. The L and L^{-1} operators enclose every loop. Their purpose is to create a new context for all activity names within a loop by stacking the input activity name (done by the L operator), and to return the results back to the old context upon loop exit (done by L^{-1}). For example, if the function f in the above example were

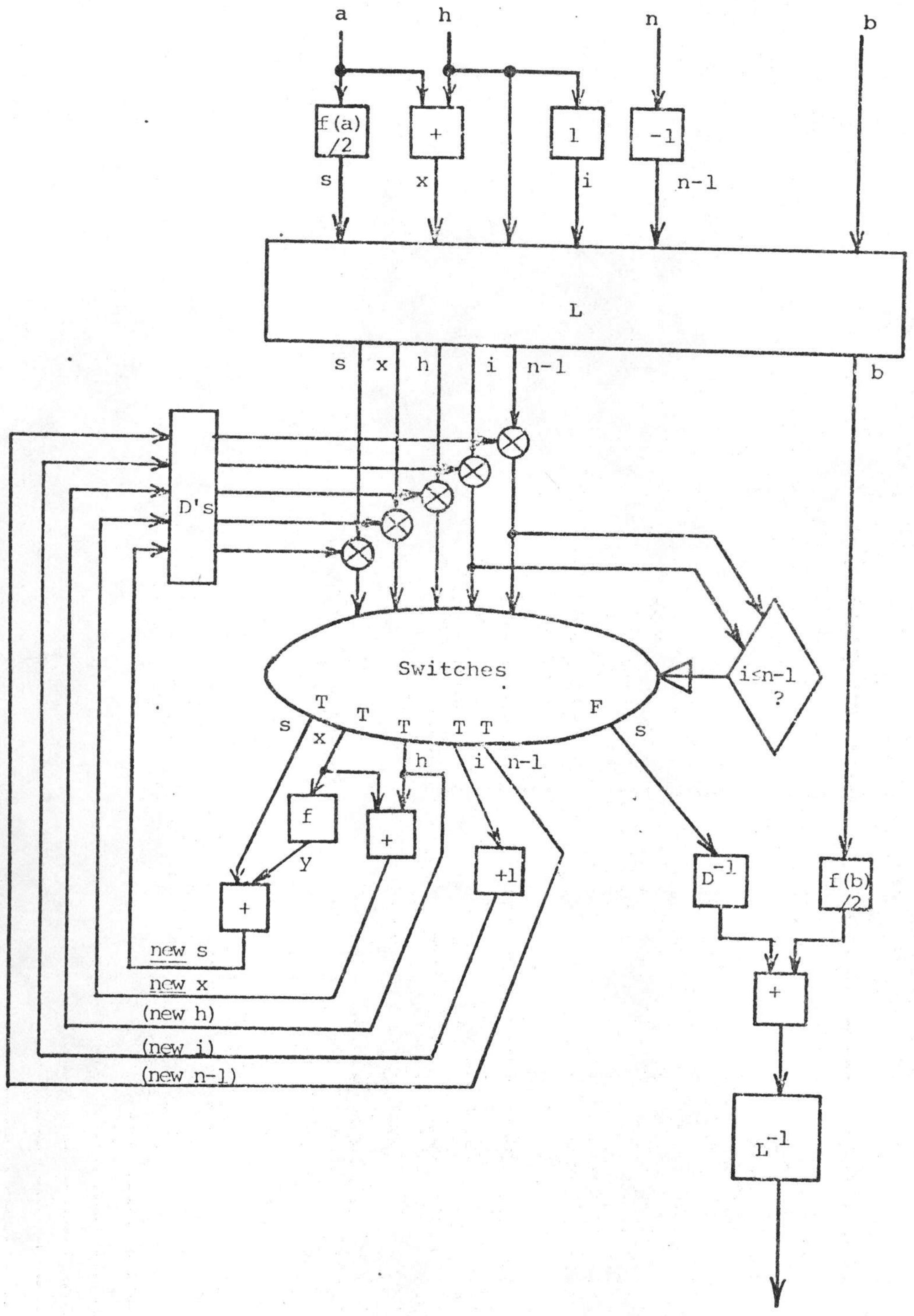


Figure 5

Compilation of the loop expression (3)
(All five SWITCH and D operators have been represented by a single symbol to improve readability.)
]

a loop, then each iteration of the outer loop would create an independent execution of the inner loop, so all initiations of that inner loop could proceed concurrently. This can dramatically reduce the time complexity of an algorithm as shown below.

2.4 Procedure application

Figure 1 showed the ID sqrt function implemented by the machine primitive SQRT. If sqrt were implemented instead as a procedure application, then the SQRT box would be replaced by the APPLY schema at the extreme left in Figure 6. The APPLY operator expects one input token carrying a procedure definition value and another token carrying the argument value. It applies the procedure definition to the argument when both have been received. Note also that "sqrt" is a reference to an output, and we would now say that the block expression to compute the roots of a quadratic needs sqrt in addition to a, b, and c as inputs. The output sqrt then, presumably, refers to a box that produces a constant value of type "procedure definition" that describes a square root function, for example:

sqrt ← <u>procedure</u> (x)(x↑(0.5))

APPLY is actually two operators: A (activate) and A^{-1} (terminate) as shown in Figure 6. The A operator accepts the procedure description and creates an instance of its execution. It does so in a manner similar to that already described by the L and L^{-1} operators by creating a new context for activity names. This allows concurrent executions of procedures from the same APPLY.

3. Program Execution

This section gives an example program in ID, its base language translation, and explains its execution. In particular, we hope to show how demands for large numbers of small processors can be created by dataflow programs and the tradeoff possible between time and space complexity. However we must first explain structure values which are used to represent arrays and other value aggregates.

A structure value is either the distinguished empty structure Λ, or a set of <selector:value> ordered pairs, where "value" may be a simple value (such as an integer or a string) or a structure value. There are exactly two operators defined on structure values -- SELECT and APPEND. If t is the structure in Figure 7a, then t[3]=103 (i.e., the selection of component 3 from t), t[4] is itself a structure, and t[5]=Λ. Multiple selectors are also allowed, so t[4,1]=(t[4])[1]=201. To create the structure of Figure 7b, we append the value 102 to t with selector 2 by writing t+[2]102. Most importantly, the structure created by append in Figure 7b is neither the original structure t nor any modified version of t, since dataflow values cannot be modified. Rather, each append creates a new and logically distinct structure, meaning that the input stucture t has an existence independent of and possibly concurrent with the new structure that is the output of the APPEND operator. This means that the value of t

