A Computer Capable of Exchanging Processing Elements for Time

by

Kim P. Gostelow

and

Arvind

Technical Report #77

January 1976

Revised: January 1976 ((1) corrections to Figure A.5 and A.6 and references to Figure A.5 on page A-3.

(2) minor editorial changes.))

INDEX

Abstract

With the advent of LSI technology large numbers of inexpensive processors have become available, yet our ability to realize the full potential of this technology has fallen short. A primary reason for this failure has been due to an inability to move beyond the sequential control structure of conventional programming languages. The notion of a sequential and/or centralized control for a machine composed of large numbers of processing elements severely restricts the design and complexity of possible computer architectures. We claim that data flow languages (i.e., languages in which the execution of statements is constrained only by the availability of the required operants) provide an alternative base machine language more suited to a technology (such as LSI) which favors distributed processing.

The objective of the proposed research is to formulate and evaluate (by simulation) a machine whose design is based on a new technique of interpreting existing data flow languages. The most significant and distinguishing aspect of this new scheme of interpretation is that it permits a literal exchange of processing elements for computation time, in a very general and mechanical way. An architecture based on the new interpreter will allow a computation to unfold and spread dynamically over a space of processing elements. The machine also has the capability of partitioning itself into disjoining domains of activity, each domain corresponding to the execution of a distinct process.

When the proposed research is carried out, we will have shown that processing elements (space) can be exchanged for time, and that this makes possible a computer organization capable of effectively utilizing large numbers of processors.

I. INTRODUCTION

1. Objective

The objective of the proposed research is to formulate and evaluate a machine capable of dynamically unfolding and spreading a computation over a space of processing elements, and hence a machine capable of utilizing large numbers of processors.

2. Significance

2.1 Direction of the proposed research

For the first time, LSI technology makes available large numbers of inexpensive processors, and thus the capacity to do computation which previously was not possible. However, this capacity has not yet become a capability; in short, we do not know how to utilize this new technology. This proposal is directed towards a possible solution of this problem.

We claim that the problems involved in utilizing the new technology are not related to simply providing an interconnection mechanism, or to designing specialized machines which, for example, can efficiently manipulate arrays. Rather, the problems are due to one of the fundamental premises of computer architecture, that is, the sequential control of von Neumann-type computers [GIMT73].

We note that the primary architectural implication of LSI is a disposition favoring "distributed processing" among many distinct processing elements with essentially autonomous control.

A workshop on Software-Related Advances in Computer Hardware, which was conducted during the Symposium on the High Cost of Software, stated in its final report:

> "Architectural notions that promise to engender greater modularity in computing systems can be expected to have very beneficial effects on the costs of development and maintenance of such systems." ...[Even so]... "Our field has witnessed repeatedly the heated arguments for and against these types of system organization without, so far, being able to settle on their true merits. A thoughtful basic research program will uncover the potential of such system organizations without necessarily requiring the expenditure of outlandish amounts of money." [HCS73,ppll2-113]

To build a system with any reasonable degree of autonomy and distribution, it becomes impossible to think in conventional architectural terms, for example, "instruction streams" and "data streams". Instead, the starting point must be a base machine language founded on an asynchronous control structure, and on this ground we cannot function within the realm of conventional programming languages. In this vein, we repeat here two of the seven research recommendations made by this same workshop on Software-Related Advances in Computer Hardware:

> "The research should aim at:
>
> (1) Reaching a better understanding of the software structure for systems utilizing a number of semi-autonomous processors: chiefly whether this will result in greater modularity ... .
> (2) Means for obtaining better theoretical grasp of the control issues of such configurations." [HCS73,pp 112]

With this need for asynchrony and distribution, we feel that a <u>data flow language</u> is an eminently suited machine-level language for distributed processor machines [GIMT73]. Using as a

base a data flow language already devised by others [D73], we have developed, in an abstract form, a new interpreter [AG75] for this language. This new interpreter dynamically removes all ordering between operations due only to time, while retaining those orderings required by the need for partial results. The primary manifestations of this scheme of interpretation are the automatic unfolding of loops and the simultaneous execution of distinct invocations of the same operation. In terms of hardware, the interpreter permits an exchange of blocks of processors for slices of time, and facilitates the dynamic grouping of processing elements into localized regions of process activity [AG75]. This will allow far greater utilization of a machine composed of distributed processing elements than previously possible, and will permit faster execution of a program as more processing elements are provided. The flexibility of trading processing elements for time in a general and mechanical way is extremely important in view of the technology, and is the primary feature distinguishing our proposed work from others.

2.2 Significance of data flow

Of secondary significance (as far as the proposed research is concerned) is capitalizing on the use of data flow as the base machine language. By data flow, we mean a language in which

(1) an instruction executes when and only when all operands needed for that instruction become available, and

(2) instructions, at whatever level they might exist, are purely functional and produce no side-effects.

There are several advantages of data flow languages over conventional programming languages, such as an absence of variables (explicit memory), and a highly modular program structure [K73,D73]. A particular advantage of data flow is its inherently functional nature. We suggest that many of the difficulties now realized in constructing and verifying the behavior of large systems can be traced to the non-functional behavior of their components. To support this point, we note that principles of structured programming are often advocated in order to produce systems with greater modularity and simpler flow of control. Yet many of the positive aspects of structured programming can be viewed simply in terms of the desire for more functional and less procedural semantics. That is, we are looking for languages with an absence of side-effects both in control structure and in computational results. Also, with both program and machine exhibiting a more functional character, simpler verification of operation and predictability of behavior should result.

3. Method

We propose to use the following method to reach the objective of formulating a distributed processor machine capable of utilizing large numbers of LSI processors:

(1) The new data flow interpreter (described in Section II) forms the abstract basis of the proposed machine and is essentially complete. We wish to move from this abstract basis directly to an architecture capable of carrying out the function of the abstract interpreter. Initial steps already taken in this direction, in the form of a basic architecture, are given in section III. To complete the architecture, we propose the following:

(a) To study the behavior of data flow programs by computer simulation. Section IV states several questions concerning data flow program behavior we need to answer.

(b) To refine and further formulate the architecture to provide support for the expected behavior of programs.

(2) To carry out experiments, by simulation, of the resulting architecture in order to determine:

(a) The performance of the machine.

(b) The degree to which the research objective has been achieved, i.e., how well does the proposed architecture utilize numbers of processors, how effectively are processors exchanged for time, and how well does the mechanism which localizes process activity operate?

Section IV of this proposal gives a schedule stating those problems and questions we have identified, and how and when we expect to answer them.

II. ABSTRACT BASIS OF THE MACHINE

1. Data flow, and the distinction between feedback and non-feedback interpreters

Several data flow languages and schemas have already been devised [e.g., B72, DFL72 and D73, K73, KM69, P62]. Our approach does not directly involve work on data flow languages themselves; rather, we are building on the work of those cited above and are concentrating on the underlying interpretation mechanism. In particular, we have selected Dennis' data flow language [D73] (hereafter called DDF) as the vehicle in which data flow programs are expressed, and we execute these programs according to a new interpreter which is described in detail in [AG75]. We have chosen DDF over other data flow languages primarily because of the availability of theoretical results, the presence of a very good data structure facility, and its relatively advanced state of development.

To contrast the new interpreter with the usual data flow interpreter, we first describe how the usual interpreter executes a data flow program. Consider the segment of a procedure P in DDF shown in Figure 1a. The variously shaped boxes are program statements connected by arcs along which tokens flow. The tokens may be considered to carry all computation values -- both inputs to and results from computations performed in the program statements. Statement P.1 in Figure 1a is a merge and operates by absorbing a control token (true or false value only) on input

C which then specifies which of the remaining two inputs is to be absorbed. In this case, the token at input C is an initial token with value *false*, and thus the token at the False input of the *merge* is absorbed. The output from the *merge* is a copy of whichever of the True or False data inputs was selected. The output of P.1 actually forks in Figure 1a to three statements -- function f, a predicate statement, and function g; thus three tokens are actually output from P.1, each token carrying the value 1 (Figure 1b). At this point statements f, g, and the predicate may compute in parallel since they have no other inputs and thus need not wait for other data. Each of these statements will execute at its own rate. Let us say that f and the predicate have now completed, and furthermore, the predicate has evaluated as *true* (Figure 1c). Now let P.4 execute. Statement P.4 is a *Gate-if-True* statement, and initiates only when both inputs are present. The output of P.4 is a copy of its data input since the control token was a *true* token (Figure 1d); that is, P.4 gates data or destroys data depending upon the control input line. (If the control input had been a *false* token, the data input token would have been absorbed but no output would have been produced.)

At this point (Figure 1d) we have returned to P.1, but this time the data from the True input of the *merge* will be selected since the control input token is *true*. However, note that statement g still has not executed, though it could have at any time. *The fact that there now remains a token at the output of P.1 prohibits P.1 from executing*. That is, there is a feedback

link from all sinks (e.g., statements f, g, and the predicate) to source (statement P.1) such that all sinks must initiate execution before further inputs to these sinks can be made available. (Please note: the above essentially states that there is a maximum input token queue length of 1 for each input to a statement; the problem noted below is not that the maximum is 1, but rather that it is any particular number.)

The usual interpreter, which moves tokens in a data flow language in the manner described above, is called a feedback interpreter. The interpreter proposed in [AG75] operates in such a way that no feedback is present, hence we call it the non-feedback interpreter. The non-feedback interpreter, operating on the program in Figure 1a, will produce any number of inputs to function g. Each of these inputs is, in fact, destined for a distinct invocation of g which we call an activity. These invocations, or activities, under a feedback interpreter are ordered in time. That is, if we label function g in Figure 1 with the name P.5, then the first invocation of g is activity P.5.1, the second is activity P.5.2, etc. However, in the non-feedback scheme the inputs appear whenever they are produced and any number of invocations of g (activities of the form P.5.1, P.5.2, ..., P.5.i, ...) may exist in execution at the same time. Ordering in time has been eliminated where it is not necessary. In [AG75] it is proved that the non-feedback interpreter produces the same computational results as the feedback interpreter.

The architecture we propose is for a machine which implements the non-feedback interpreter.

One of the most significant aspects of the non-feedback interpreter described above and in [AG75] is that it permits faster execution of a DDF program if more processing elements (PEs) are provided. Two examples are detailed in Appendix A which illustrate the generality of processing power-time tradeoff. In order to simplify the argument we have assumed that an unbounded number of PEs are available to us. The details of the proposed architecture described in Section III will make it clear that a bounded number of PEs does not complicate the PE-time tradeoff. If a program could use more PEs during execution than what is currently available, it will execute somewhat slower.

It is our contention that data flow and the non-feedback interpreter are not only a promising approach to the solution of some of the outstanding problems in computer system design [GIMT75,HCS73], but that they also provide ample opportunity for new and novel ideas in machine architecture.

2. Other proposals to implement data flow machines

In this section we mention briefly three proposals by other researchers for data flow architectures. Please note that these systems were not necessarily motivated by the same forces as ours (i.e., the utilization of LSI technology), but each in some way approaches the problem to a degree better than conventional

systems. Historically, designs based on cellular automata [V66] have been attempted for highly distributed machines [H60, C63]. These earlier designs were unsuccessful mainly due to the programming complexity of the base machine and the inadequate technology then available. Since that time we feel that technology has dramatically eased its constraints, and that data flow can simplify the earlier programming difficulties.

To contrast our proposed architecture (appearing in Section III) with the following machines, we might characterize each of them as being an interpreter with feedback; this characterization holds regardless of the particular base machine language used. Again, the machine we propose is an interpreter without feedback, and consequently, at any given time, has an opportunity to call upon many processors which otherwise might not be demanded.

2.1 The Basic Data Flow Processor

Dennis, et al [DM74] have proposed an architecture for the direct execution of programs in a subset of DDF. The basic architecture is shown in Figure 2 and incorporates a memory of many complex cells. Each cell is (essentially) preassigned a statement of the DDF program to be executed. A cell then waits for the input tokens required by the statement, and after all inputs arrive (as determined by a memory controller which constantly monitors all memory cells) the cell with its function code and operands (called an "instruction packet") leaves memory for the functional units. The proper functional unit is then

allocated with the help of an arbitration network, the specified operation is applied to the operands, and result tokens are produced. The result tokens then leave the functional units and head for their individual destination cells back in memory, providing input to some other waiting operation.

2.2 Rumbaugh's Machine

Rumbaugh has proposed a data flow machine [R75] which executes programs in a modified DDF language. The machine (Figure 3) consists of a number of Activation Processors with local memory, each of which can execute a DDF procedure. A procedure executing in an Activation Processor has access to the Structure Memory which holds the data structures implied by the value tokens. Structure Memory is manipulated by controllers which may be shared among several Activation Processors. While the requests of a process to the Structure Memory are being carried out, the process can be swapped out to the Swap Memory by a central scheduler. The Scheduler is a separate hardware unit that controls the allocation of Activation Processors. All requests for the creation and destruction of processes (due to the execution of Apply statements) are also carried out by the Scheduler. Clearly, Activation Processors are a critical resource and high utilization is attained by proper scheduling.

2.3 The Graph Machine

Sonnenburg and Irani [SI74] have proposed a data flow language which is very similar to Rodriguez's [R69]. They have also suggested an architecture for a computer called The Graph Machine to execute their data flow language. The Graph Machine (Figure 4) consists of a large number of processing elements (called operation units), each of which has two inputs and one output. A PE can execute any instruction in the Sonnenburg-Irani data flow language, and all PEs are connected to all other PEs by a control switch of (N^2) simple switches. The simple switches are controlled by a graph memory where program interconnections are stored. PE allocation occurs at graph memory load-time. Miller and Cocke [MC72] have also suggested architectures called configurable computers which are very similar to the Graph Machine.

III. THE PROPOSED DISTRIBUTED PROCESSOR ARCHITECTURE -- SOME IDEAS

1. General - We now describe an architecture to implement the non-feedback interpreter. We begin with the schematic representation of Figure 5, in which the machine is composed of an array of some number of equally powerful processing elements (PEs), each attached to a communication system and to a memory system. During program execution, each PE is dynamically allocated, that is, given an activity name. The PE then performs the computation corresponding to that activity, outputs tokens destined for other activities, and finally becomes free by deallocating itself. PEs may be added to the array to provide increased computational power, or deleted from the array, as desired.

As shown in Figure 6, each PE is composed of four subsections which perform the following functions:

(1) activity name recognition

(2) token input and output

(3) computation

(4) memory system interface.

The token input and output subsection of each PE is interfaced with the communication system. The communication system carries the tokens, in the form of messages, between the PEs. Each token carries with it the activity name of its destination. The communication system accepts tokens output from a PE and circulates the tokens among all PEs. When a match is found

between the destination activity name on the token and the current activity name of some PE, that PE accepts the token as input. When all necessary tokens have been received by the PE, it begins the computation designated by the activity name. When the computation terminates, output tokens are injected back into the communication system by the PE and the PE deallocates itself by clearing its activity name. The PE is then available for reallocation to another activity.

2. The communication system - A key aspect of the system described above is the communication system through which tokens circulate and visit the PEs. Figure 7 shows our initial approach to the communication system, and the location and interconnection of PEs to the system.

The PEs are grouped in columns (labelled i-1, i, i+1 in Figure 7a) such that all PEs on the same column share a local bus. This bus is the internal bus and is present so that all PEs lying along one column may communicate tokens amongst themselves very quickly. Thus, a PE that produces a token places that token (which includes its destination activity name) on the bus. All allocated PEs constantly monitor their internal bus looking for input tokens. If the destination activity has been allocated a PE which lies on the same bus, then the token will be absorbed immediately. Thus local communication is easily accomplished. Now we must account for communication to and from other columns of PEs. Communication from one column to another is handled by a ring bus encircling the columns. The top of the ring is detailed

in Figure 7a as the double-width line passing from right to left through the switches T_i. The bottom of the ring passes from left to right through the switches B_i. Finally, the buses are closed by <u>firewalls</u> at each end to complete the ring: the left end is closed by a firewall at column 1 by switch T_1 forcing all token traffic to go down the line labeled L_1; the right end is closed by a firewall at column n by switch B_n forcing all token traffic to go up the line labelled R_n. The arrangement of the ring and the various switches are shown in Figure 7b (the PEs and internal buses are not shown).

With the ring bus, tokens may circulate and visit each column looking for their respective destinations. Switch T_i of each column contains an associatively addressed <u>activity name table</u> which records the activity names of each allocated PE within that column. (The idea of an activity name table is similar to the process nametables maintained in the ring interfaces of the Distributed Computing System [F73].) When a token enters switch T_i, its destination name is compared with the activity names in the table. If the destination name is present, the token is diverted onto the internal bus where the waiting PE will absorb it. If the destination name of the token entering switch T_i does not match any of the allocated activity names, then the token continues on to the next column. One may think of the token as circulating around the ring as many times as necessary until a match is found. Switch B_i is used to put a token produced by a PE within a column onto the ring when no PE on the internal bus accepts that token.

We envision a ring bus implemented by shift registers, and thus both token storage and token movement are provided by the same simple device. Also, new columns of PEs may be easily attached by joining onto the ring. The amount of storage (and thus the length of the ring and the communication delay) may vary dynamically as the number of circulating tokens grows and diminishes over time. However, significant parameters such as token size, token delay, and token bit rates remain unknown, although estimates of these parameters indicate feasible token communications can be obtained with current and near-term technology.

For example, let there be an average requirement of n input tokens per PE, and let a token be k bits long. Also, let there be N PEs per column in simultaneous operation, a fraction f of which require communication with the ring. Then if the average wait and execution time of a PE is T, the ring must support a mean rate of $(nk/T)Nf$ bits per second. A crude estimate for the worst case is n=3 input tokens per PE, k=100 bits per token, N=8 PEs in simultaneous execution, a fraction f=0.5 of which require communication off the column, and a PE wait and execution time of T=100 microseconds. This results in a bit rate of 12 megabits per second, an entirely reasonable figure for near-term LSI shift register speeds. Better estimates for shift register requirements can be obtained by simulation and will determine the shift register characteristics.

The communication system is discussed in further detail in section 5: Execution domains.

3. PE allocation and deallocation - Allocation of a PE in some column to an activity P.s.i is dynamic during execution, and is noted in the activity name table contained in that column's switch T. Allocation is distributed in that no central controller selects a PE and assigns it an activity name. Rather, one of the inputs to each statement of a program is selected at compile time to be the allocation input, and any token sent to that input is marked as an allocation token. Thus there is one and only one allocation token per activity.

The presence of an allocation token in the system implies that a free (unallocated) PE must be found and assigned the activity name appearing on the allocation token. However, we have little experience with data flow programs and their behavior, and hence only some understanding of what might constitute a good allocation scheme. One heuristic which we feel is good states that PEs should be allocated whenever possible on the same column in which the allocation token was produced; the second best position is an adjacent column, and so on. That is, PEs should be grouped together for close token interaction. The heuristic as applied to procedure calls is discussed further in section 5: Execution domains.

4. Deadlock - There are two limited resources in the system so far discussed that could be sources of deadlock, unless some precautions are taken. These two resources are (1) the finite

number of PEs, and (2) the finite storage capacity of the ring. Even if only one of these two resources is finite, a deadlock is still possible. We describe a deadlock situation when the communication system is assumed to have infinite storage capacity but finite delay, and the number of PEs is limited.

(1) Suppose all the PEs are waiting for an input token, that is, all PEs have been allocated but none have begun execution. Also suppose that tokens needed by the allocated activities are not present in the communication system either, (otherwise there would be no deadlock). This is possible only if we assume that the tokens for some activity C that will produce the needed tokens are all present in the communication system, (otherwise the program itself is malformed [AG75]). But the activity C cannot be carried out because no PE is available.

To avoid this deadlock possibility, we have instituted a rule which holds for all activities or functions with which we work. This rule states first that any PE which has been allocated (and thus holds at least one input token) may deallocate itself at any time simply by clearing its activity name and returning to the communication system all the tokens which it has absorbed up to the time of deallocation. Secondly, the rule states that any PE which has received all its input tokens and has initiated execution will always be able to go to completion. (This rule has implications for the memory system discussed below. Any request to the memory by a PE must be satisfied without requiring any more PEs. In other words, the memory system and the PEs must function independently.) Thus, for example, each PE may have a randomly set time-out period which begins at allocation, and if the time-out expires before initiation, then the PE deallocates itself and becomes available for reallocation. In this way there will always be at least one

PE available to carry the computation forward, and thereby avoid deadlock. Actually a scheme slightly more sophisticated than the random time out is needed to ensure a deadlock-free system. One must ensure that no activity is excluded indefinitely from being allocated.

(2) Suppose the communication ring is full, but no token on the ring can find a destination PE. Thus, there must be an activity C in execution, which when it teminates will output a token which causes allocation of a new activity. If the allocation cannot be made on that column (for example, all the PEs are in execution) then the token output from C must enter the ring. But the token cannot enter the ring since the ring is full.

Also, the scheme described in (1) above to avoid deadlocks will work only if essentially unlimited capacity is guaranteed in the communication system. We propose to do this by providing a memory access port to switches T and B. A switch may take some tokens out of the ring if the ring overflows and it may put these tokens back onto the ring at some appropriate time later. Taking tokens out in this fashion increases the storage capacity of the ring without increasing the communication delays for most tokens.

5. Execution domains - Section 3 above on PE allocation and deallocation briefly discussed a heuristic for allocation: that an activity should be allocated a PE near the PE which produced the allocation token. This heuristic attempts to realize a belief about data flow programs. The belief is that there is locality in data flow programs - that statements (or activities) which are "near" in terms of program graph distance, should also be near in execution. Specifically, activities within some

procedure would generally be nearer to themselves than to any activity outside that procedure. In fact, the only interactions which any statement within a procedure has with any statement outside that procedure are the passing of input parameter tokens to the procedure and the returning of result tokens from the called procedure back to the caller.

We have selected the procedure as a basic program element with which the machine's architecture is to interact. The machine is to partition itself into disjoint domains of localized process activity during execution. Each domain corresponds exactly to a single invocation of a procedure. We believe the existence of such domains as machine partitions will increase machine speed by isolating those sets of tokens which would not usefully be mixed, and by reducing the token destination search space to a particular subportion of the entire machine. (We note that a complete activity name as defined in [AG75] actually contains one more field 'u' to denote the context from which a procedure is called. Hence, a complete activity name has the form u.P.s.k. However, the partitioning of the machine solves the problem of the length of token activity names by knowing that the context is the same for all activities within a given domain. Hence, the u.P part of the activity name will be implied by the execution domain, and only the s.k part of the activity name will be carried by the token within the domain [AG75].)

For example, Figure 8 shows a procedure at a particular point in execution as indicated by the presence of tokens at various statement inputs. Figure 9a shows the machine and the fact that procedure P lives within a particular area bounded by a left firewall L_p and a right firewall R_p. The switches T and B are responsible for constructing and maintaining the firewalls, and they do so in two ways. First, the firewalls force all tokens with destination in procedure P to remain on that portion of the ring surrounding P. Second, any token which does not have destination in P, but which finds itself inside P, is allowed to continue and to pass on through P. The firewalls thus create small subrings which circumscribe collections of PEs with high internal token interaction, yet allow tokens going to other domains to pass quietly through. A schematic representation of switches T and B is shown in Figure 10.

Now let the statement apply Q be performed in P, where procedure Q is some data flow procedure. As detailed in [AG75], an apply Q statement is actually two separate activities: an activate Q and a terminate Q activity, both of which execute within domain P. An activate causes two actions to occur:

(1) domain Q is created,

(2) the input tokens to apply Q are sent as inputs to procedure Q in the newly created domain.

Returning to Figure 9a we see domain P and the remaining unused portions of the machine. All unused portions are known, and all continguous columns of unused PEs form free domains from which free PE columns are allocated to enlarge existing domains

or to create new domains. In domain P in Figure 9a, the apply Q statement is about to be executed and domain Q created. Domain Q is created when the activate Q portion of the apply produces a create domain Q token as output (along with the input parameter tokens for Q). The create domain token then leaves domain P. When the token reaches a free domain of sufficient size, the free domain is partitioned into the new domain Q, and a free domain of lesser size.