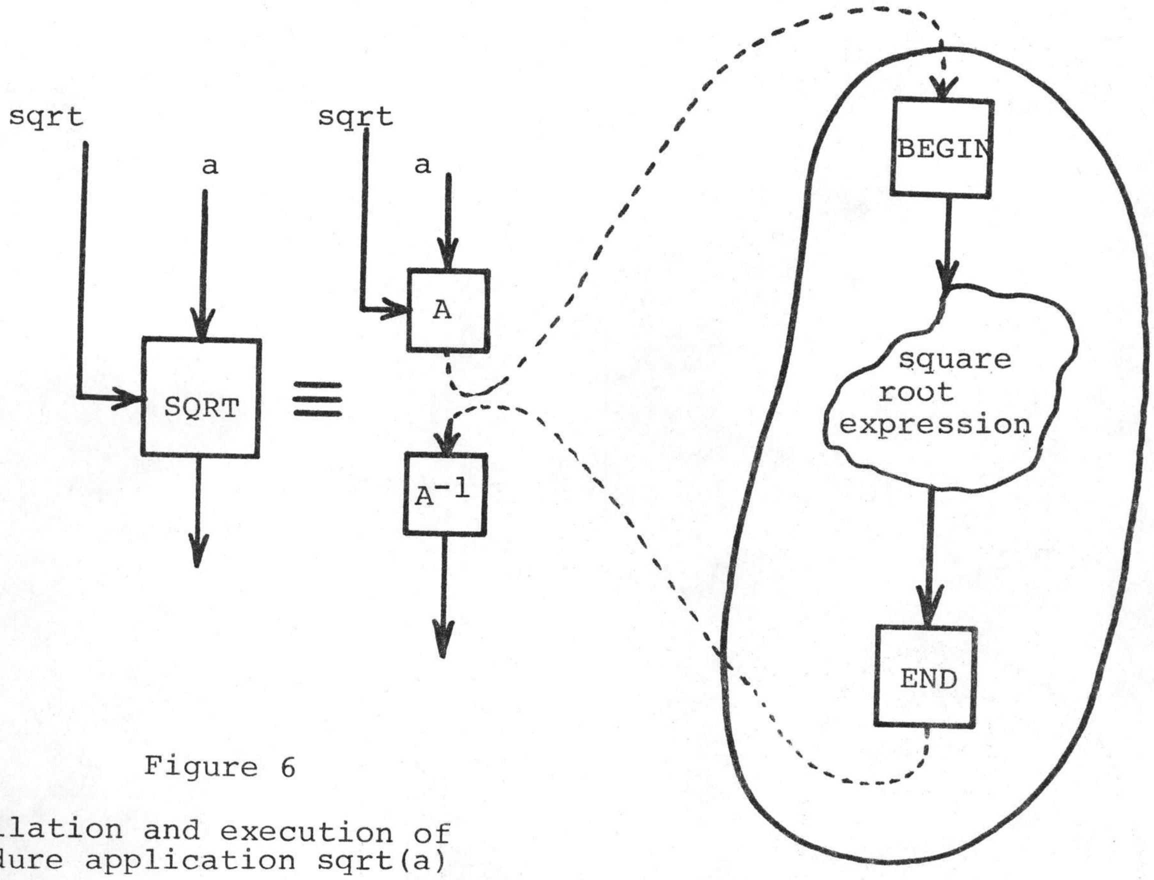


Figure 6

Compilation and execution of procedure application sqrt(a)

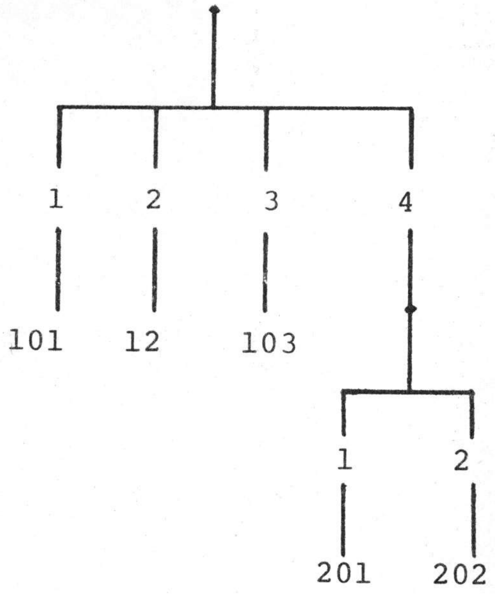


Figure 7a

The structure value t

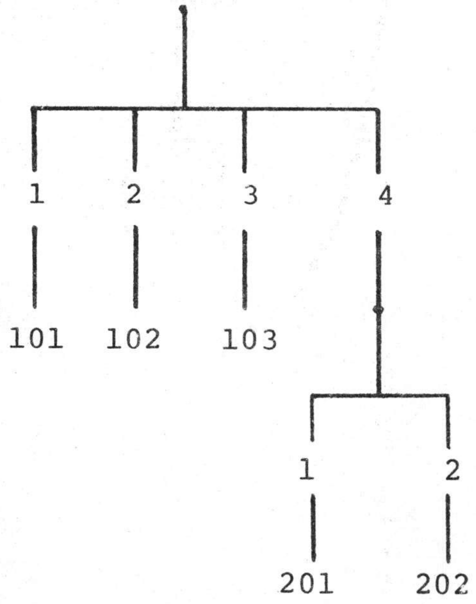


Figure 7b

The structure value t+[2]102

may be referenced by some other expression in the program even after an append to t has been completed. (In most cases implementation tricks can be used to avoid the copying implied by APPEND.)

Also, reminiscent of contemporary languages, we use the syntactic shorthand of (5) for (4):

$$\underline{\text{new}}\ x \leftarrow x+[i]y \tag{4}$$

$$\underline{\text{new}}\ x[i] \leftarrow y \tag{5}$$

The following ID procedure multiplies an $\ell \times m$ matrix a by an $m \times n$ matrix b in the straightforward way. A matrix is represented as a vector of row vectors.

```
procedure multiply (a,b,ℓ,m,n)
   (initial c←Λ
    for i from 1 to ℓ do
     new c[i] ← (initial d←Λ
                 for j from 1 to n do
                  new d[j] ← (initial dp←0
                              for k from 1 to m do
                               new dp ← dp+a[i,k]*b[k,j]
                              return dp)
                 return d)
    return c)
```

A compiled version of the above procedure is shown in Figure 8. In order not to obscure the principles, only the non-constant variables are shown inside the loops. That is, in the formal system (but not necessarily the implementation), the variables a,b,i,j,m are all constants but must all cycle in the innermost loop since they are all inputs to operators in the body of the loop. To include them would simply clutter the Figure. Also, each loop has been outlined by a box to indicate its scope.

Recall from the discussion of Section 2.3 that the L and L^{-1} operators allow each instance of execution of a loop to proceed independently, just as if that loop were a primitive box relative to the surrounding context. However, a loop is unlike a primitive in that the loop itself may be composed of other loops (nesting), each of which might again proceed independently in execution. The effect of this at execution is for the program loops to "unfold," and its significance in terms of generating large numbers of small tasks is most clearly seen in terms of the time complexity of execution. For matrix multiply the time complexity is $O(\ell+m+n)$ rather than the $O(\ell mn)$ for a sequential machine. The space complexity (processors) for our dataflow interpreter is $O(\ell m)$, whereas for a sequential machine it is a constant 1.

The reduction in time complexity for matrix multiply is dramatic; nevertheless, it is quite common for the unfolding interpreter to reduce complexity by a factor of n or n^2. This is not always the case, and some programs maintain the same time complexity in dataflow as for a sequential machine. Others show only marginal gains, for example, Hoare's quicksort goes from $O(n \log n)$ to $O(n)$.

We wish to emphasize that there is no centralized controller issuing commands for parallel operation such as in ILLIAC IV. The basic operation here is the generation of activities, each activity being carried out independently by a small processing element that forwards its result tokens to other activities for continued processing.