In Figure 9b two procedures and their domains now exist. Next, it is necessary to transmit input parameter tokens from P to Q. Input parameter tokens to Q may be passed out of P simply by marking their destination as being in Q, and hence outside of P. Given the rule above that tokens which find themselves inside a region which is not their destination region are simply allowed to pass through, these input parameter tokens will be released from the domain P and they will find their way to Q. Once inside domain Q, these input parameter tokens appear just as any other token inside Q would appear.

The above has described the creation of a domain and the passing of input parameter tokens into that domain. The inverse of the above occurs at termination of the called procedure Q. First, result tokens are passed back to the terminate Q portion of the apply Q statement in P. Then the terminate activity outputs a destroy domain Q token which will find domain Q and destroy it by adding the PEs in it to a free domain.

There are several unresolved points concerning procedure domains. How do domains acquire more PE resources (how do firewalls move from one column to another) when it would be useful for them to do so? What would comprise a good scheduler for allocating domains? Is there a way to reduce the overhead in procedure calls, possibly by specifying some maximum number of concurrent invocations of a procedure and reuse the domains?

These and other questions require further investigation to be resolved. However, we feel that the notion of an execution domain is viable, and brings an intuitive feeling for how procedures operate into the basic structure of the machine.

6. The memory system - Values in data flow are carried by tokens. However, these values can have rather complex structure (such as a tree) and be of significant size. For the purpose of reducing the quantity of information carried by a token, [D73] presents a technique whereby data values can instead be maintained in a memory and only pointers to those values need be present on the tokens themselves. Based upon pure LISP, the technique also allows garbage collection to be handled by simple reference count techniques. (This can be done since data is never modified after it is created, and is destroyed when all tokens carrying (pointing to) that data have been input to their destination PEs.)

Since memory is present only to reduce the bit rate which the communication system would otherwise have to support, we require the memory system itself to be responsible for all name

and storage management. Also, since any PE may perform any data flow statement, all PEs must be physically able to gain access to all of memory.

The above are the reasons for, and the logical requirements of, the memory system. We have not as yet fully determined a structure for the memory system discussed below. however some points have emerged. First, we expect a "locality" effect, in that data moves from a producer PE to neighboring consumer PEs, and we expect many such activities to occur simultaneously. Thus we feel a use for a distributed memory composed of many independent units capable of functioning simultaneously, and a given memory unit must be accessible by any PE. Second, a given memory unit should be closely associated with one or more PEs of a given column, with newly created data values placed as closely as possible to the PE causing the creation of that data. Thus, due to the expected close interaction among the PEs within a single column, any PE on a given column should have faster access to the memory unit(s) associated with that column, than any PE not on that column. Lastly, we expect to be able to tolerate some access delays that are longer than in current conventional machines.

Clearly, several questions remain and can be answered only with research into actual data flow program behavior patterns. Such questions are: How many memory units should be highly accessible by a given PE? Under what circumstances is it better to copy a structure to another area of the machine rather than

suffer repeated access delays? Given the experience with Holland machines [H60,C63] and C.mmp [FSS73], what kind of bussing structure will allow all PEs access to any memory unit, but without obstructing other PE memory unit accesses?

7. Toroids

Lastly, we intend to unify the machine's structure from that given in Figure 7a. As shown in Figure 11a, PE columns 1 and n will be connected together by the ring bus so that the "ends" of the machine itself will be defined only by firewalls (which can also move). Also, switches T_i and B_i in each PE column i, can be molded into a single physical device. Figure 11b shows that the machine's resulting logical form is a toroid.

IV. SCHEDULE OF PROPOSED WORK

We propose to carry out the work by building a simulator for data flow programs, measuring the behavior of those programs, and orienting the architecture of the machine to function in concert with that behavior. As architectural decisions are finalized they will be incorporated into the simulator to study their effects on program execution. When all components of the architecture are complete, we will have a simulator of that architecture. Our proposed schedule of work is as follows:

Months 1-3: This first period will be devoted to building a data flow program simulator, and to gathering a collection of test data flow programs on which measurements will be made (two simple candidates, quicksort and matrix multiply, appear in Appendix A).

Months 4-6: During this period we plan to measure the behavior of the test data flow programs, assuming a grid of processing elements and a basic execution time for each data flow operation. No particular communication system nor memory system will be assumed; thus the simulator will initially function just as the theoretical interpreter functions, and the measures will reflect the properties of data flow itself. We plan to measure:

(1) data structure access and creation behavior

(2) token density and flow patterns

(3) the effect of varying the size of a "unit of computation" (activity) on items (1) (2), above.

This work will impact data flow language design itself. Initial efforts in determining a "unit of computation" produced the work covered in Appendix A, and we anticipate advances in data flow language design to grow out of these beginnings. That work which is easily accomplished and which will contribute to the design of data flow languages, we plan to do here.

During this period, the above information will be used to determine the functional power to be given to a processing element. The results of these experiements will also be used during the following period.

Months 7-12: (1) During this phase, the major goal is to determine an architecture for the memory system. The design will reflect expected near-term LSI capabilities, and the characteristics of behavior of data flow programs determined during Months 4-6. We then plan to impose the memory system upon the data flow simulator, and to measure the effects of the memory system on program execution. Of particular importance is the memory allocation scheme, but we expect that the "locality" scheme described in Section III (or some variant thereof) will prove to be good. Also, the effects on performance of some parameters concerning the memory system must be determined; these parameters are

(a) density of memory units distributed throughout the machine

(b) memory system bandwidth

(c) the extent of a processing element's "local neighborhood" of memory, i.e., the memory to which it has direct access.

(2) A second goal during this period will be to determine a good PE allocation scheme. (Again, we anticipate that the "locality" scheme described in Section III will be appropriate.) Then, when the allocation scheme is decided, it will be possible to more accurately determine parameters concerning the communication system bandwidth and storage capacity.

We feel that the memory system will be the most difficult and sensitive area of the machine, while we anticipate little difficulty with the token communication system.

Months 13-18: During this phase, we plan to incorporate the memory and communication system designs into the simulation. We will then have a simulator of the architecture and will be able to evaluate the performance of the machine. This period will also be devoted to determining system bottlenecks and improving those subsystems where difficulties might arise. To determine machine performance, we propose to measure the following:

(1) The overall rate of computation of the machine, assuming basic component speeds consistent with near-term LSI capabilities.

(2) The utilization of the machine's processors based upon the number of processors in execution concurrently.

(3) Evaluation of the degree to which processors are exchanged for time. This measure is somewhat complex, and is best accomplished by comparing two executions of the same data flow program - one execution in which the processor/time exchange mechanism is inhibited, and one execution in which it is fully operative.

(4) Evaluation of the effect of the mechanism which localizes process activity into disjoint domains of execution. This measure will be accomplished (similarly to (3) above) by comparison of two executions - one execution in which the mechanism is inhibited, and one in which it is operative.

Months 19-24: We propose to investigate the requirements and resource demands which a kernel operating system might place upon the machine, where the kernel contains basic functions such as creation and destruction of processes, protection, interprocess communication, and input/output. We feel that this work is necessary to gain a clearer view of the machine and how it might function in a real computing environment. Some of these basic functions appear straightforward (e.g., creation and destruction of processes) while others are still problems even in conventional machines, but perhaps more easily solved in the proposed machine (e.g., protection).

V. REFERENCES

[AG75] Arvind and K. P. Gostelow, A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture, TR 72, Dept. of Information and Computer Science, University of California, Irvine, November 1975.

[B72] Bahrs, A., Operation Patterns (An Extensible Model of an Extensible Language), Symposium on Theoretical Programming, Novosibirsk, USSR, Aug 72, (pp 217-246).

[C63] Comfort, W. T. "A Modified Holland Machine" Proceedings FJCC, 1963, (pp 481-488).

[D73] Dennis, J. B., First Version of a Data Flow Procedure Language, MAC TM 61 (originally published as Computation Structures Group Memo 93, Nov 1973), Project MAC, MIT, May 1975.

[DFL72] Dennis, J. B., J. B. Fosseen and J. P. Linderman, Data Flow Schemas, Symposium on Theoretical Programing, Novosibirsk, USSR, Aug 72, (pp 187-216).

[DM74] Dennis, J. B., D. P. Misunas, A Preliminary Architecture for a Basic Data Flow Processor, The 2nd Annual Symposium on Computer Architecture, Houston, January 1975, (ACM-SIGARCH Vol 3, No. 4, Dec 74) (pp126-132).

[F73] Farber, D. J., et. al., The Distributed Computing System, Proc. Seventh Annual IEEE Computer Society International Conf., Feb 1973, (pp 31-34).

[FSS73] Fuller, S. H., D. P. Siewiorek, and R. J. Swan "Computer Modules: An Architecture for Large Digital Modules" Proceedings of First Annual Symposium on Computer Architecture, Dec. 9-11, 1973, University of Florida, (pp 231-237).

[GIMT74] Glushkov, V. M., M. B. Ignatyev, V. A. Myasnikov and V. A. Torgashev, Recursive Machines and Computing Technology, Information Processing 74, North-Holland Publishing Company, Stockholm. Aug 1974, (pp 65-70).

[H60] Holland, J. C., "Iterative Circuit Computers" Proceedings Western Joint Computer Conference, 1960, (pp 259-265).

[HCS73] Gagliardi, Ugo O., et. al. "Software-Related Advances in Computer Hardware" Proceedings of a Symposium on the High Cost of Softare, Monterey, California, J. Goldberg, ed., Stanford Research Institute Project 3272, Sept. 17-19, 1973, (pp 99-119).

[K73] Kosinski, P. R., A Data Flow Language for Operating Systems Programming, Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices Vol. 8, No. 9, Sept.

1973. (pp 89-94).

[KM69] Karp, R. M. and R. E. Miller, Parallel Program Schemata, JCSS Vol. 3, No. 2, Nov 1969, (pp 147-195).

[M75] Misunas, D. P., Deadlock avoidance in a Data-Flow Architecture, Proc. of 3rd ACM-IEEE Milwaukee Symposium on Automatic Computation and Control, Apr 1975. (pp 337-343).

[MC72] Miller, R. E. and J. Cocke, Configurable Computers: A New Class of General Purpose Machines, Symp. on Theoretical Programming, Novosibirsk, USSR, Aug 72, (pp 285-298).

[P62] Petri, C. A., Communication with Automata, RADC-TR-65-377, Vol 1, Griffiss Air Force Base, New York 1966. (originally published in German: Kommunikation Mit Automaten, University of Bonn, 1962)

[P75] Petri, C. A., Keynote Address, First Conference on Petri Nets and Related Methods, MIT, July 1975, (proceedings to appear 1976)

[R69] Rodriguez, J. E., A Graph Model for Parallel Computations, TR-64, Dept. of EE, Project MAC, MIT, Sept. 1969.

[R75] Rumbaugh, J. A Parallel Asynchronous Architecture for Data Flow Programs, Ph.D. Thesis (MAC TR 150), Dept. of EE, MIT, May 1975.

[SI74] Sonnenburg, C. R. and K. B. Irani, A Configurable Parallel Computing System, TR 82, Dept. of EE, University of Michigan, Oct. 1974.

[T74] Turn., R., Computers in 1980s - Trends in Hardware Technology, Information Processing 74, North-Holland Publishing Company, Stockholm, Aug 1974, (pp 137-140).

[V66] von Neumann, Theory of Self-Reproducing Automata, A. W. Burks, ed. University of Illinois Pres, 1966.

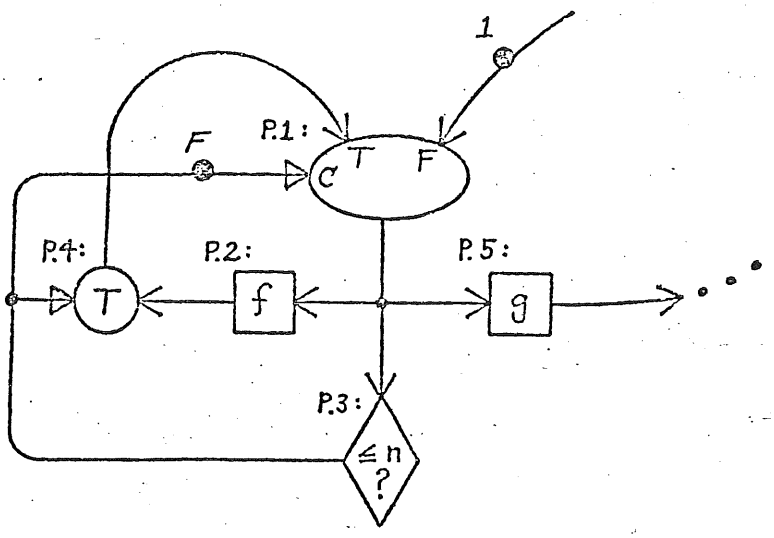


Figure 1a

A portion of a data flow procedure P in its initial configuration

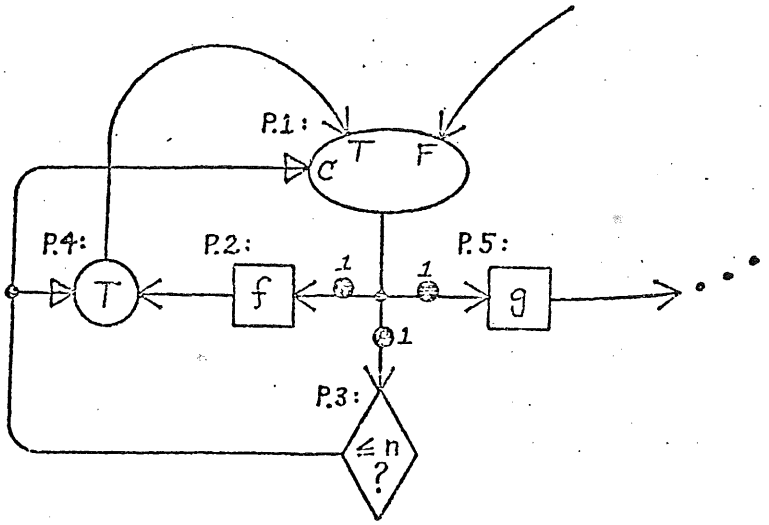


Figure Ib

Status of the program after P.1 executes

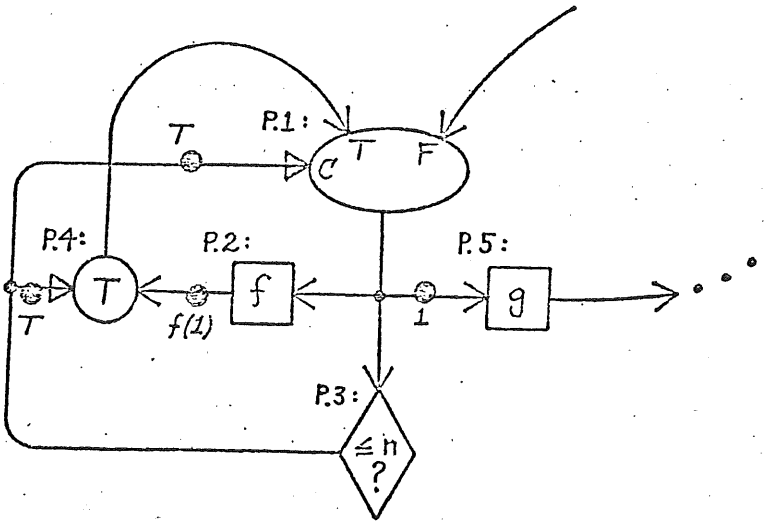


Figure 1c

Statement f and the predicate complete execution

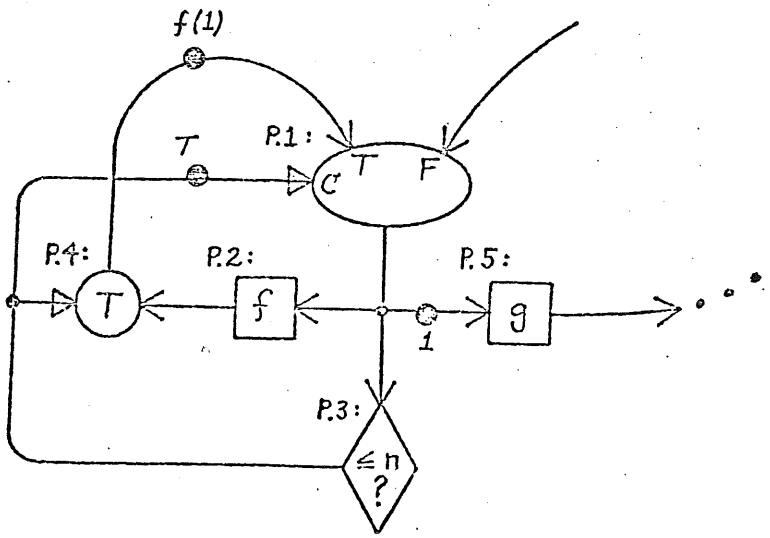


Figure 1d

The gate produces its output

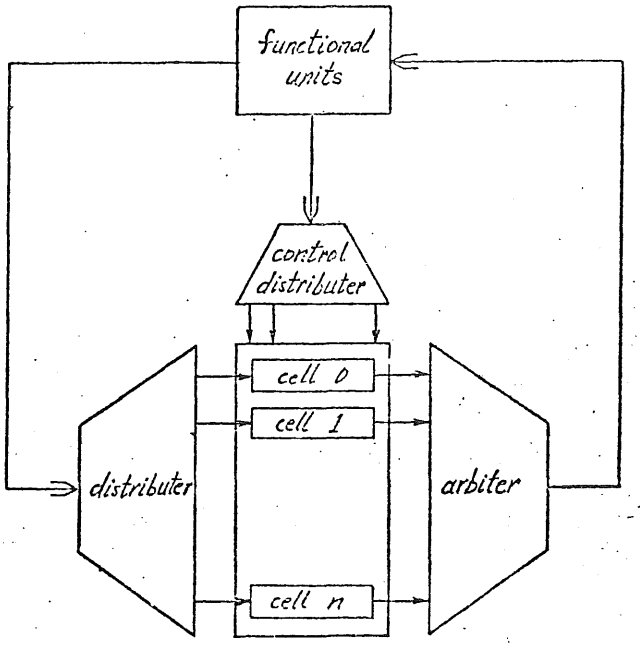


Figure 2

Basic Dennis-Misunas data flow machine

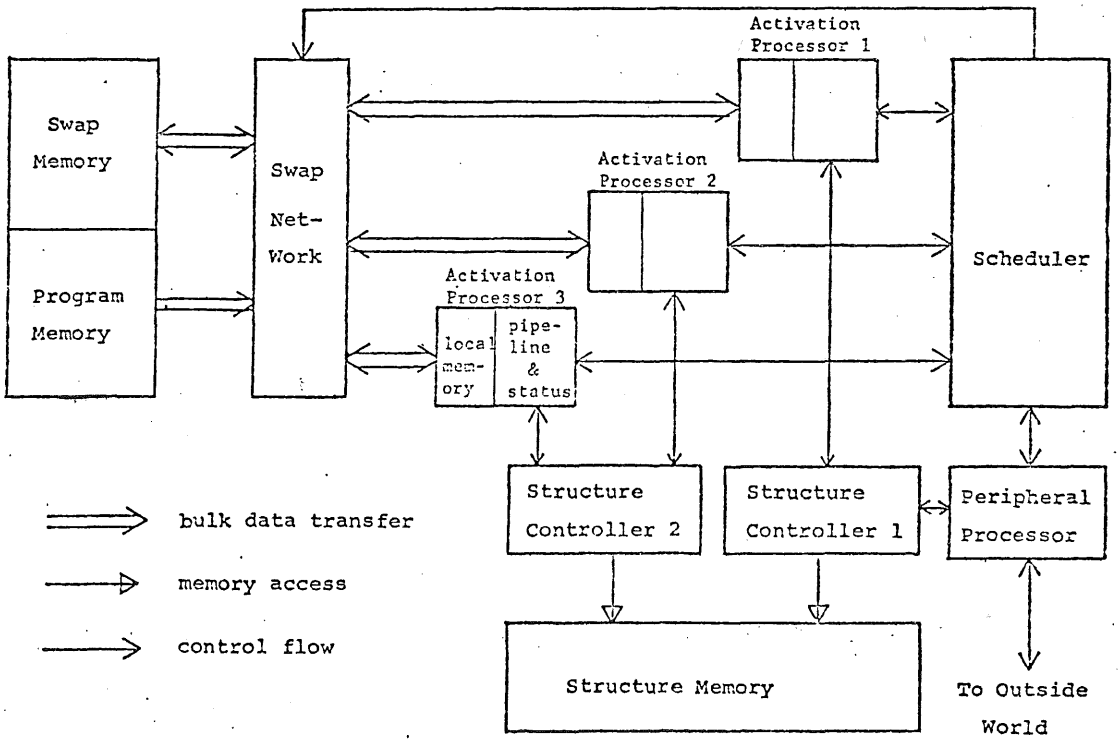


Figure 3

Rumbaugh's machine (figure from the thesis [R75])

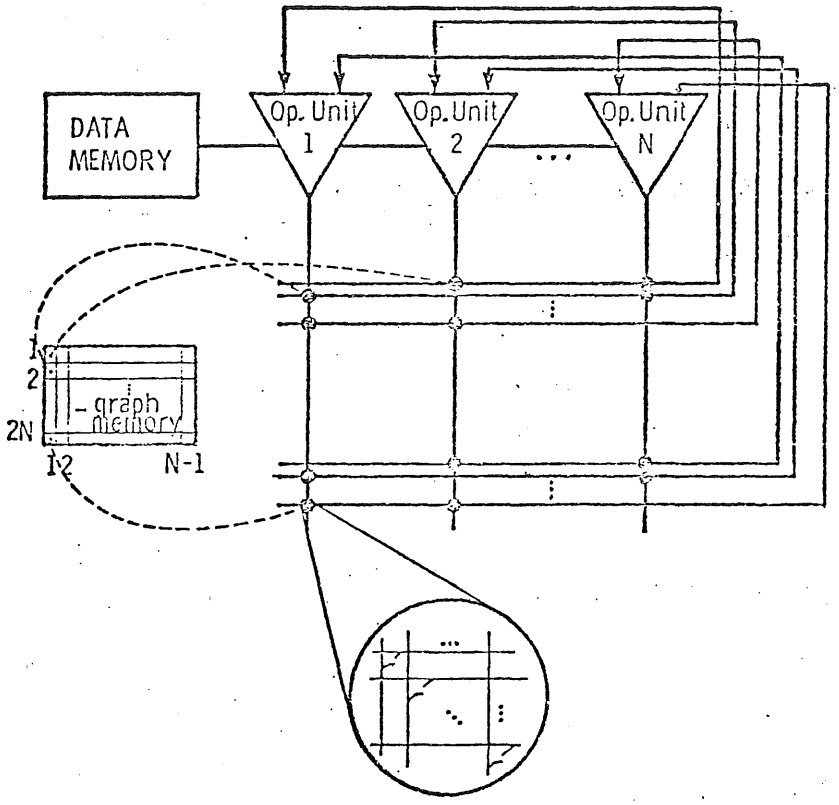


Figure 4

Sonnenberg-Irani machine (figure from the thesis [SI74])