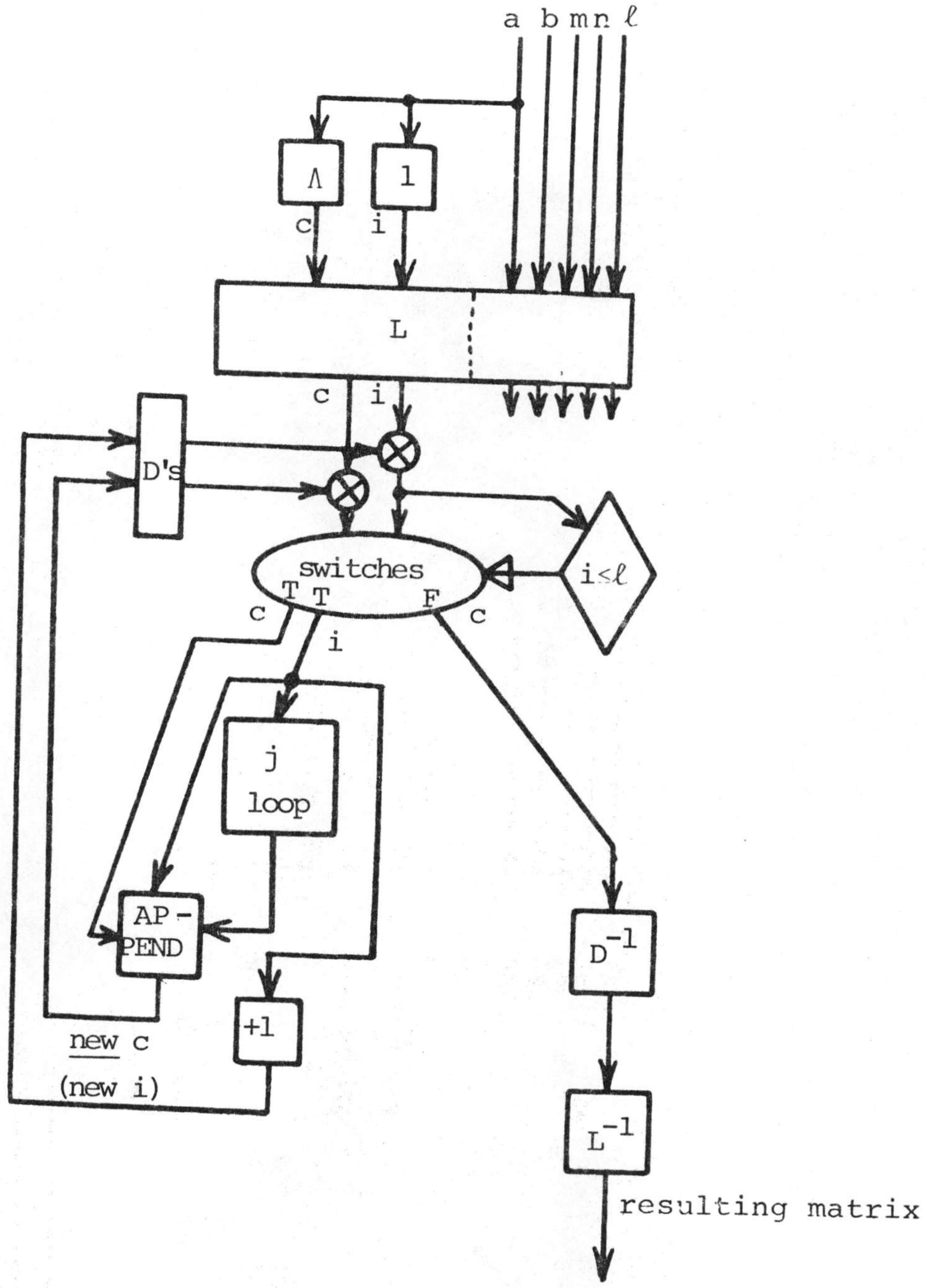


Figure 8 - Part 1

The outermost loop (i loop) for matrix multiply

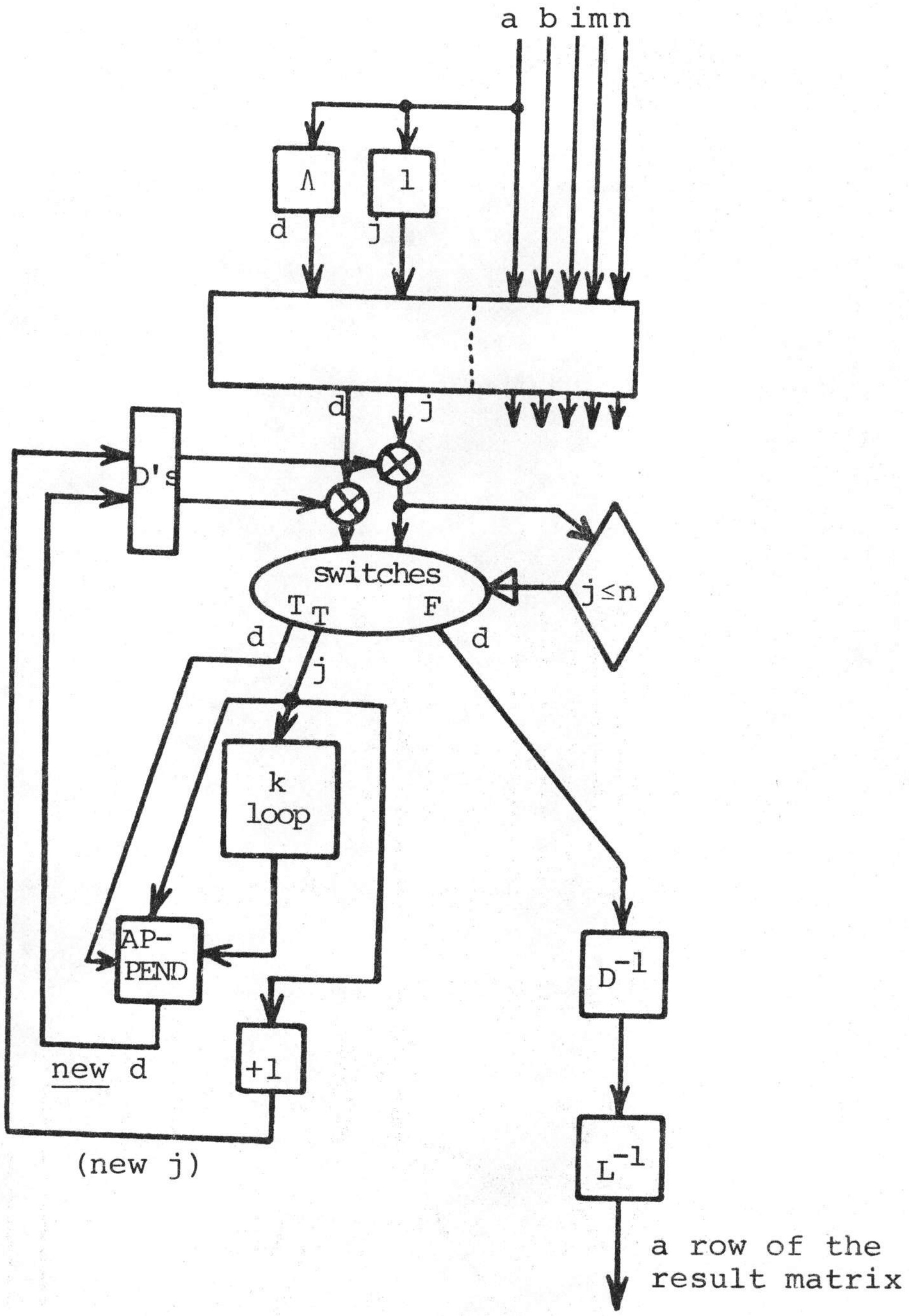


Figure 8 - Part 2

The middle loop (j loop) that
returns a row to the outermost loop (i loop)

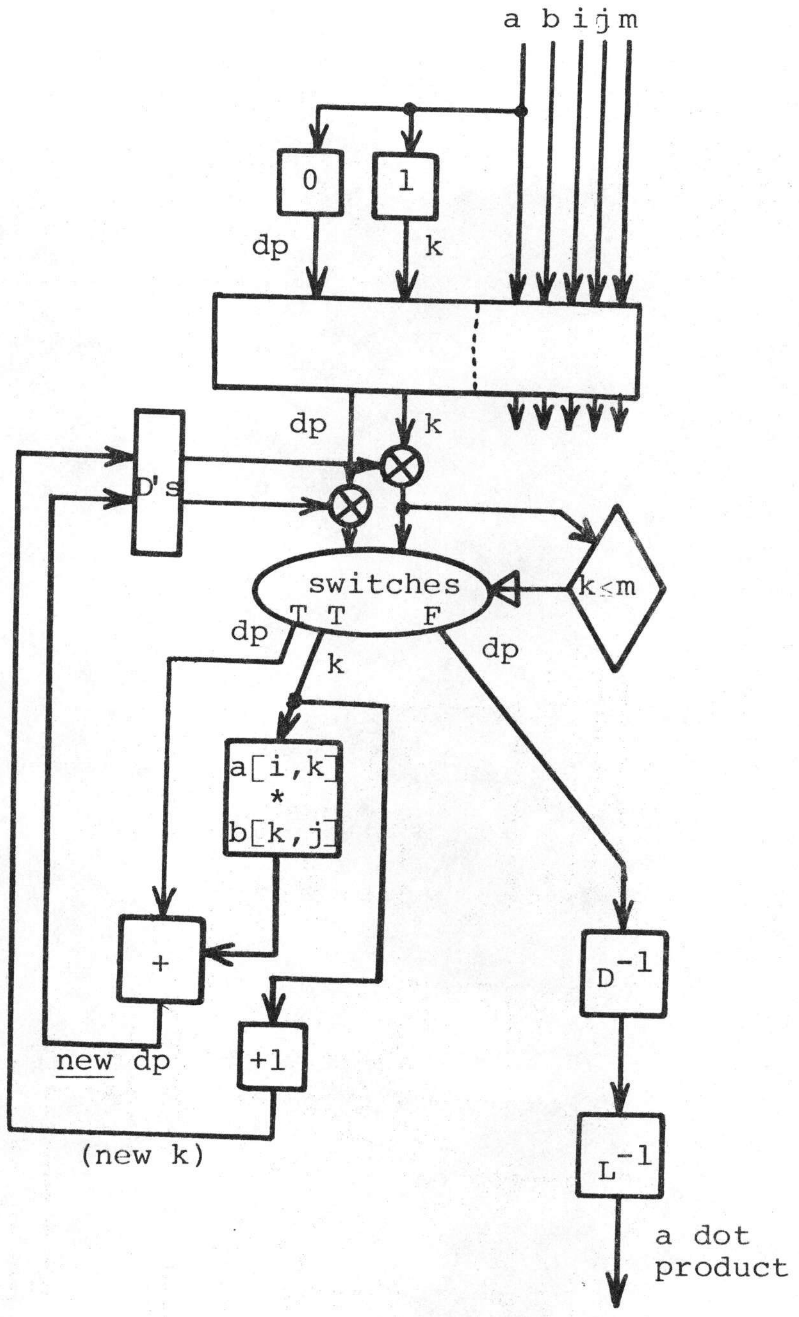


Figure 8 - Part 3

The innermost loop (k loop) that
returns a dot product to the middle loop (j loop)

III. The Proposed Machine

1. Introduction

This section presents the initial architecture we plan to investigate during the proposed research period. The goal of the design is to implement the unravelling interpreter discussed in Section II. Many problems associated with such a machine must be solved. Some of these problems are directly related to the architecture (such as selecting an appropriate token bus, and ensuring a modular and expandable design) while most are "system"-oriented (such as how to achieve fault-tolerance, and some problems in programming and implementing input/output). The focus of the work is the eventual specification of the machine and its components. To that end we intend to study several issues required for a complete system. Even though the major portion of the following discusses architecture (Section 2), we will devote no less attention during the research period to these other important issues (Section 3).

2. The architecture

Figure 9 shows an ensemble of processing elements (PEs) connected on the one side to a token communication system (token bus) and on the other side to a memory system. Memory, of course, is not intrisic to dataflow semantics; its purpose is simply to avoid transmission of large pieces of data (such as structures and program code) by tokens. This is accomplished by storing the structure in the memory and by sending only a pointer to the structure on the token itself. This scheme implies a memory controller MC attached to each PE which, when presented with a pointer, can retrieve the corresponding structure (or portion thereof) from the memory. We assume there is a large number of memory modules, each module connected to one MC. A PE makes all requests for stored information through its attached MC. Since data may be stored in a non-local memory, it is clear that an MC must have the ability to communicate with other MCs. Memory controllers are therefore interconnected by a bus as shown in Figure 9.

The semantics of the high-level language ID guarantee that no two invocations of a procedure interact with one another except through input parameters and results. A similar statement holds for invocations of a loop expression. All activities (invocations of operators) belonging to the execution of a particular procedure or loop, but not to any inner procedure or loop, are said to comprise a <u>logical domain</u>. It is clear, then, that tokens belonging to two different logical domains are best kept physically apart. Similarly, structures stored in memory and referenced by activities in different logical domains should also be kept apart from one another and near their respective accessing domain. To actually separate logical domains, the machine will have the ability to partition into separate <u>physical domains</u> where each physical domain will comprise some number of PEs, MCs, and associated bus capacity. Hence, initiation of a logical domain may bring about the creation of a new physical

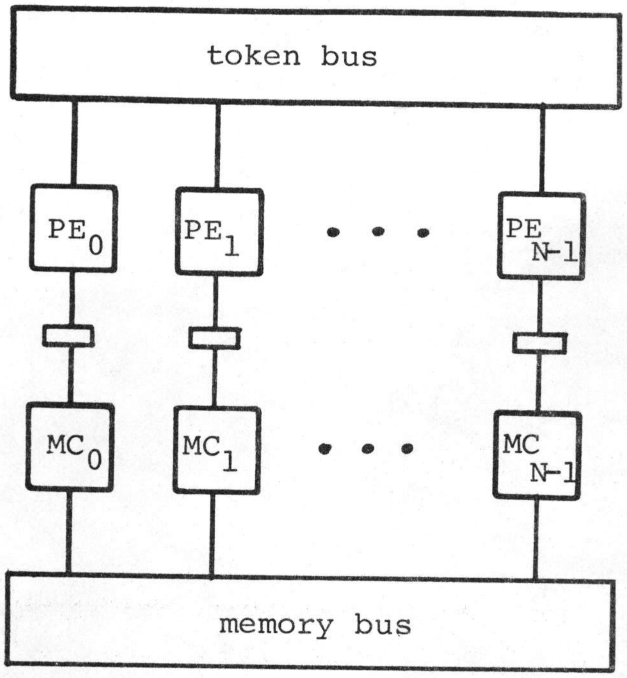


Figure 9

The basic architecture

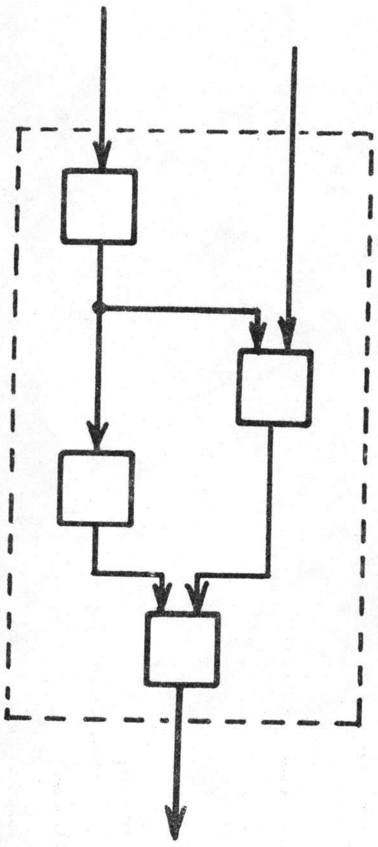


Figure 10

Coalescing of operators

domain. One or more logical domains may share the resources of a single physical domain. The ability of the system to partition itself into disjoint physical domains is considered very important in reducing communication interference and overhead. We plan to verify (or disprove) this conjecture during the research period.

The following subsections describe the token bus, memory bus, and PEs in greater detail and what we need to learn about them. The primary tools include the simulator and the compiler to translate ID programs into base language code for input to the simulator. The simulator collects many useful statistics regarding PE and bus requirements as a function of time. Besides the size of the machine (which may vary dynamically according to load), the relative speed of the communication system and the PEs are also input parameters.

2.1 The token bus

The system with which we are currently experimenting employs a segmented ring as a token bus with PEs attached to each segment. By attaching or detaching segments, the ring can be made larger or smaller during execution. A ring defines a physical domain, and by dividing a ring into separate sub-rings we can create physical domains dynamically.

Concerning addressing of tokens to PEs, simulation results have shown that purely logical addressing (by activity names) of tokens to PEs is not practical -- performance was not good and degraded severely under sub-optimal conditions (e.g., shortage of PEs). To implement physical addressing, we must assign an activity to a particular PE and be sure that every token input for the activity is sent to that PE. More than one activity may be assigned to one PE. Operationally, each PE in a physical domain of N PEs is given an address between 0 and N-1. When a token is output by a PE, a physical address is computed and the token moves via the bus to that PE. For speed and efficiency, a hash function is used to compute a physical destination PE address from the activity name. So far, we have experimented with only a very simple hashing function: a linear combination of the operator label and a numeric label given to the token's logical domain, modulo N. However, any such hashing function must satisfy several criteria:

1. Activities should be evenly distributed over time and space (the PEs).
2. To avoid set-up time for PEs (such as code fetch) many invocations of the same operator should take place on the same PE.
3. The hash function must operate over a physical domain of any given size.