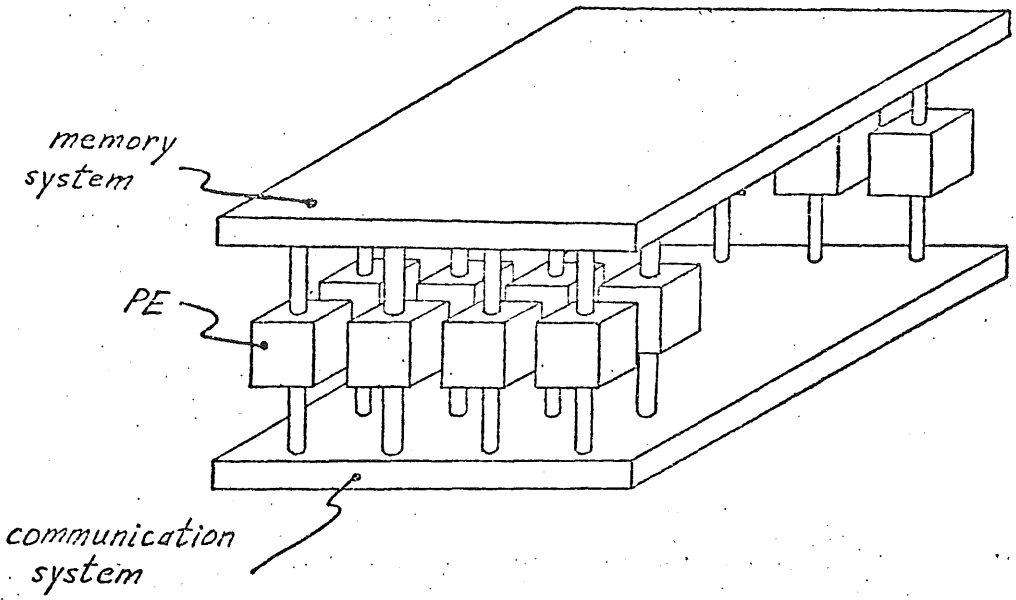


Figure 5

Basic view of proposed data flow architecture

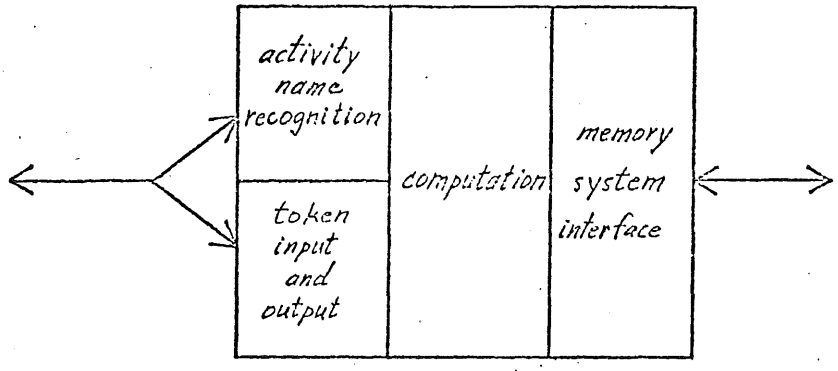


Figure 6

The four subsections of a PE

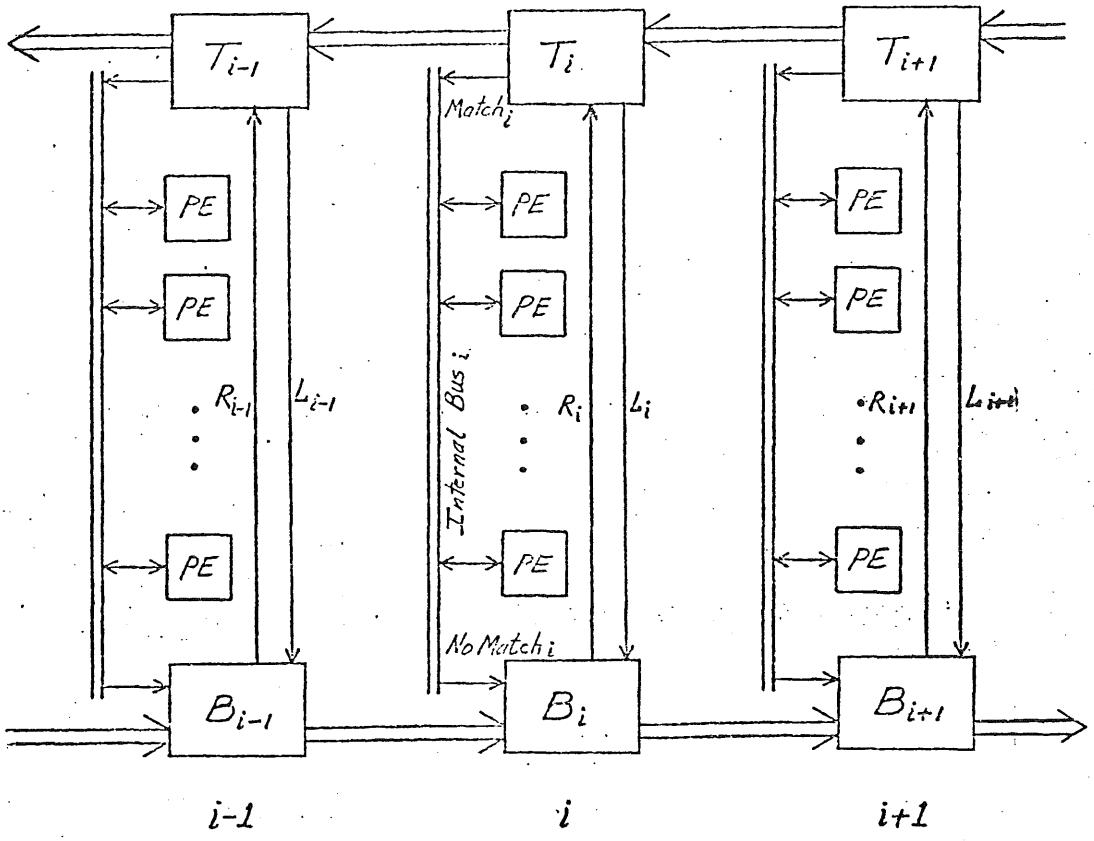


Figure 7a

The communication system

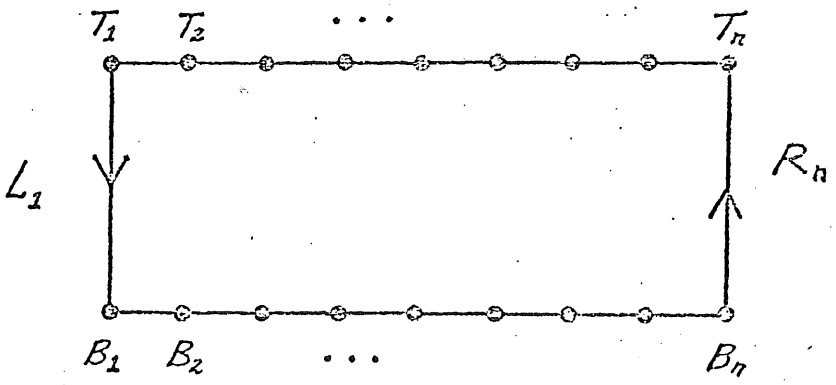


Figure 7b

The ring bus encircling the PE columns

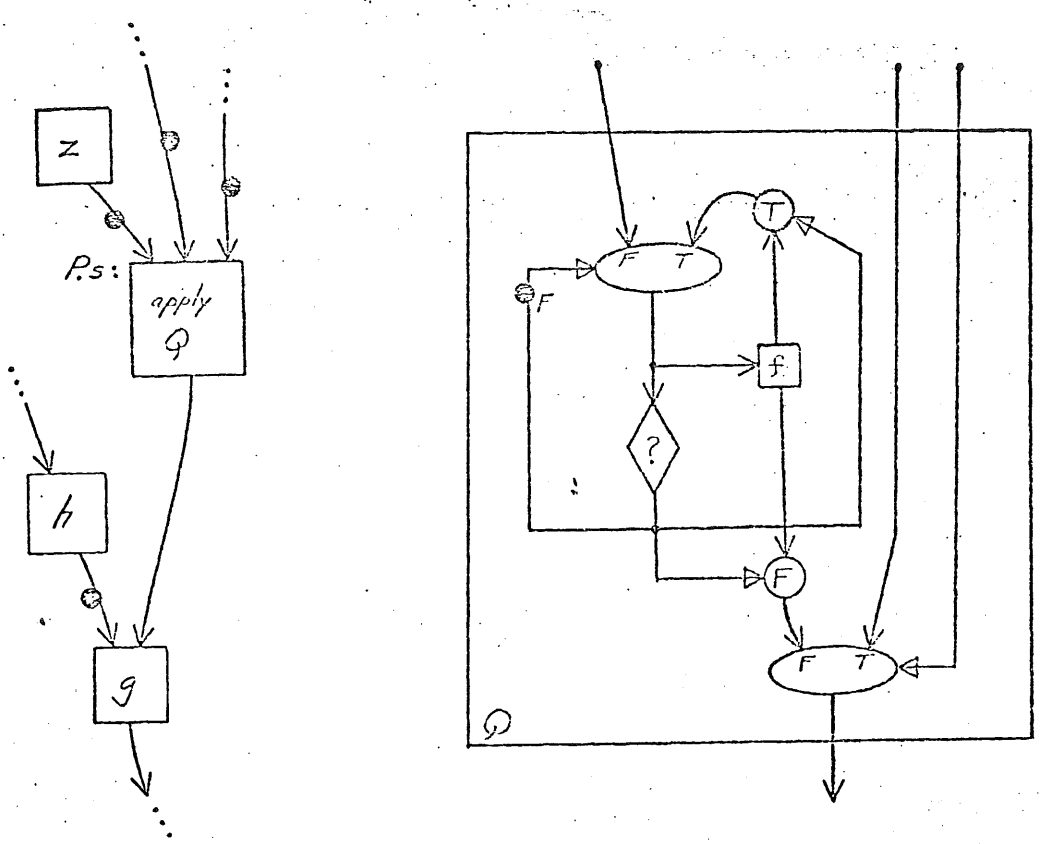


Figure 8

An apply statement in procedure P calls procedure Q

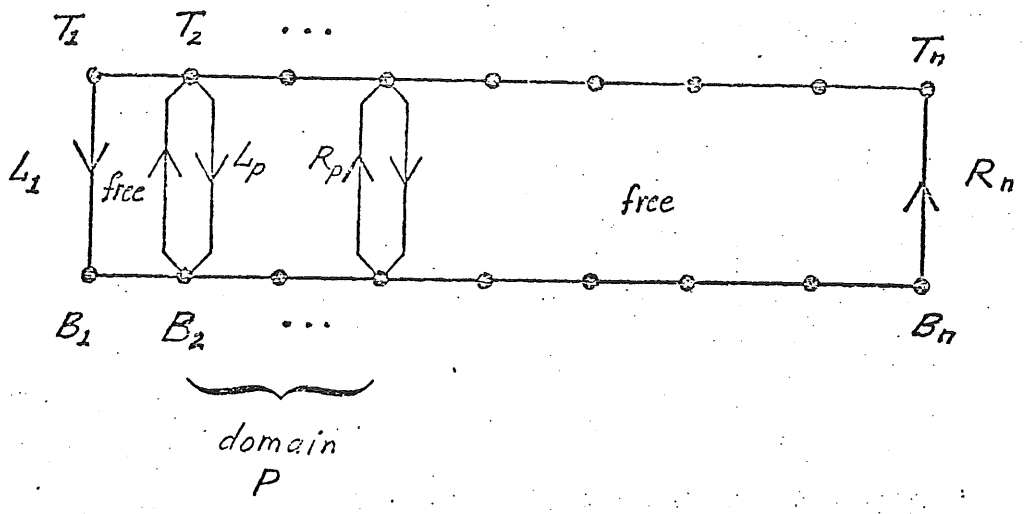


Figure 9a
Domain P within the machine

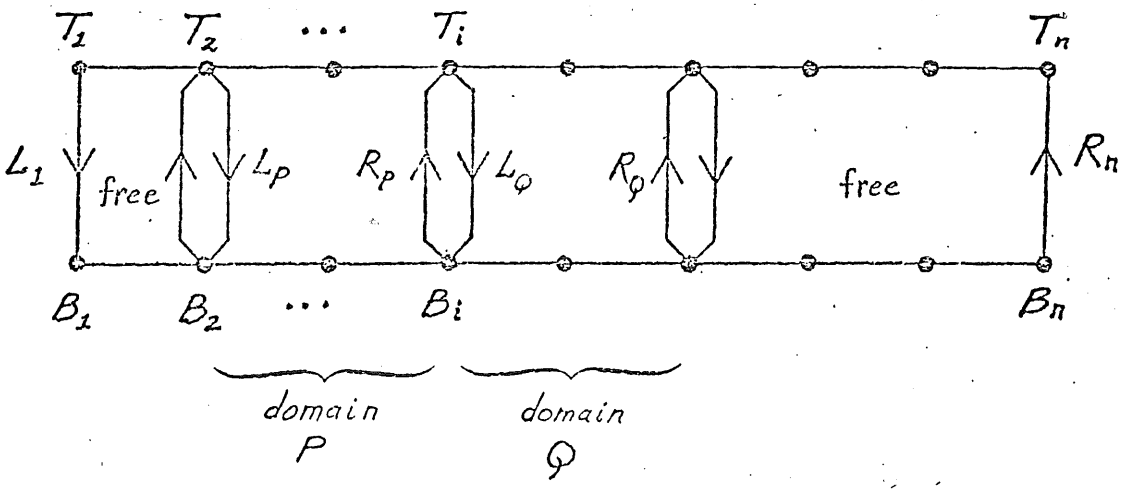


Figure 9b
Domain Q is created by P

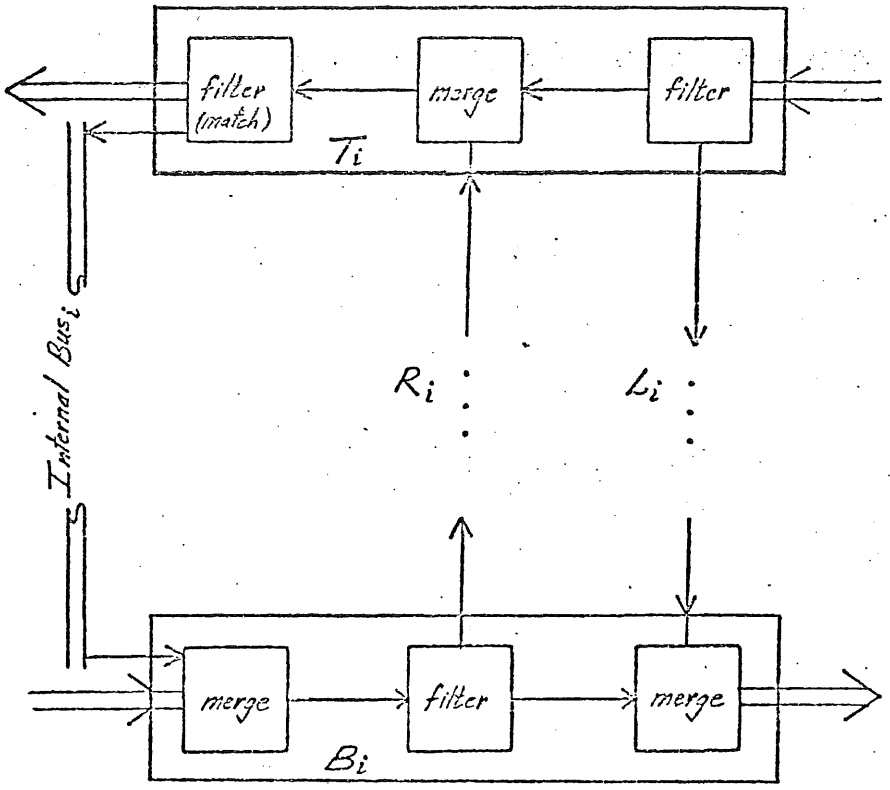


Figure 10

Internals of the switches T_i and B_i

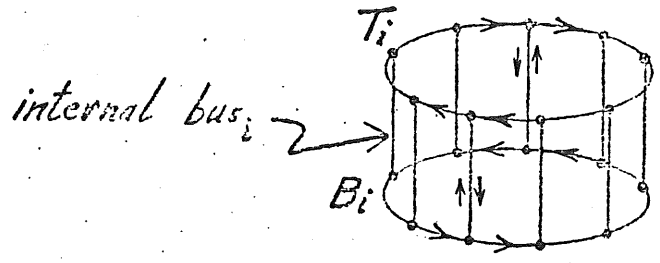


Figure 11a
Connecting the ends of the machine together

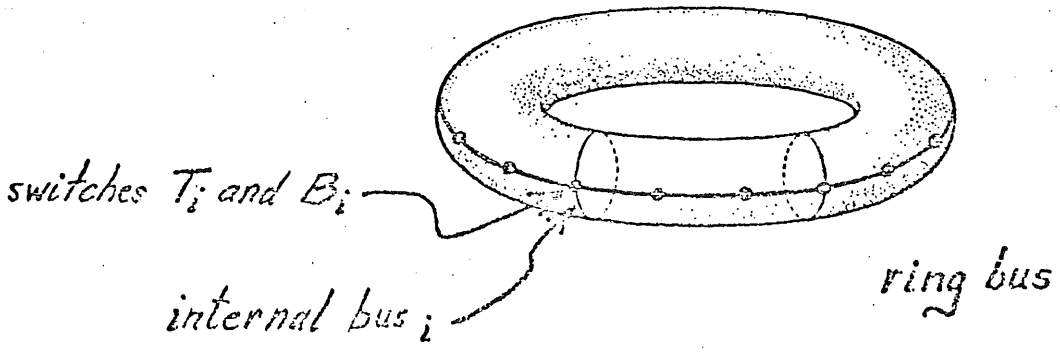


Figure 11b
The resulting toroidal architecture

APPENDIX A

Two Data Flow Programs and Their Analyses

A.1 Macros in DDF

Before we give the sample programs, we will define two new primitive operators in the DDF language. These primitives can also be viewed as macro, that is, an equivalent group of DDF statements exists which can be substituted for each occurrence of these operators. The upper portion of Figure A.1 shows a DDF program which is often a part of larger DDF programs. The equivalent program using the macros (yet to be defined) is shown in the lower portion of Figure A.1.

Essentially, on arrival of an input token v, n tokens with values f(v), f(f(v)), ... are produced by loop 1. After each of these tokens is transformed by some program g they are input to loop 2. Loop 2 accepts these n tokens and forms one result token according to function h. A very common situation is when f is the identity function and h is the Σ operator.

The following are detailed definitions of the two macros:

(1) time-Distributer (t-Dist): There is a function f, a constant n, and an input v associated with each t-Dist. Once the input v has been received, then n tokens with values f(v), f(f(v)), ... will be produced, ordered in time. When a t-Dist is executed under the feedback interpreter, an output token is produced on an arc only after the previous token output on that arc has been absorbed. However, under the non-feedback

interpreter all n tokens can be produced and output (almost) simultaneously with activity numbers u.P.s.k, u.P.s.k+1, ... u.P.s.k+n-1. (The value of k will depend upon the activity number on the input token v; for example, if the input token has the number u.P.t.i, then k=(i-1)*n+1.)

Figure A.2 gives the firing rules and the DDF program equivalent to t-Dist. If f is an identity function then t-Dist will simply produce a sequence of n copies of v. Another very useful f function is "+1" which will produce a sequence of tokens with values 1, 2, 3, ..., n assuming the initial input v had value zero.

(2) time-Collector (t-Coll): This operator performs, in a sense, the inverse of t-Dist. t-Coll also has a function f and constant n associated with it. It has an input s_o (the initial value), and an input port v on which n tokens, ordered in time, with values v_1, v_2, ..., v_n are received. Under the non-feedback interpreter tokens with activity numbers u.P.s.k., u.P.s.k+1, ..., u.P.s.k+n-1 will be needed for one execution of t-Coll. (For the i^{th} initiation of t-Coll, the value of k would be (i-1)*n+1.) After all the input tokens have been received, an output token with value $f(v_n, f(v_{n-1}, \ldots f(v_2, f(v_1, s_o))\ldots))$ is produced. Two useful functions f for this macro are "Σ" and "append". Figure A.3 gives the DDF program equivalent to t-Coll.

The macros t-Dist and t-Coll not only simplify the writing of programs, but also result in less token traffic and demands on PE resources. It is easy to see that a single PE could be given the capability to execute t-Dist. A PE executing t-Dist under the non-feedback interpreter will produce n tokens in roughly c_o+rn units of time, where c_o is the set-up time of a PE and r is the additional time needed to produce a token. For simple functions (e.g., f=identity), $r<<c_o$. Equivalent statements can be made concerning a single PE implementation of t-Coll, except that a slight modification is needed to enable a single PE to accept tokens within a range of activity numbers rather than just

one particular activity number.

We note one more point regarding these two macros before giving the example programs. It may be desirable to have n, the number of tokens to be produced or accepted by t-Dist or t-Coll, as an input token to the macros. This can be accomplished by changing the interpretation rules of the macros in Figures A.2c and A.3c to include a dummy token (similar to that for gates and merges [AG75]) to indicate the range of activity numbers to be produced (in the case of t-Dist) and the range of activity numbers to be accepted (in the case of t-Coll).