We hope during the research period to measure machine performance under various hashing functions and thereby to determine a good activity assignment algorithm. However, it may be that no simple function can be found, in which case we will supplement it with heuristics to incorporate the most important parameters. This is, in fact, just a first attack on the assignment and scheduling problem which we have selected for special investigation and listed separately in Section 3 below.

In describing the token bus, we have been using the example of a ring implementation. The proper bus structure, however, is a matter we intend to study. There appears to be a wide variety of possibilities here, and we propose to study at least three schemes: the ring structure (the current system), Pierce loops [Pierce-72], the Wittie bus [Wittie-76], and possibly other communication networks. It is very difficult to determine a priori what the machine's behavior should be under each of these schemes, and theoretical analysis is not possible due to the non-stationary nature of the system. Thus, we must simulate. One characteristic of the ring system, however, is that as the number N of PEs increases in a physical domain, communication delay increases as O(N), while other more point-to-point systems (the Wittie bus, for example) tend to O(log N) increases in delay. Each of the above busing systems is capable of partitioning in some way and thereby capable of supporting physical domains, however the ring systems may require more complex control than, again, a Wittie bus. On the other hand, the Wittie bus appears less amenable to modular construction than the ring systems.

2.2 Processing elements (PEs)

Basically, each PE is a three-component pipeline composed of token input, computation, and token output sections. Since there may be many activities assigned to a given PE, a token (which may belong to any one of several activites) enters the input section of a PE where it is queued. Part of the duty of the computation section is to cycle and remove each token from the input queue, sort it, and append it to the appropriate partial (as yet incomplete) activity list. When the last token arrives and completes an activity, a request to carry out that activity is queued. These requests are also processed by the computation section (according to a scheduler yet to be devised). When the activity has been processed, the computation section manufactures the output tokens and queues them for output. The output section then sends the tokens back into the token communication system to make their way to their respective destinations.

The most unusual aspect of a PE is the need to queue tokens and sort them. Experiments conducted so far on our set of sample programs show that a properly balanced system (a system where the number of PEs per physical domain achieves minimum execution time) requires that the list of partial activities holds an average of 15 tokens; the maximum we have seen is 64 tokens for the worst case of one physical domain with one PE to execute an entire matrix multiplication. In terms of capacity (we assume a maximum of 100 bits per token) no problem is foreseen. Speed, however, will quite possibly require careful design, for although average token arrival rates may be reasonable, burst rates may often exceed (high-speed) queue lengths. We are aware of the potential for problems here, and plan to resolve them during the research period.

The computation section appears straightforward, though it may be appropriate to have more than one computation section in a single PE in order to handle all the duties that a PE must perform, including token sorting, code and data fetch, and the computation itself. There are, however, at least three ways in which performance may be improved. First, code fetch requests can be generated even before an activity is complete. Second, we anticipate that a single PE will be assigned to execute several invocations of the same operator and the code need be fetched only the first time that operator is executed. Third, even though previous discussion has implied that each operator box in the base language program is a distinct activity, this need not be so. For example, the sequence of operators in Figure 10 could be <u>coalesced</u> into a single operator or macro-activity, all of which the computation section carries out internally in sequence. This eliminates unnecessary token traffic and PE set-up overhead.

We do not intend to emphasize fast PEs, but rather hope for speed of processing (along with reliability) as a result of distributing the work over a large number of processors and eliminating overhead where possible. We expect that the coalescing of operators mentioned above will be significant here. Thus we propose to investigate the effects of coalescing, one result of which is to increase the "grain" of an activity in an indirect way. That is, we are quite sure that scalar addition is, in most cases, too small to be treated alone as an activity. Coalescing can be used to raise the granularity in a natural way to an appropriate level.

Finally, the output section poses no functional problems other than one shared by all sections of a PE -- finiteness of its local working store. Overflow is possible at any point, and if not handled deadlock will follow. We assume that as a last resort storage can be obtained from the memory system to avoid deadlock, but we need to study the effects of these boundary conditions on performance.

2.3 <u>The memory system</u>

The memory system is perhaps the least understood component of the machine, though we anticipate patterns of use similar to that for the token bus. As mentioned before in motivating the concept of a physical domain, just as locality is a factor in the token communication system, so is locality a factor in the memory system. For example, we can imagine a situation where each of several physical domains contains some logical domains that repetitively reference a particular structure. Since values in dataflow can never be modified, the memory system can make copies of the structure and send the copies once and for all to the respective physical domains rather than suffer repeated delays from non-local access. Such an arrangement can be compared to conventional cache memory organizations used to speed memory access. We expect that only parts of a (large) structure will be copied on demand. As the computation proceeds, copies may be deleted to make room for more active data structures.

We have much to investigate in this area of the machine, and during the research period we plan to determine just what kinds of memory system demands are generated by the PEs. We then plan to investigate some alternative bus structures and to determine their performance. Finally, we must investigate the mechanisms that will accomplish the above mentioned copying of data structures to avoid non-local access and permit parallel accessing of future memory requests.

3. Other Areas of Investigation

Along with our investigation into the architecture of the dataflow machine, we plan detailed studies in the following areas which must be understood before such a machine can be built.

3.1 Fault-tolerance

Reliability and fail-soft are two very important characteristics of any complex system. Computers, however, have been notoriously deficient in this regard. (Reference [Misunas-76] shows a first approach for one dataflow machine.) We feel that the unique characteristics of the machine proposed here bring new opportunity for improvements in this area. Our approach is multi-level. At the highest level is programmer-controlled error handling and protection. In this regard two projects are currently underway. One graduate student is investigating a software-reliability and error-handling facility for ID which also impacts the base language operators. The goals here are to develop a model by which the programmer can view exceptional conditions and handle them completely in ID, if desired. A second goal is that computation should otherwise proceed asynchronously as far as possible, but without allowing the computation to run away. Also at the higher-level is the work of another graduate student on incorporating a protection mechanism into ID ([Bic-77a,b]). This work is scheduled for completion in June 1978, and results are encouraging. The scheme is based on attaching "password" tags to the data to define that data's protection domain. Every base language operator then accepts tagged input data and computes a new tag (or aborts the operation if the tags are not commensurate) for the output data. The scheme is capable of solving protection problems not previously solved on any machine, including the proprietary services problem in all published (and several unpublished) forms. It is a general scheme that is quite easy to implement, and its success is based in large measure on the fact that dataflow is a functional (side-effect free) language.

Once this higher-level work is complete, we will be able to begin work on the lower-level aspects involving message retry, and logical removal of failed components (such as processors and memory boxes) from the pool of available resources. We hope to be able to accomplish this by explicitly incorporating error checking facilities into the design of the machine's components, and by taking advantage of the fact that the major resources are already pooled and that much of the data may exist in several copies in different areas of the machine.

3.2 Performance analysis

This is a broad area which, even for a particular architecture, requires investigation in order to achieve the most from the design. Within this category we include activity assignment and scheduling studies, and similar investigation into the memory system's MCs and how they represent and manage the data they store.

To get a feel for the machine's behavior and what we hope to be able to achieve, we give some preliminary results recently gathered from the simulator for the current design. Figure 11a shows how execution time varies with the number of PE resources devoted to the computation* when the entire computation is confined to one physical domain. The leftmost end of the graph is the total execution time for one PE. However, as additional PEs are allocated to the computation, execution time decreases rapidly. In general, there is a broad flat portion of the curve where the minimum execution time is produced by a range of PE resources (typically a factor of 5 between points A and B). Points to the right of B indicate that the benefits of additional PEs are offset by increased communication costs. The relatively flat spot between A and B is advantageous since a rather crude resource estimate may be sufficient to produce the best performance.

In fact, it appears that the parameters

$\bar{C}$ = mean token transmission time
$\bar{Q}$ = mean computation section queue time
$\bar{O}$ = mean output section queue time

are good monitors of the optimum number of PEs. That is, whenever the number of PEs N is increased or decreased so that

$$\bar{C} \approx \bar{Q} + \bar{O}$$

then N invariably falls between the points A and B. If N is outside A and B, these same parameters also indicate the direction in which N should be moved. The parameters are easily measured by a machine during execution, so self-regulation may be possible, How many PEs is optimum depends

*The programs we have used in our tests (compiled from ID code) are matrix multiply, Gauss elimination, Gauss-Seidel approximatiom, least square regression, quicksort, and optimal binary tree generation.

upon the program, but numbers range from five to forty PEs on our (small) test programs; for example, forty PEs was optimum for multiplying two 7×7 matrices.