A.2 Sample Programs

Example 1 - Matrix Multiply Subroutine

A pseudo Algol program for multiplying two matrices A and B is given in Figure A.4. This program has been hand translated into the DDF language and is given in Figure A.5. There are eight loops connected with eight different merge operators. Loops 1, 2, and 3 are nested and generate the proper value of k. Every time the select operator is executed a token is needed on all its input arcs. Therefore n^2 copies of each of indices i and j must be produced; this is accomplished by loops 4, 5, and 6. Loop 7 constructs the sum s and loop 8 constructs the new matrix C by storing s in the proper place.

We have not shown the loops for the input n to the predicate statements, nor have we shown the loops for matrix names A and B for the select statements. These loops can be drawn in a

straightforward manner, however, they are not necessarily needed since any parameter that remains constant during one activiation of a procedure could be substituted just before the activation of that procedure.

A DDF program using t-Dist and t-Coll is shown in Figure A.6. As already discussed, the execution time of these macros under the non-feedback interpreter is roughly $c_o + rn$. For an arbitrary PE structure it is safe to assume that r will be very much smaller than the time needed to multiply two numbers together. Let us assume that it takes c_f units of time to carry out each of the statements 7, 8, and 9. Given a sufficient number of PEs (i.e., $> n^3$) the program in Figure A.6 will take $5c_o+2c_f+4rn+rn^2 \simeq c+rn^2$ units of time (where c is some constant $>>r$). On computers without parallel operations (i.e., without pipelining, etc.) the matrix multiply will generally take $c'+c_fn^3$ units of time. If $r<<c_f$ then $c+rn^2 << c'+c_fn^3$ for large n. This is understandable in view of the fact that under the non-feedback interpreter all n^3 multiplications can be executed simultaneously, and the time to execute the program is proportional to the time needed to collect the results of the multiplications.

If the number of available PEs is smaller then n^3, then the execution time would be proportionately longer.

Example 2 - Subroutine Sort

This procedure sorts an array of n numbers according to the quicksort method. A pseudo Algol program for quicksort is given in Figure A.7. The procedure is recursive. In order to simplify writing the procedure in DDF we define a function called Comparator shown in Figure A.8. A DDF procedure Sort(a,n) using t-Dist, t-Coll, and Comparator is given in Figure A.9. If n is greater than 2, the procedure simply selects the middle element a_m in array a and constructs two new arrays b and c, where b contains all the elements of array a less than or equal to a_m, and c contains all the elements of array a greater than a_m. The dimensions of arrays b and c are identical, and one less than the dimension of a. If the i^{th} element of array a is stored in the i^{th} position in array b (i.e., a_i a_m) then a null is stored in the i^{th} position in array c. Similarly, for every non-null entry in c there is a corresponding null entry in b. Box 3 in Figure A.9 counts the number of non-null entries in b. Boxes 1 and 2 (i.e., t-Coll with function f=Build) construct new arrays where the null entries have been deleted. Then both arrays b and c are sorted simultaneously by parallel recursive calls to procedure Sort. The final result is produced by concatenating the sorted array b, element a_m, and the sorted array c.

Again, if we assume that the statements t-Dist and t-Coll take c_o+rn time, the Comparator takes c_f units of time (where $r<c_f$), and more than n PEs are available, then one invocation of procedure Sort will require $c+2rn$ time besides the time taken by the apply operators. On an average the parallel recursive calls will be nested $O(\log_2 n)$ deep. In the worst case

there can be n nested calls to procedure Sort. Therefore, if sufficient numbers of PEs are available, the program will take on an average $(c+2rn)\log_2 n$ time. In the worst case time taken will be $(c+2rn)n$. For sequential computers these times would be $c_f n\log_2 n$ and $c_f n^2$, respectively, where $c_f >> r$.

These two examples illustrate the significant reductions we expect in the execution time of most algorithms. These reductions will occur automatically, when the algorithms are written in DDF and executed under the non-feedback interpreter.

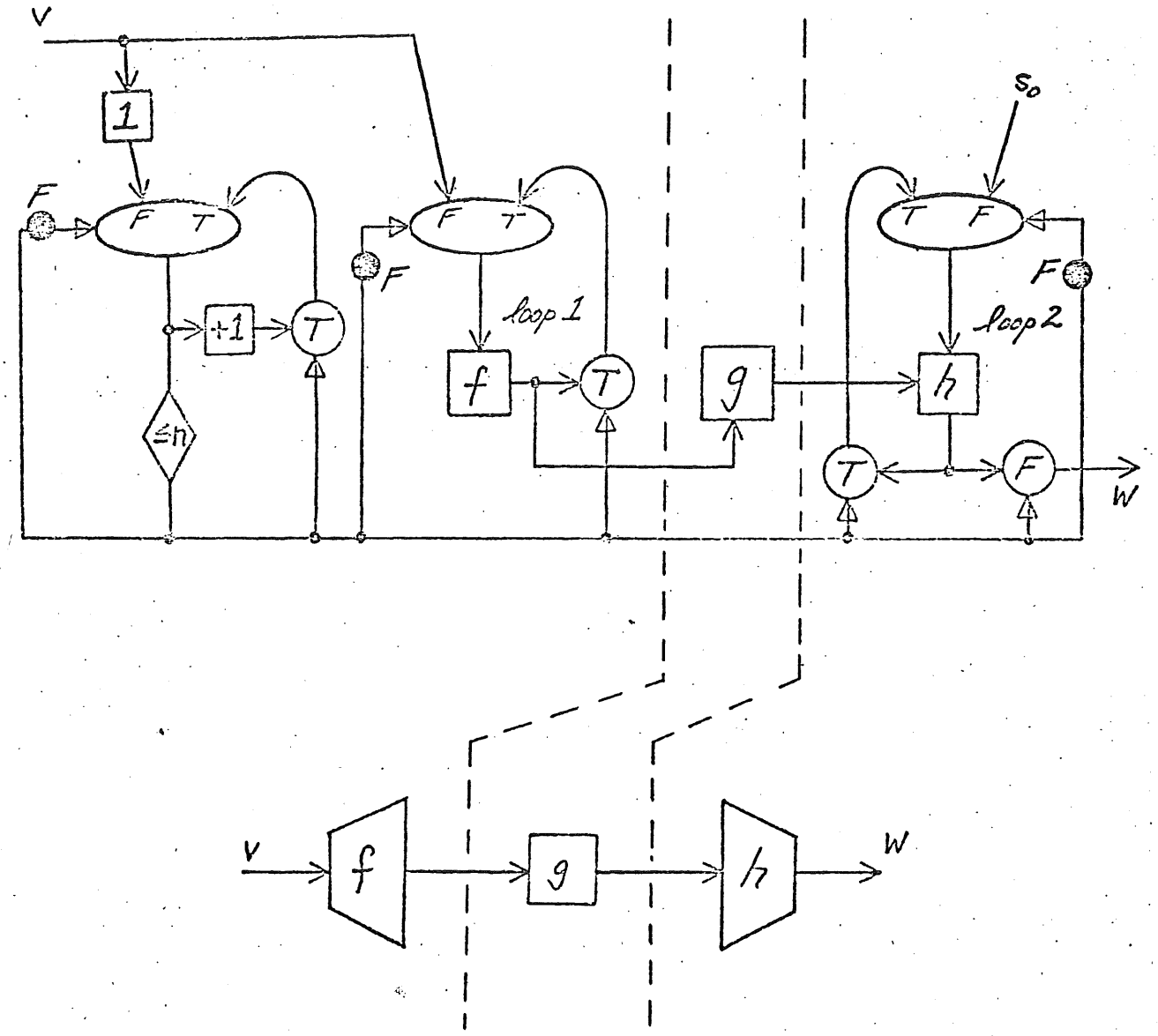


Figure A.1

A common program component in the upper portion of the figure, and its macro shorthand equivalent in the lower portion

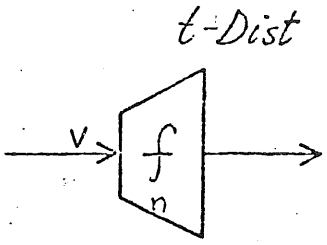


Figure A.2a
The t-Dist macro symbol

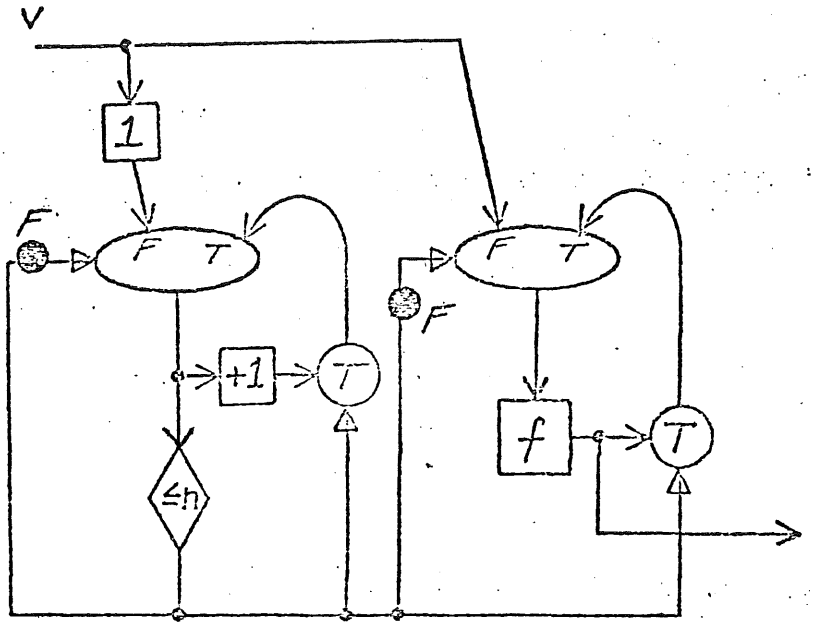


Figure A.2b
Data flow definition of t-Dist

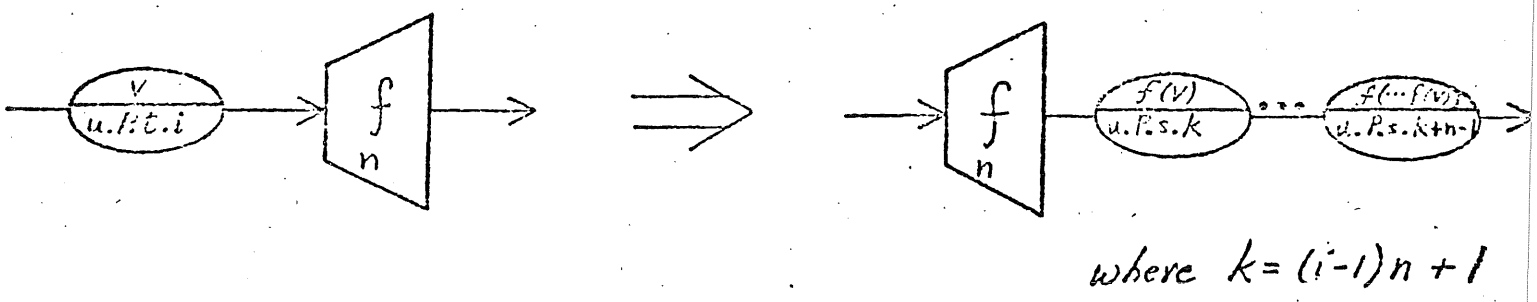


Figure A.2c
Snapshot of behavior of t-Dist

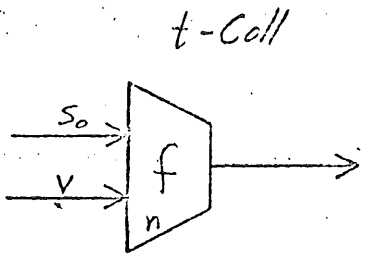


Figure A.3a

The t-Coll macro symbol

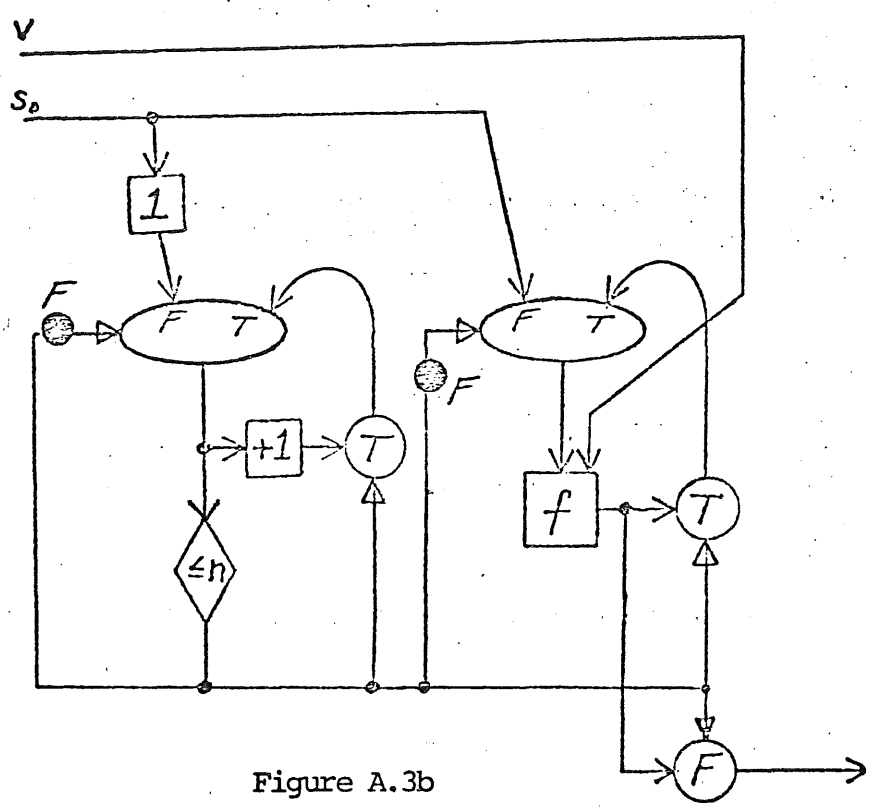
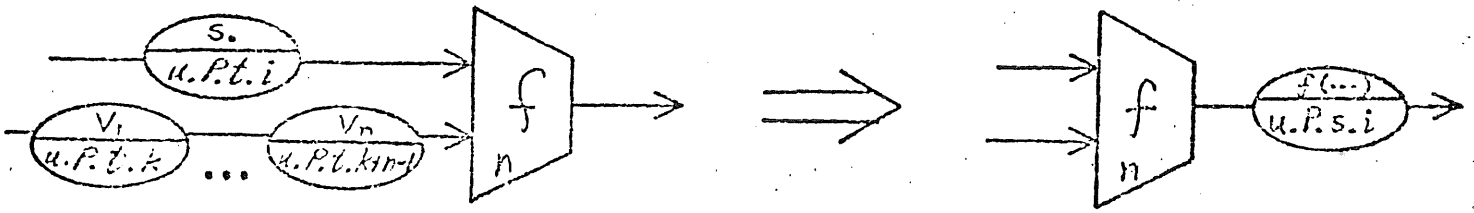


Figure A.3b

Data flow definition of t-Coll



where

$k = (i-1)n + 1$

Figure A.3c

Snapshot of behavior of t-Coll

```
Multiply (n,A,B,C):
    for i ← 1 to n do
        for j ← 1 to n do
            s ← 0;
            for k ← 1 to n do
                s ← s + A(i,k) * B(k,j)
            end
            C(i,j) ← s
        end
    end
```