Figure 11b compares the simulator with theoretical bounds. The complexity k is the size of the input, where for the case of multiplying an $\ell \times m$ and an $m \times n$ matrix we let $k=\ell+m+n$. Observing the bounds we do not achieve the minimum performance $O(k)$, but we do better than $O(k^2)$ and certainly better than the $O(k^3)$ of a sequential machine. The two solid lines may be compared to show the effect of a memory system. In fact, the measure shown that includes memory is for a system that takes advantage of locality of reference by copying structures as discussed in Section 3.3 above. Without this very important feature, performance falls dramatically and approaches $O(k^3)$.

A final point concerns the utility of dynamically varying the size of a physical domain in concert with the demands of an ongoing computation. Experiments have shown that tracking a computation so that near optimal PE resources are provided at all times reduces computation time only minimally. That is, a good constant domain size has low internal costs. However, high external costs can result since other neighboring physical domains can suffer from inadequate resources that otherwise might have been available had that domain reduced its size and thus freed PEs for use by those neighbor domains. This result is in many respects a verification of Figure 11a, and has important implications for how resources might be allocated in a dataflow machine -- a problem we have only begun to touch.

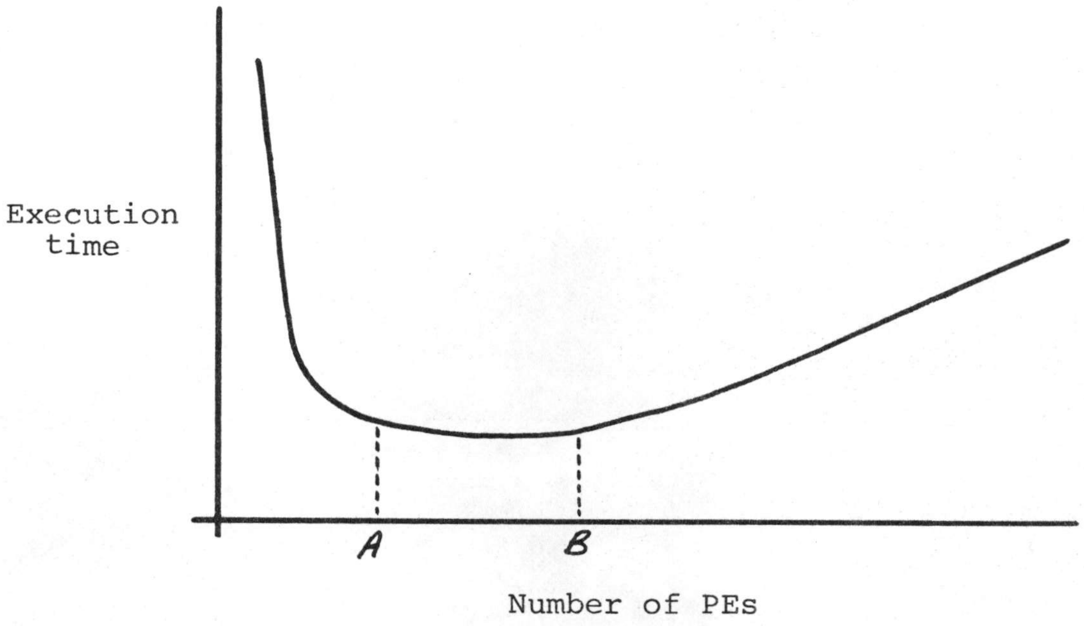


Figure 11a

General curve for execution time vs. allocated PE resources

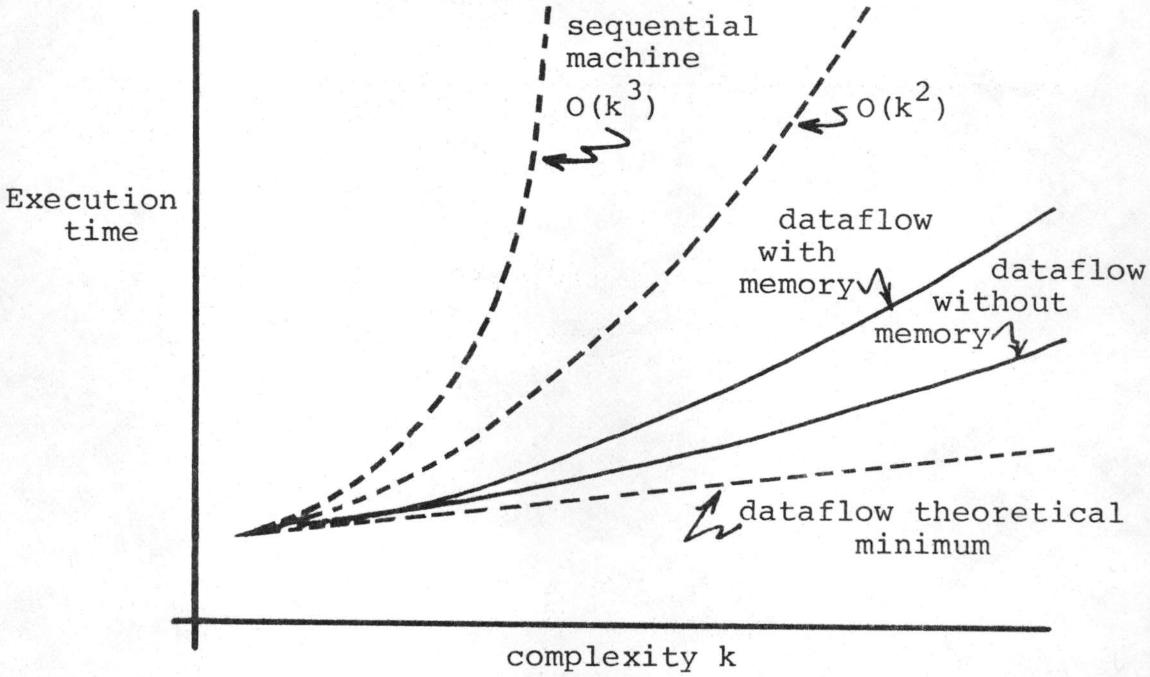


Figure 11b

Execution time vs. complexity for matrix multiply

3.3 Evaluation and improvement of ID

ID is a complete programming language. However, several aspects are in need of study and improvement. A particularly tricky problem is specifying sequentiality in a convenient and efficient manner. This need arises especially in input/output programming.

We also plan a more overall appraisal of ID by gathering the experiences of undergraduate and graduate students who will be using the language through our ID to LISP compiler. Some particular difficulties we expect to find in ID are in nested loops where the language often forces the programmer to be aware of and to name partial results in which he has no interest, for example the results passed back from an inner loop to the next outer loop.

4. Summary

The system presented here is not in the usual mold of a "parallel" computer. Rather it is based upon sound fundamentals markedly distinct from the more traditional approaches. Many old problems (e.g., scheduling of tasks) must be solved anew in order to create a useful machine, but many other problems (e.g., protection) seem significantly eased. It has been the purpose of this section to discuss our ideas on the architecture for a machine, and to present some of the problems we hope to solve. These problems include the basic design of the computer and its components, and how these components are linked together. Communication is the fundamental difficulty, and is followed closely by scheduling and the memory system; other principal areas in need of study are fault-tolerance and some special issues in language design.

IV. UCI Dataflow Architecture Project Literature List

1. Papers and technical reports

1. Arvind, K. P. Gostelow, "Microelectronics and Computer Science," <u>Proceedings of the 2nd IEEE(G PHP) /ISHM University/Industry/Government Microelectronics Symposium</u>, University of New Mexico, Albuquerque, New Mexico, January 3-5, 1977. (Also available as Tech Report #106.)

2. Arvind, K. P. Gostelow, Wil Plouffe, "Some Relationships Between Asynchronous Interpreters of a Dataflow Language," <u>Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts</u>. St. Andrews Canada (TC2-Programming Languages), August 1-5, 1977. (Also available as Tech Report #88A.)

3. Arvind, K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," <u>Proceedings IFIP Congress '77</u>, Toronto, Canada, pp. 849-853. (Also available as Tech Report #77A.)

4. Arvind, K. P. Gostelow, W. Plouffe, "Indeterminacy, Monitors, and Dataflow," <u>Proceedings of the Sixth ACM Symposium on Operating Systems Principles</u>, Purdue University, November 16-18, 1977. (Also available as Tech Report #107.)

5. Arvind, K. P. Gostelow, "A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture," Tech Report #72. Department of Information and Computer Science, University of California, Irvine, (October 1975).

6. Arvind, K. P. Gostelow, Wil Plouffe, "Programming in a Viable Data Flow Language," Tech Report #89, Department of Information and Computer Science, University of California, Irvine (Aug. 1978).

7. Thomas, Robert, "Performance Analysis of Two Classes of Dataflow Computing Systems," M.S. Thesis, Department of Information and Computer Science, University of California, Irvine (Jan. 1978).

8. Arvind, K. P. Gostelow, and Wil Plouffe, "The ID Report: Syntax and Semantics of a Higher-level Dataflow Programming Language," Tech Report #114, Department Information and Computer Science, University of California, Irvine (March 1978).

2. Dataflow Notes (by number)

These notes are working papers only. Those marked as obsolete are no longer available -- their contents have either been discarded or have been incorporated into technical reports and papers. Dataflow Notes are not complete papers, are often brief, and are a first pass over an idea which may change at any time.

1. Gostelow, K. P., "Physical Domains," (May 3, 1976).

2. Gostelow, K. P., "Logical Domains," (May 3, 1976).

3. Plouffe, W., "Replacing Looping Structures by Data Flow Primitives," (May 6, 1976) (obsolete).

4. Plouffe, W., "A Case Study in the Use of Transformations to Improve Performance," (May 6, 1976) (obsolete).

5. Thomas, B., "Simulator Progress," (July 26, 1976) (obsolete).

6. Thomas, B., "Variable Bindings," (June 13, 1976).

7. Gostelow, K. P., "Object Domains," (August 5, 1976) (obsolete).

8. Arvind, "A Note on the Gate Operator," (July 19, 1976).

9. Gostelow, K., "Interprocess Communication," (August 21, 1976)(obsolete)

10. Minne, J., "Introduction to RED Languages," (January 24, 1977).

11. Arvind, and Gostelow, K. P., "Semantics of Loop Expressions in ID," (March 4, 1977).

12. Plouffe, W., "An Alternative Scheme for Token Matching," (February 14, 1977).

13. Minne, J., "Programming in Backus' RED Language," (March 24, 1977).

14. Arvind, and Plouffe, W., "Definition of Dataflow Operations," (June 30, 1977) (obsolete).

15. Arvind, Gostelow, K. P., and Plouffe, W., "Compilation of ID Loops Involving Streams in the New Base Language," (July 5, 1977)(obsolete).

16. Arvind, Gostelow, K. P., and Plouffe, W., "Environments and Procedures," (July 8, 1977)(obsolete).

17. Arvind, and Gostelow, K. P., "Some Relationships Between Asynchronous Interpreters of a Data Flow Language," (August 1977).

18. Thomas, B., "Compiler Documentation: ID to Base Language," (August 16, 1977).

19. Morris, P., "Compiler Documentation: ID to LISP," (August 16, 1977).

20. Arvind, and Gostelow, K. P., "Presentation at IFIP Congress '77 - Introduction to Dataflow," (August 22, 1977).

21. Arvind, and Gostelow, K. P., "Annual Technical Report Summary," (July 20, 1977).

22. Bic, L., "A Basic Model for Protection in Dataflow," (September 27, 1977).

23. Bic, L., "An Extended Model for Protection in Dataflow," (November 10, 1977).

24. Minne, J., "Stream Programming in RED Languages," (December 14, 1977).

25. Bic, L., "Protection in Dataflow," (November 20, 1977).

26. Bic, L., "Confined Interprocess Communication," (March 1, 1978).

27. Bic, L., "Proprietary Services for Dataflow," (February 8, 1978).

28. Gostelow, K. P., "Dynamic Linking of Programs," (February 14, 1978).

V. References

[Arvind, Gostelow, & Plouffe-76], Arvind, Kim P. Gostelow, and Wil Plouffe, "Programming in a Viable Dataflow Language," TR89, Department of Information and Computer Science, University of California, Irvine, (August, 1976).

[Arvind, Gostelow, & Plouffe-77], Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, Monitors and Dataflow," Proceedings of the Sixth ACM Symposium on Operating Systems Principles, Purdue University, Operating Systems Review, Vol. II, No. 5., pp. 159-169, (Nov. 16-18, 1977).

[Arvind & Gostelow-77a], Arvind, and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Proceedings IFIP Congress '77, Toronto, Canada, pp. 849-853.

[Arvind & Gostelow-77b], Arvind, and K. P. Gostelow, "Some Relationships Between Asynchronous Interpreters of a Dataflow Language," Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, St. Andrews, Canada (TC2-Programming Languages)(August 1-5, 1977).

[Ashcroft & Wadge-77], Ashcroft, E. A. and W. W. Wadge, "LUCID, A Nonprocedural Language with Iteration," CACM, Vol. 20, No. 7, pp. 519-526 (July 1977).

[Bährs-72], Bährs, A., "Operation Patterns," International Symposium on Theoretical Programming (Novosibirsk, 1972), Lecture Notes in Computer Science 5, Springer-Verlag, pp. 217-246, (1974).

[Backus-73], Backus, John, "Programming Language Semantics and Closed Applicative Languages," IBM Research Report RJ 1245, San Jose, CA (July 5, 1973).

[Bic-77a], Bic, Lubomir, "A Base Model for Protection in Dataflow," UCI Dataflow Note #22, Department of Information and Computer Science, University of California, Irvine (September 1977).

[Bic-77b], Bic, Lubomir, "An Extended Model for Protection in Dataflow," Dataflow Note #23, Department of Information and Computer Science, University of California, Irvine, (November, 10, 1977).

[Chamberlin-71], Chamberlin, Donald D.,"The 'Single-Assignment' Approach to Parallel Processing,"IBM Research Report RC 3308, T. J. Watson Research Center (March 1971).

[Davis-77], Davis, A. L.,"The Architecture of DDM-1: A Recursively Structured Data Drive Machine," Report UUCS-77-113, Department of Computer Science, University of Utah, Salt Lake City (October, 1977).

[Dennis-73], Dennis, J. B. "First Version of a Data Flow Procedure Language," MAC TM 61, MIT, (May 1975).

[Dennis & Misunas-74], Dennis, J. B., and D. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," Proc. 3rd Annual ACMSIGARCH-IEEE Symposium on Computer Architecture, pp. 126-132, (December 1974).

[Friedman & Wise-76], Friedman, D. P. and D. S. Wise, "The Impact of Applicative Programming on Multiprocessing," Proc. of the 1976 International Conference on Parallel Processing, (P. H. Enslow ed.) pp. 263-272, (August 1976).

[Guttag-77], Personal Communication.

[Karp & Miller-66], Karp, R. M. and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing," SIAM J. Appl. Math. Vol. 11, No. 6, (November 1966).

[Kessels-77], Kessels, J. L. W. "A Conceptual Framework for a Nonprocedural Programming Language," CACM, Vol. 20, No. 12, pp. 906-913, (December 1977).

[Kosinski-73], Kosinski, P. R., "A Data Flow Language for Operating Systems Programming," ACM-SIGPLAN Notices, Vol. 8, No. 9, pp. 89-94 (September 1973).

[Misunas-76], Misunas, D. P., "Error Detection and Recovery in a Dataflow Computer," Proc. 1976 IEEE International Conference on Parallel Processing, Long Beach, CA, pp. 117-122, (August 1976).

[Patil-70], Patil, S. S., "Closure Properties of Interconnections of Determinate Systems," Record of the Project MAC Conf. on Concurrent Systems and Parallel Computation, pp. 107-116, (June 1970).

[Pierce-72], Pierce, J. R., "Network for Block Switching of Data," Bell System Technical Journal Vol. 51, No. 6, pp. 1133-1145, (July-August 1972).