Figure A.4

Pseudo-Algol nxn matrix multiply C=A*B

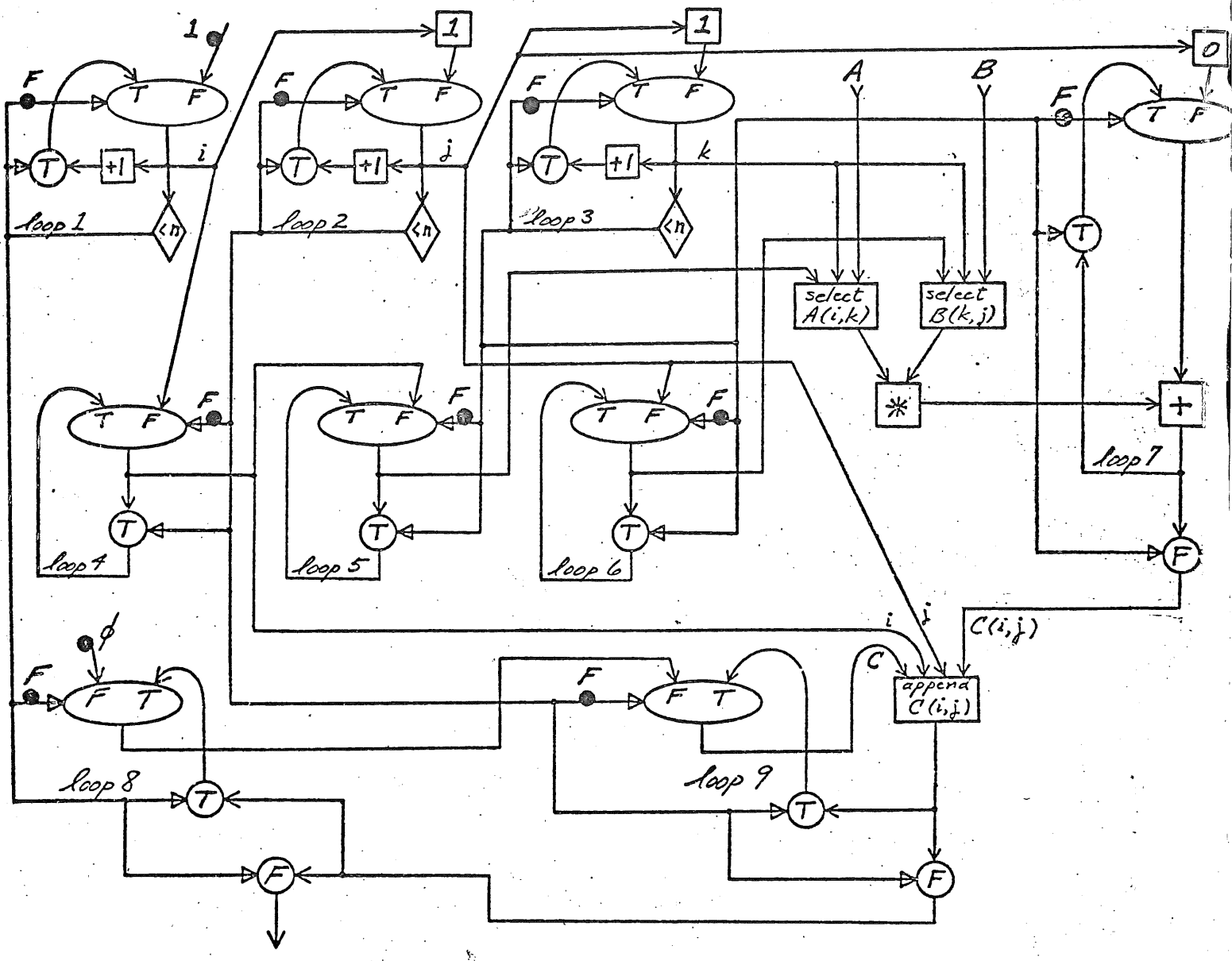


Figure A.5

First data flow matrix multiply program

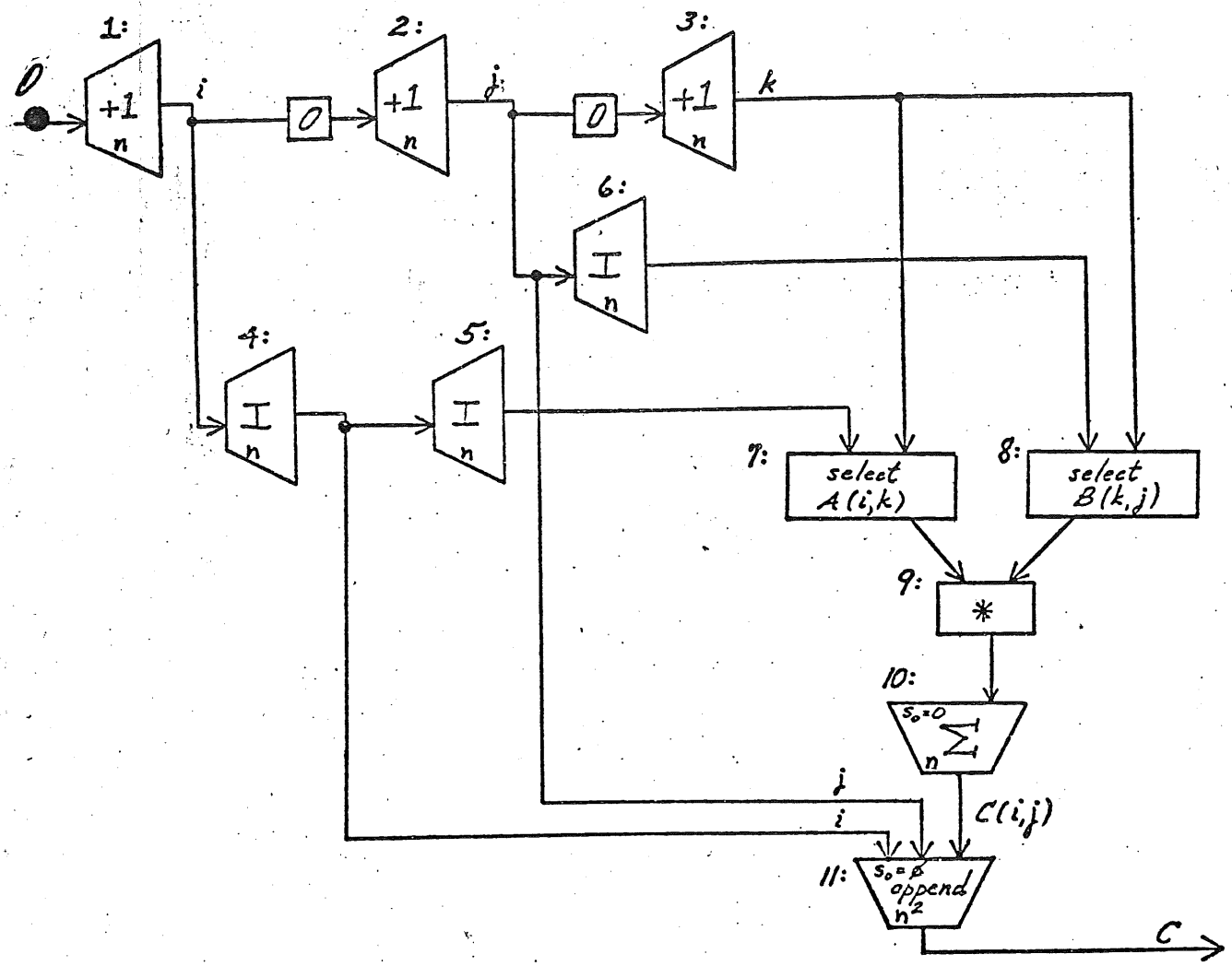


Figure A.6

Improved version of matrix multiply under non-feedback interpreter

```
sort (a,n):
    m ← ⌊n÷2⌋ ;
    k←0; j←0;
    for i←1 to n do
        if i≠m then [if a_i ≤ a_m
                        then [j←j+1; b_j←a_i]
                        else [k←k+1; c_k←a_i]]
                   else [ ]
    end
    sort(b,j) ; sort(c,k) ;
```

Figure A.7

Pseudo-Algol recursive program for quicksort

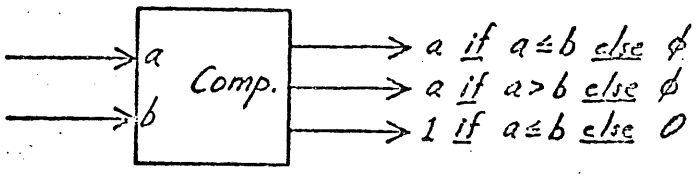


Figure A.8

The Comparator function

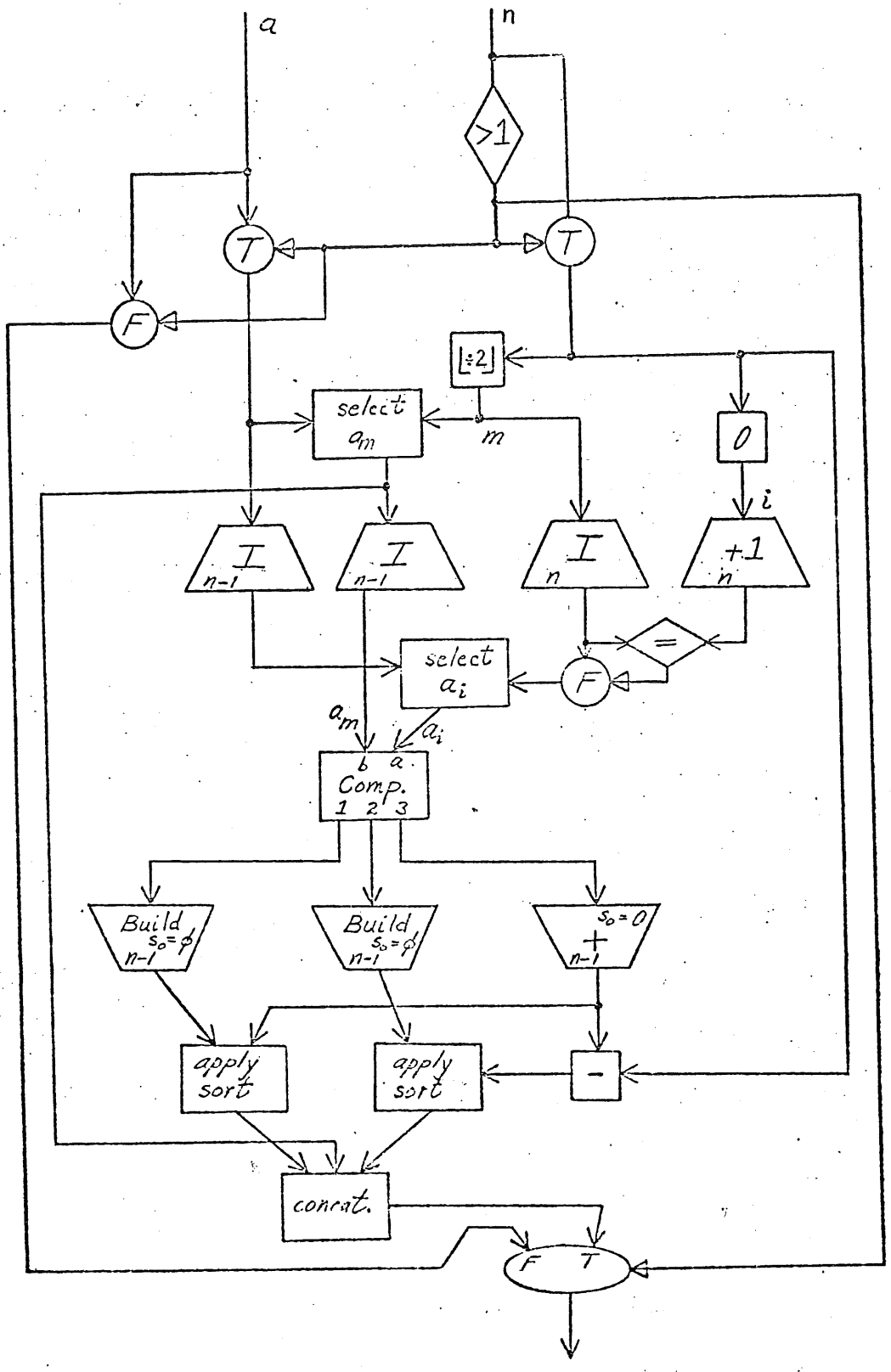


Figure A.9

Recursive quicksort to be executed by non-feedback interpreter

ACKNOWLEDGEMENTS

We wish to thank Professors Tim Standish and Fred Tonge for their constructive criticism during the preparation of this proposal, and to Mr. Wil Plouffe for his innumerable suggestions in the analysis of the algorithms of Appendix A. Many people, especially Messrs Larry Rowe, Paul Mockapetris and Professor Dave Farber of the Distributed Computing System Project have offered suggestions. We are also grateful to Professor Gerald Estrin for his comments and criticism during the early stages of this work. And, of course, thanks to Peg Gray for typing, and to our resident PDP-1Ø text expert Shirley Rasmussen for preparing several versions of this proposal.