[Rodriquez-69], Rodriquez, J. E., "Rodriquez, J. E., "A Graph Model for Parallel Computations," TR-64, Department of EE, Project MAC, MIT, (September 1969).

[Rumbaugh-77], Rumbaugh, J. E., "A Dataflow Multiprocessor," IEEE Transactions in Computers, Vol. C-26, NO. 2, pp. 138-146, (February 1977).

[Sonnenburg & Irani-74], Sonnenburg, C. R., and K. B. Irani, "A Configurable Parallel Computing System," TR82, Department of Electrical Engineering, University of Michigan, (October 1974).

[Thomas-78], Thomas, R. E., "Performance Analysis of Two Classes of Dataflow Computing Systems," M. S. Thesis, Department of Information and Computer Science, University of California, Irvine (January 1978).

[Wittie-76], Wittie, L., "Efficient Message Routing in Mega-Micro-Computer Networks," Proc. 3rd Annual Symposium on Computer Architecture, ACM-SIGARCH, pp. 136-140 (January 1976).

APPENDIX

The Interpreter - The Details of Activity Name Creation

Activity names are generated in such a way that those tokens with identical activity names are exactly those input tokens destined for that named activity. To demonstrate, the symbology

$$a = x|u.c.s.i$$

means that output a of a dataflow operator in a program is producing a token that carries the value x to the destination activity named "u.c.s.i", i.e., to the i th initiation of the operator labelled s found within the procedure code c executing in context u. Thus a is the output name, x is the value of the token produced by that output, and u.c.s.i is the name of that token's destination activity. For example, consider Figure 1 and let

$$a = \alpha|u.c.s.i$$

be the token from output a destined for operator s. (We note that $i \geq 1$ would imply that the code in Figure 1 was within the body of a loop and that this token was produced on the i th iteration of that loop.) Let the remaining token input to activity u.c.s.i be

$$4|u.c.s.i$$

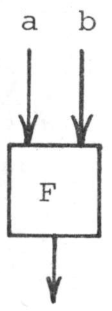
(Note that the output of the constant function 4 has no name.) Thus the output of operator s is

$$4\alpha|u.c.t.i$$

Consider the activity names of these input and output tokens and note that the ith set of input tokens to operator s produces the ith set of output tokens. In fact, observing the input activity name, only the operator label is different on output.

In the following, we generally assume u.c.s.i is the activity being executed, and operator t is the output destination.

1. Functions and predicates: The following more formally characterizes the manipulation of activity names by the class of function and predicate operators. That is, if operator s performs the binary function F and if operator t is a destination of the output of operator s, then



input: $a = x|u.c.s.i$

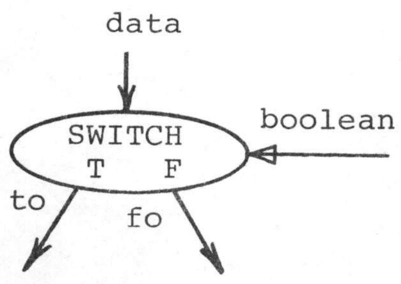
$b = y|u.c.s.i$

output: $F(x,y)|u.c.t.i$

Also, if an output forks to n distinct inputs, then that output will produce n distinct tokens, one for each input. Note that an acyclic circuit composed of any number of interconnected function and predicate operators has the property that the ith set of input tokens produces the ith set of output tokens over that entire circuit.

2. Conditionals: An *if*-expression is composed of the following two additional operators. Note that an *if*-expression behaves as a function box from input to output, that is, the ith set of input tokens produces the ith set of output tokens.

(a) The SWITCH operator:



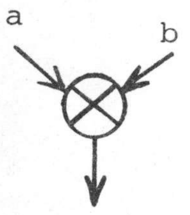
input: data = $x|u.c.s.i$

boolean = $y|u.c.s.i$

output: to = $x|u.c.t_T.i$ if y = true

fo = $x|u.c.t_F.i$ if y = false

(b) The MERGE operator: This operator requires only one of its two inputs, so

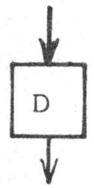


input: a or b = $x|u.c.s.i$

output: $x|u.c.t.i$

3. Loops: We consider these four new operators.

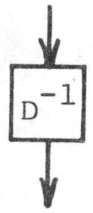
(a) The D operator: This operator is used if and only if there is a cycle in the base language program. A token going through this box is an indication that preparation for the next iteration of the loop is underway. The D box simply increments the initiation count of the token passing through it.



input: $x|u'.c.s.i$

output: $x|u'.c.t.i+1$

(b) The D^{-1} operator: This operator is the inverse of the D operator, and serves to return the initiation count of the token output along the F branch of the SWITCH back to the value 1. Recall that this was the initiation count of the input that originally began the loop.

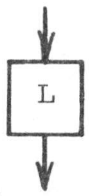


input: $x|u'.c.s.i$

output: $x|u'.c.s.1$

The remaining two operators are L and L^{-1}. We note here that it is possible to define a language without them [Kosinski-73, Dennis-73]; they are key to the unfolding interpreter.

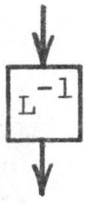
(c) The L (loop begin) operator: The L operator creates a new logical loop by the following manipulation of activity names:



input: $x|u.c.s.i$

output: $x|u'.c.t.1$ where $u' = (u.s.i)$

(d) The L^{-1} (loop end) operator: This operator is the inverse of the L operator. It expects only one token on each logical input and changes the activity name back to the environment to which the output token belongs.



input: $x|u'.c.r.1$ where $u' = (u.s.i)$

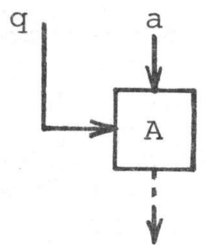
output: $x|u.c.t.i$

Each instantiation of a loop and the corresponding new context u' is called a logical domain. All activities within a logical domain can proceed independent of activities outside that domain and any inner domain at arbitrary nesting levels.

4. Procedure application: The purpose of the activate operator A is to create a new logical domain similar to the way in which the L operator creates a logical domain. The BEGIN operator serves only to distribute arguments passed by A to the procedure body. The END operator gathers results and returns them to the caller, specifically to the terminate operator A^{-1}. Assume the A operator

is labelled s_A and its mate A^{-1} is labelled s_T; let BEGIN and END be labelled b and e, respectively. The value c_Q is the procedure value being applied.

(a) The A(activate) operator: This operator changes the context and passes the argument list copied from a along with a "return address" to the new domain.



input: $a = x|u.c.s_A.i$

$q = c_Q|u.c.s_A.i$

output: $x|u'.c_Q.b.1$

<u>where</u> $u' = u.c.s_T.i$

(b) The BEGIN operator: This operator matches input arguments with formal parameters. Assume operator t is a destination.

input: $x|u'.c_Q.b.1$

output: $x|u'.c_Q.t.1$

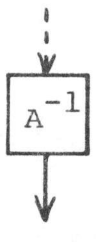
(c) The END operator: This operator simply returns the results back to the calling domain.

input: $x|u'.c_Q.e.1$

<u>where</u> $u' = (u.c.s_T.i)$

output: $x|u.c.s_T.i$

(d) The A^{-1}(terminate) operator: This operator serves only to interface the applied procedure back to the applying procedure. Assume operator t is a destination.



input: $x|u.c.s_T.i$

output: $x|u.c.t.i$

ACKNOWLEDGEMENTS

Several people aside from the authors have contributed to the work that has been described here. In particular, Wil Plouffe has been instrumental in every phase of our research which includes dataflow semantics, ID, and machine architecture. We also thank Bob Thomas for his patient and careful programming of the simulator, and for making it into a really useful and easy to use tool. Without his effort, we could not have progressed as far as we have. Also, to Lubomir Bic for his work on protection which will soon greatly enhance the utility of ID, and dataflow in general. Joe Minne and his work on reduction languages constantly remind us that other systems are also possible, thereby keeping our heads on earth. Last, but not least, thanks to Shirley Rasmussen for reading illegible writing and transforming it into neat pages of print.